FLORIDA STATE UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

RETARGETING AN ASSEMBLY OPTIMIZER FOR THE MIPS/SCALE ASSEMBLY LANGUAGE

By

ARTHUR KARAPATEAS

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

2021

Arthur Karapateas defended this project on July 30th, 2021.
The members of the supervisory committee were:

Dr. David Whalley
Professor Directing Project

Dr. Xiuwen Liu
Committee Member and Department Chair

Dr. Sonia Haiduc
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the project has been approved in accordance with university requirements.

TABLE OF CONTENTS

LIST OF FIGURES

…

ABSTRACT

An assembly optimizer takes in assembly code as input and produces improved and/or translated assembly code as output. In this project I took an existing assembly optimizer that was targeted for the SPARC assembly language and retargeted it to the MIPS/SCALE assembly language. To do this I also solved the problem of determining what registers are live at each point in the assembly program without performing any interprocedural analysis to allow for determining if the assembly optimizer properly handled instructions in a more efficient manner. Additionally, I expanded all pseudo assembly instructions that would generate two or more machine instructions so that there is a one-to-one correspondence between each assembly instruction and machine instruction, which allows for more effective instruction scheduling. I also provided the option to generate SCALE assembly, which has some slightly different features than the conventional MIPS assembly, to be used for further research projects. I extensively tested the MIPS assembly optimizer (ASOPT) and was able to process all MIPS assembly files produced by the gcc compiler when compiling the MiBench and SPEC 2006 benchmark suites. This included successfully translating each assembly instruction through the assembly optimizer with no assembly time errors while also verifying that no instructions with unnecessary dead assignments were introduced by the assembly optimizer.

CHAPTER 1

INTRODUCTION

1.1 Motivation

An assembly optimizer takes in assembly code as input and produces optimized or translated assembly code as output. An assembly optimizer abstracted outside of a compiler provides many benefits to a conventional compiler, such as gcc. One primary benefit is it allows for us to perform additional low-level optimizations or translation at the assembly level without having to actually modify the complex implementation of a compiler, gcc for example is several hundred thousand lines of code. At the same time, it allows us to take advantage of the optimizations and code generation automatically performed by the compiler producing the initial assembly code.

This Masters project is part of the SCALE research project. SCALE stands for Statically Controlled, Asynchronous Lane Execution and this research is funded by NSF. The SCALE project will have several assembly language instructions that will support a variety of new architectural features. This retargeted MIPS assembly optimizer will be used on future projects involving various SCALE ISAs, and this assembly optimizer can also function as a learning tool for implementing various low-level assembly optimizations on MIPS assembly code.

This project involved porting an assembly optimizer used in the COP6622 Advanced Topics in Compilation course. The assembly optimizer used in that course was for the SPARC assembly language. In order for the SCALE project to use the assembly optimizer it was necessary to retarget it to support all MIPS instructions generated by the gcc compiler. Additionally, I provided the option to translate some MIPS assembly instructions to SCALE-specific instructions. One of the significant challenges of this project was determining what registers are live at each point in the assembly program without performing interprocedural analysis. This document explains the process and implementation taken to retarget the SPARC assembly optimizer previously written by Dr. Whalley to the MIPS assembly language.

1.2 Overview of MIPS Assembly Optimizer

This project began with an initial SPARC assembly optimizer written by Dr. Whalley. The SPARC assembly optimizer was split up into machine independent code stored in a "lib" directory and SPARC dependent code was placed into a "sparc" directory. I had the task of creating the code in the mips directory that contains the mips dependent code so the assembly optimizer could analyze and process mips assembly instructions. Beginning with this initial assembly optimizer the task at hand was to support all MIPS assembly instructions that would be encountered in a gcc generated assembly file. I then ran the assembly file along with other necessary generated files through the assembly optimizer. As I encountered unhandled assembly instructions I would add support for them manually when necessary rather than going through the MIPS assembly language manual and adding every possible MIPS instruction. The means for generating the MIPS gkd file and objdump file shown in **Figure 1** will be explained in later sections of the document**.**

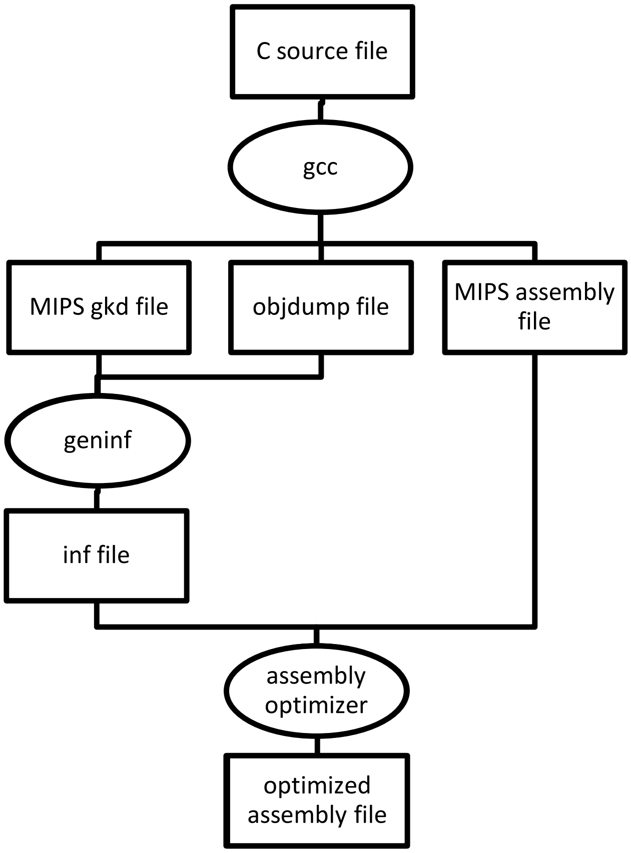


Figure 1: Generating Optimized Assembly File Process

1.2.1 Processing Assembly Instructions

As assembly instructions were encountered that were unhandled, I would add each of these to an *instinfo* struct that shows the fields for each type of instruction. Each of these instructions and key details about them are stored within the *instinfo* struct which is shown below. These fields allow the assembly optimizer to properly analyze and manipulate each assembly instruction.

```
        struct instinfo {
           char *mneumonic;            /* mneumonic of instruction                 */
           enum insttype type;         /* instruction class                        */
           int numargs;                /* number of arguments                      */
           int numdstregs;             /* number consecutive registers associated with destination*/
           int numsrcregs;             /* number consecutive registers associated with source*/
           int latency;                /* number of cycles to produce value        */
           int setscc;                 /* condition codes set?                     */
           int lastsrccanbeconst;      /* can last src operand be a constant?      */
           int datatype;               /* datatype of instruction                  */
           int lanes;                  /* lanes in which instruction can execute   */
};
```

Figure 2: Structure for Storing Assembly Instructions

The *insttypes* is an array of *instinfo* structs that is interlaid to contain proper values for each instruction. Provided below is an example of some of the instructions added to this structure. As each new MIPS instructions is encountered, we would add the instruction to the structure.

```
struct instinfo insttypes[] = {
 {"addiu",     ARITH_INST,  3, 1, 1, 1, FALSE, TRUE,  INT_TYPE,    GENINT_LANE},
 {"addu",      ARITH_INST,  3, 1, 1, 1, FALSE, FALSE, INT_TYPE,    GENINT_LANE},
 {"add.d",     ARITH_INST,  3, 2, 2, 1, FALSE, FALSE, DOUBLE_TYPE, GENFP_LANE},
 {"add.s",     ARITH_INST,  3, 1, 1, 1, FALSE, FALSE, FLOAT_TYPE,  GENFP_LANE},
 {"and",       ARITH_INST,  3, 1, 1, 1, FALSE, FALSE, INT_TYPE,    GENINT_LANE},
 {"andi",      ARITH_INST,  3, 1, 1, 1, FALSE, TRUE,  INT_TYPE,    GENINT_LANE},
 {"b",         JUMP_INST,   1, 0, 0, 1, FALSE, FALSE, 0,           TOC_LANE},
 {"beq",       BRANCH_INST, 3, 0, 1, 1, FALSE, FALSE, INT_TYPE,    TOC_LANE},
 {"break",     BREAK_INST,  1, 0, 0, 1, FALSE, TRUE,  0,           TOC_LANE},
 …
 {"c.eq.s",    CMP_INST,    2, 0, 1, 1, TRUE,  FALSE, FLOAT_TYPE,  GENFP_LANE},
 {"c.eq.d",    CMP_INST,    2, 0, 2, 1, TRUE,  FALSE, DOUBLE_TYPE, GENFP_LANE},
 …
 {"j",         JUMP_INST,   1, 0, 0, 1, FALSE, FALSE, 0,           TOC_LANE},
 {"jal",       CALL_INST,   1, 0, 0, 1, FALSE, FALSE, 0,           TOC_LANE},
 {"jalr",      CALL_INST,   1, 0, 1, 1, FALSE, FALSE, 0,           TOC_LANE},
 {"jr",        RETURN_INST, 1, 0, 1, 1, FALSE, FALSE, 0,           TOC_LANE},
 {"la",        ARITH_INST,  2, 1, 0, 1, FALSE, FALSE, INT_TYPE,    GENINT_LANE},
 …
 {"lw",        LOAD_INST,   2, 1, 1, 1, FALSE, FALSE, INT_TYPE,    LOAD_LANE},
 …
 {"sw",        STORE_INST,  2, 0, 1, 1, FALSE, FALSE, INT_TYPE,    STORE_LANE},
 {"",          0,           0, 0, 0, 0,     0,      0,        0,              0}

};
```

Figure 3: Organization of Assembly Instructions Handled

CHAPTER 2

METHODS

2.1 Generating Additional Files to Determine Live Register Information

In order to run an assembly file through the assembly optimizer we must first generate 3 files associated with the assembly file, which are the RTL dump file *(*.s.gkd*), the object dump file (**.objdump*), and the **.inf* file. In this section we will explain the need for each file generated and how it is used by the assembly optimizer to help determine which registers are live at each assembly instruction to properly perform analysis and transformations on the assembly. Once the associated **.inf* file is created that corresponds with the MIPS assembly file, we can finally begin processing the assembly file through the assembly optimizer.

2.1.1 Generating RTL Dump File

A **.s.gkd* file associated with a MIPS assembly file contains the RTL dump information of our assembly file. This file is necessary for determining which registers are passed as arguments to each function call so that we can properly determine any implicitly used registers used by a call instruction. This information is crucial for determining live register information. The flags to produce the assembly file and the **.s.gkd* file while running gcc are "-fdump-final-insns -S".

Figure 4 shows an example section of a **.s.gkd* file. As functions are encountered in the **.s.gkd* file via the keyword "**Function**" preceded by **";;"** within the RTL dump we extract the function name. This name will be stored and later searched for within the **.objdump* file when it is processed to determine the return type of the function. Next, we search for each "**call_insn**" following the extracted function name. The RTL dump is used for determining calls to functions by searching for the key word "**call_insn**" and extracting the information following it. We also extract which register values are used by the function call by explicitly searching for the keyword **"(use (reg**" and then extracting the arguments following the keyword. Via this information we can determine which registers are passed as arguments to the function so we can determine the implicit uses of the function call when performing proper live variable analysis.

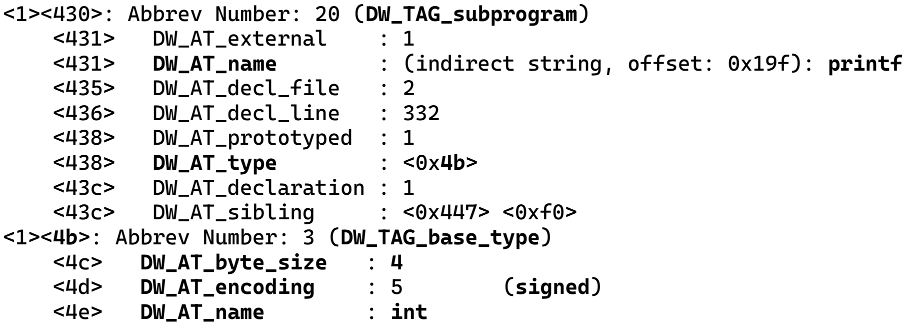
```
;; Function printlist2 (printlist2, funcdef_no=6, decl_uid=1298, symbol_order=6)

(note# 0 0 NOTE_INSN_DELETED)
(note# 0 0 [bb 2] NOTE_INSN_BASIC_BLOCK)
(insn/f# 0 0 (set (reg/f:SI 29 $sp)
        (plus:SI (reg/f:SI 29 $sp)
            (const_int -24 [0xffffffffffffffe8])))
/home/karapate/infogen/test/sort/sort.c:71# {*addsi3}
     (nil))
(insn# 0 0 (clobber (mem/c:BLK (reg/f:SI 29 $sp) [  A8]))
/home/karapate/infogen/test/sort/sort.c:71#
     (nil))
(insn/f# 0 0 (set (mem/c:SI (plus:SI (reg/f:SI 29 $sp)
                (const_int 20 [0x14])) [  S4 A32])
        (reg:SI 31 $31)) /home/karapate/infogen/test/sort/sort.c:71# {*movsi_internal}
     (expr_list:REG_DEAD (reg:SI 31 $31)
        (expr_list:REG_FRAME_RELATED_EXPR (set/f (mem/c:SI (plus:SI (reg/f:SI 29 $sp)
                        (const_int 20 [0x14])) [  S4 A32])
                (reg:SI 31 $31))
            (nil))))
(code_label 33 0 0 33 "" [1 uses])
(note# 0 0 [bb 7] NOTE_INSN_BASIC_BLOCK)
(insn# 0 0 (set (reg:SI 4 $4)
        (symbol_ref/f:SI ("*$LC2") [flags 0x2] <var_decl # *$LC2>))
/home/karapate/infogen/test/sort/sort.c:77# {*movsi_internal}
     (nil))
…
(call_insn# 0 0 (parallel [
            (set (reg:SI 2 $2)
                (call (mem:SI (symbol_ref:SI ("printf") [flags 0x41] <function_decl #
printf>) [ printf S4 A32])
                    (const_int 16 [0x10])))
            (clobber (reg:SI 31 $31))
        ]) /home/karapate/infogen/test/sort/sort.c:77# {call_value_internal}
     (expr_list:REG_DEAD (reg:SI 5 $5)
        (expr_list:REG_DEAD (reg:SI 4 $4)
            (expr_list:REG_UNUSED (reg:SI 2 $2)
                (nil))))
    (expr_list:SI (use (reg:SI 5 $5))
        (expr_list:SI (use (reg:SI 4 $4))
            (nil))))
```

Figure 4: Example Portion of sort.s.gkd File

2.1.2 Generating an Object Dump File

We compiled the .c file with the -g -c options to produce symbolic debugging information in the file. We use the "**objdump**" command to produce a parseable *.*objdump* file. Producing a *.*objdump* file associated with our assembly file is necessary for determining the return type of the function encountered within the assembly. This information is then used to determine what register is implicitly used by return instructions so that we can determine which register is live when returning from the function. In the command to generate the *.*objdump* file below, "$base" is the base name of the assembly file. The command to generate .o file: "gcc -g -c $base.c ". The command to generate *.*objdump* file is "objdump -Wi $base.o > $base.objdump".



```
<1><430>: Abbrev Number: 20 (DW_TAG_subprogram)
    <431>   DW_AT_external    : 1
    <431>   DW_AT_name        : (indirect string, offset: 0x19f): printf
    <435>   DW_AT_decl_file   : 2
    <436>   DW_AT_decl_line   : 332
    <438>   DW_AT_prototyped  : 1
    <438>   DW_AT_type        : <0x4b>
    <43c>   DW_AT_declaration : 1
    <43c>   DW_AT_sibling     : <0x447> <0xf0>
<1><4b>: Abbrev Number: 3 (DW_TAG_base_type)
    <4c>   DW_AT_byte_size   : 4
    <4d>   DW_AT_encoding    : 5       (signed)
    <4e>   DW_AT_name        : int
```

Figure 5: Example Portion of sort.objdump File

Figure 5 shows an example portion of a *.*objdump* file. After encountering the function name from the *.*s.gkd* we then begin our search in the .objdump file for that specific function name. We first validate that the section the name is located in is the section denoted by the keyword DW_TAG_subprogram, this denotes that the following section provides details pertaining to a function. The next keyword we search for is DW_AT_name, the string furthest to the right denotes the name of the function for this section and if this function name matches the function name we encountered within the *.*s.gkd* we then know we can continue within this section to determine the return type for the function. To determine the return type, we then search for the first DW_AT_type keyword and the number between the "< >" in this section denotes where in the objdump we can find the return type for the function of this name, as every line is denoted with a hexadecimal number between "< >". If there is no DW_AT_type keyword before the next section is encountered, that function has no return type (void return). In the example above the type is located at “<0x4b>” and upon searching for the line denoted by “<4b>” we find within a section denoted with the tag “DW_TAG_base_type”. In this section we are provided important details such as the byte_size, the name of the type, and whether it is signed or unsigned encoding. Other more difficult scenarios that were handled were when a return type was a pointer type, structure type, typedef, or enumeration. In order to handle these scenarios further recursion was necessary to make sure that the deepest nested primitive type of the data being returned was accessed, as this is the information that is necessary when performing live variable analysis on the registers.

2.1.3 Generating the .inf file

Using the **.s.gkd* file and **.objdump* files as input into a *geninf* program we can then generate a **.inf* file. Generating a **.inf* file became necessary for the SCALE project using the assembly optimizer in order to support hand-written assembly files. The **.s.gkd* files could not be generated automatically when there was no corresponding *.c file to run through the gcc compiler. For a handwritten assembly file, we generate a **.inf* file by hand as the format is much simpler than a **.s.gkd* and **.objdump* file. **Figure 6** shows an example of a **.inf* file. The *geninf* program, written by Dr. Whalley, was created using much of the code I implemented for processing and accessing the live variable information from the **.s.gkd* files and the **.objdump* files described above. Before running the *geninf* program, the **.s.gkd* and the **.objdump* files associated with the assembly file must be generated and located within the same directory as the assembly file as they are used to generate the **.inf* file. The command to use the *geninf* program to generate the *.inf file is "geninf $base.s". The process for generating a **.inf* file is illustrated in **Figure 1.**

```
function myabs int
function myrand int
calls myabs $4
function swap void
function choose_pivot int
function quicksort void
calls choose_pivot $4 $5
calls swap $4 $5
calls swap $4 $5
calls swap $4 $5
calls quicksort $4 $5 $6
calls quicksort $4 $5 $6
function printlist void
calls printf $4 $5
calls printf $4
function printlist2 void
calls printf $4 $5
function main int
calls myrand
calls printf $4
calls printlist2 $4 $5
calls quicksort $4 $5 $6
calls printf $4
calls printlist2 $4 $5
calls printf $4
```

Figure 6: Example sort.inf File

2.2 Generating Optimized MIPS Assembly Output

For each assembly line in the generated MIPS assembly, the function *setupinstinfo* was called in which the assembly line would first have its items (arguments) associated with an instruction (done in *makeinstitems* function). This process involved stripping out any initial blank characters, checking if a mnemonic was encountered, checking for a comma (denoting the encountering of items next), and replacing the space before the arguments with a tab if there is no tab already (for the proper alignment of the generated MIPS assembly). Next all commas separating each argument were replaced with tabs and the arguments were then captured as well for later usage when producing the output.

The next step in the *setupinstinfo* function is to classify the instruction in the current assembly line (done in *classifyinst* function). This is the step in which we search through the list of known instructions previously discussed and shown in **Figure 3**. The information provided in that list of instruction structures denotes the assigning of the type for the instruction on the current assembly line, the number of destination registers, and the number of source registers. Here we also handle special types that are unique to the mnemonic being used in conjunction with its arguments. For example, if encounter a jump instruction and its argument is *$31* (*j $31)* this denotes that the type for this instruction should not be categorized as a jump instruction but rather a RETURN_INST. If we encountered a JUMP_INST, the argument contains a "$" and it is not *$31* or *$L* it is a CALLRETURN_INST. If we encountered a JUMP_INST, and the argument does not contain a "$" then the type is an INDJUMP_INST.

The final step in the *setupinstinfo* function is to set the bits associated with the registers set and used in the instruction (done in the *setsuses* function). This will assist us when performing live variable analysis. First, we initialize the variable state to be NULL for the sets, uses, and implicit uses fields. Next, we check if the registers being read in are in the proper format for the MIPS instruction. Finally, we assign the register sets and uses based on the type of instruction being handled. The different types encountered that needed to be handled for each MIPS assembly line included: ARITH_INST, CONV_INST, MOV_INST, SPMOV_INST, PRED_MV_INST, BRANCH_INST, CALL_INST, CALLRETURN_INST, CMP_INST, BREAK_INST, JUMP_INST, JUMPTABLE_LINE, INDJUMP_INST, NOP_INST, LOAD_INST, STORE_INST, RESTORE_INST, SAVE_INST, RETURN_INST, COMMENT_LINE, and DEFINE_LINE. When processing CALLRETURN_INST it was necessary to also set scratch registers since the called function could overwrite all of them. **Figure 7** shows the function to initialize the variables state to indicate all scratch registers. We use relevant information in the generated *.*inf* file associated with each assembly line that is a function call to determine the uses for it, since the *.*inf* file contains information about what registers each function receives as arguments and are thus live entering the function.

```
/*
 * scratchinit - initialize scratch registers in mips
 */
void scratchinit(varstate v) {
   v[0] = 0x03003FFE; //scratch registers in mips are 1-13 and 24-25
   v[1] = 0x000FFFFF; //fp scratch are 0-19
   v[2] = 0x00000007; //for hi, lo, cc
   v[3] = 0x03003FFE; //scratch registers in mips are 1-13 and 24-25
   v[4] = 0xFFFF0000; //predicate registers
   v[5] = 0x00000000; //variables
}
```

Figure 7: Scratch Registers Assigned in scratchinit Function.

Finally, after completing the processing of the instruction for the assembly line and performing these steps for each assembly line in the file, block by block, while ignoring directives and removing assembly comment lines, we are then able to generate useful output for someone using the assembly optimizer. All comments beginning with the “#” symbol were inserted into the output of the processed MIPS assembly file with relevant data pertaining to each block of assembly, loops that occur, and what the sets, uses, and dead registers are at each line. **Figure 8** shows example assembly optimizer output for a simple function.

```
        .text
        .align  2
        .globl  myabs
        .set    nomips16
        .set    nomicromips
        .type   myabs, @function
# block 1
# preds:
# succs: 2 3
#  doms: 1
#   ins=$4:$31:
#  outs=$2:$4:$31:
myabs:
        move    $2,$4               # sets=$2:     uses=$4:      deads=
        slt     $1,$4,$0            # sets=$1:     uses=$0:$4:   deads=
        beqz    $1,$L2              # sets=        uses=$1:      deads=$1:
# block 2
# preds: 1
# succs: 3
#  doms: 1 2
#   ins=$4:$31:
#  outs=$2:$31:
        subu    $2,$0,$4            # sets=$2:     uses=$0:$4:   deads=$4:
# block 3
# preds: 1 2
# succs:
#  doms: 1 3
#   ins=$2:$31:
#  outs=
$L2:
        j       $31                 # sets=        uses=$2:$31:  deads=$2:$31:
```

Figure 8: Example Generated Output from Processing sort.s File.

2.3 Implicitly Used and Set Registers and Live Variable Analysis

When processing the MIPS assembly through the assembly optimizer it is important to properly determine which registers are live at each point in the program to properly perform certain compiler optimizations such as “dead assignment elimination”. To do this performing live variable analysis on the dataflow is necessary. A register is considered live at a specific point if it contains a value that is used in the future, for example being read before being overwritten. While performing the final step of the *setupinstinfo* function, the sets and uses were explicitly handled for various instruction types. Implicit uses were determined when handling returns. This involved initializing the callee save and scratch register sets to their default values. Next, we needed to take the union of all registers accessed and then subtract the scratch registers from this set of the union of all registers. This determines the callee save registers that are accessed. From this we can set the implicit uses for each return instruction. In the epilogue before a return instruction callee-save register values are returned (loaded from the scratch). The return instructions need to have these callee-save registers as implicit uses so dead assignment elimination does not improperly remove these loads. Implicit sets were handled explicitly for scenarios such as CALLRETURN_INST where the implicit sets were the initial scratch registers and the $sp register. For implicit sets we also needed to handle scenarios where a function being called appeared earlier in the file and if this did occur the implicit sets needed to include the sets from that function earlier in the file.

```
void implicituses(struct bblk *tblk) {
   struct assemline *ptr;
   varstate csets;   //callee save sets
   varstate scregs;  //scratch registers

   /* initialize the callee save and scratch register sets */
   varinit(csets);
   scratchinit(scregs);

   /* take the union of all registers accessed */
   for (tblk = top; tblk; tblk = tblk->down)
      for (ptr = tblk->lines; ptr; ptr = ptr->next)
         unionvar(csets, csets, ptr->sets);

   /* subtract the scratch registers from this set to determine
      the callee save registers that are accessed */
   minusvar(csets, csets, scregs);

   /* set the uses for each return instruction with the callee save regs */
   for (tblk = top; tblk; tblk = tblk->down)
      for (ptr = tblk->lines; ptr; ptr = ptr->next)
         if (is_return(ptr))
            unionvar(ptr->uses, ptr->uses, csets);
}
```

Figure 9: Function to Update Implicit Uses in Return Instructions

2.4 Expanding Pseudo Instructions

The next significant task to implement was traversing through the MIPS assembly and expanding pseudo instructions as they were encountered. This was done by traversing each block and checking each line for specific instructions that needed to be expanded. These lines would be replaced accordingly. **Figures 10** and **11** show examples of expanding a pseudo instruction into multiple instructions. In the examples below, numbers followed by the “$” symbol represent registers in MIPS assembly.

```
l.d $f1, 2($2)
=>
lwc1 $f1, 2($2)
lwc1 $f2, 6($2)
```

Figure 10: Example Expanding l.d or s.d Instruction

```
syscall
=>
addiu     $1,$sp,16
setkri    $4,$1,0
syscall__ 0,0
```

Figure 11: Example Expanding syscall Instruction

When dealing with a load, store, or load address of a global name, we expanded the construction of the address using these two instructions originally.

```
la $2,name
=>
lui $1,%hi(name)
ori $2,$1,%lo(name)
```

This was later altered to be expanded to the example below that are SCALE specific instructions.

```
la $2,name
=>
lalui $2,name
laori $2,$2,name
```

Doing this replaced the *%hi* and *%lo* with the new assembly mnemonic names (*lalui* and *laori*). The other change is since we are updating the destination register (*$2* above), we know *$2* can be used in both the *lalui* and *laori* instructions. The same was done for load and store instructions, using *lalui* and *laori* before the actual load or store instruction. The *$1* MIPS register is reserved for the MIPS assembler to expand pseudo instructions. We made use *$1* as we expanded all pseudo instructions. Loads did not need to use the *$1* register. For stores we did need to use *$1* register as a store does not update a register.

All *li* instructions outside of the immediate field range of -32768…32767 were also expanded. When expanding li instructions we did not need to use $1 register as we can just use the destination register. We also did not use *%hi* and *%lo* when expanding *li* instructions. We extracted the upper portion of the constant by doing a bitwise and operation with 0xffff0000 and putting the result in an unsigned variable and doing a right shift by 16. We then extracted the lower portion by doing a bitwise and operation with 0xffff and putting the result in an unsigned variable. **Figure 12** shows an example of expanding an li instruction.

```
li $2,32770 //32770 is 0x10010 in hex
=>
lui $2,1
ori $2,$2,2
```

Figure 12: Example Expanding li Outside of Immediate Field Range

Any load or store instructions that used displacements were also expanded to produce legal SCALE loads and stores. Therefore, this version of the MIPS only supports register deferred addressing mode for memory references and all loads and stores that have a nonzero displacement are expanded into two instructions where the first instruction calculates the effective address. All loads and stores that reference a static or global address directly are also expanded into three instructions. The first instruction is a *lalui* (load address load upper immediate) macro. The next instruction is *lalli* (load address load lower immediate) macro. *lalui* assigns the upper 16 bits of the static or global address and *lalli* assigns the lower 16 bits of the same static or global address. In this example below the *lw* instructions got expanded as the item in the second field contains a '('. If the item in the second field for the instruction contains a '(' symbol it means that the memory address is using a displacement or register deferred addressing mode. **Figure 13** shows some examples of expanding loads, stores and load address pseudo instructions.

```
lw $3,4($7)       sw $4,8($2)       lw $5,g1          la $6,g2
=>                =>                =>                =>
addiu $3,$7,4     addiu $1,$2,8     lalui $5,g1       lalui $6,g2
lw $3,($3)        sw $4,($1)        lalli $5,g1       lalli $6,g2
                                    lw $5,($5)
```

Figure 13: Examples of Expanding Loads, Stores and Load Address Macros

Lastly all branch pseudo instructions were replaced with expansions. This includes: *bgtz*, *bgez*, *blez*, *bltz*, *beq*, and *bne*. Each of these instructions was replaced with their associated equivalent expansion of the instructions *slt* and *bnez*, *slt* and *beqz*, or just *bnez*/*beqz*. **Figure 14** shows some examples converting MIPS branches to their SCALE equivalent versions.

```
beq  $3,$0,L1   bne  $3,$0,L1     beq  $4,$5,L1     bne  $4,$5,L1
=>              =>                =>                =>
beqz $3,L1      bnez $3,L1        seq  $1,$4,$5     seq  $1,$4,$5
                                  bnez $1,L1        beqz $1,L1
---------------------------------------------------------------------
bgtz $3,L1      bgez $3,L1        blez $3,L1        bltz $3,L1
=>              =>                =>                =>
slt  $1,$0,$3   slt $1,$3,$0      slt  $1,$0,$3     slt  $1,$3,$0
bnez $1,L1      beqz $1,L1        beqz $1,L1        bnez $1,L1
```

Figure 14: Examples of Converting Branches to Use bnez and beqz Instructions

2.5 Testing for Properly Handling Instructions and Debugging Process

To test proper handling of instructions and functionality of the MIPS assembly optimizer we began running the MIPS assembly optimizer through very simple benchmarks and progressively introduced it to more difficult benchmarks. The first batch of benchmarks were the *infogen/test* benchmarks. The next benchmarks that were handled were the MiBench benchmark suite setup for VPO and SimpleScalar. The last benchmarks we handled and tested against were the SPEC 2006 FLOAT and INTEGER benchmarks. In total there were 26 assembly files in the *infogen/test* benchmarks, 140 assembly files in the MiBench benchmark suite, and 1941 assembly files in the SPEC 2006 benchmarks. Bash scripts were also written for each test suite to automate the testing process and to test the functionality of previously passed benchmark tests as changes were made to support new benchmarks. **Figure 15** shows one of the bash scripts written to invoke other bash scripts for each benchmark test. As some of benchmarks required specific compilation settings to be built and run properly there needed to be different scripts written for each benchmark test. The output from each test script was saved away in a file that was searched through for error messages that were reported and needed to be resolved to pass the benchmark test.

```
#benchmarks in Test directory
./run_test.sh &> ./outputs_benchmarks/out_test.txt

#benchmarks in MiBench
./run_automotive.sh &> ./outputs_benchmarks/out_automotive.txt
./run_consumer.sh &> ./outputs_benchmarks/out_consumer.txt
./run_network.sh &> ./outputs_benchmarks/out_network.txt
./run_office.sh &> ./outputs_benchmarks/out_office.txt
./run_security.sh &> ./outputs_benchmarks/out_security.txt
./run_telecomm.sh &> ./outputs_benchmarks/out_telecomm.txt

#benchmarks in SPEC Float
./run_SPEC_FLOAT.sh &> ./outputs_benchmarks/out_SPEC_FLOAT.txt

#benchmarks in SPEC Integer
./run_SPEC_INTEGER.sh &> ./outputs_benchmarks/out_SPEC_INTEGER.txt

#check for errors, segmentation faults, and report transformations performed
cd ./outputs_benchmarks/
echo -e '---------------------------------------------------------------\n'
grep -r "compilation terminated" *
echo -e '---------------------------------------------------------------\n'
grep -r "Segmentation fault" *
echo -e '---------------------------------------------------------------\n'
grep -r "transformations applied by all optimization phases" *
echo -e '---------------------------------------------------------------\n'
grep -r "incorrect sets and uses calculated" *
echo -e '---------------------------------------------------------------\n'
grep -r "unexpected end of file after function" *
echo -e '---------------------------------------------------------------\n'
grep -r "encountered unhandled return type" *
echo -e '---------------------------------------------------------------\n'
grep -r "is unknown instruction" *
echo -e '---------------------------------------------------------------\n'
cd ..

exit 0
```

Figure 15:Example run_all_benchmarks.sh Bash Script File

To make sure the assembly in the benchmarks was being properly processed we relied on the gcc compiler to perform the compiler optimization “dead assignment elimination” on the generated MIPS assembly code by default. We would then run the gcc produced assembly code through our MIPS assembly optimizer and if dead assignment elimination was triggered by our MIPS assembly optimizer this was an indicator that we incorrectly determined the sets and uses for an assembly line instruction. Interestingly we encountered multiple scenarios where our MIPS assembly optimizer found a correct location for performing dead assignment elimination even after gcc was supposed to perform this operation while it generated the MIPS assembly. These cases of properly triggering dead assignment elimination via our MIPS assembly optimizer needed to be explicitly ignored as they were incorrectly denoting that we had incorrect sets and uses calculated. For example, one scenario where gcc missed a proper scenario for dead assignment elimination was in the function *save_serial_archive* in the SPEC 2006 FLOAT benchmark 433.milc. In this example there is a scenario where only the first half of the loaded float values are used, and gcc misses checking for this when performing dead assignment elimination.

As we encountered scenarios where the MIPS assembly optimizer was actually incorrectly calculating the sets and uses we would add explicit debugging logs to print out relevant information when an incorrect scenario was triggered so that the MIPS assembly optimizer could be updated to account for these scenarios. Additionally, fall through checks were setup to be triggered when no explicitly handled cases would trigger in a section of processing the assembly. This would denote that an unhandled representation of an assembly line or a new instruction was encountered. In the printout log, relevant details about the current function where the assembly optimizer encountered an unhandled scenario was printed out first followed by data relevant to resolving the unhandled scenario.

CHAPTER 3

RESULTS AND DISCUSSION

3.1 Conclusions

The aims of this project served two main purposes, to provide a tool to be utilized as part of the NSF funded SCALE research project, and to provide myself with a deeper understanding of creating an assembly optimizer. Previously my introduction to working with an assembly optimizer was in the COP66222 Advanced Topics in Compilation course. In this course we were provided with an assembly optimizer for the SPARC assembly language. Using this tool made by Dr. Whalley, we were able to learn how to implement various compiler optimizations without having to modify the code of the compiler itself to perform these optimizations as it was used on the produced assembly code in an additional processing stage abstracted from the original compiler itself. As we were provided with the SPARC version of this assembly optimizer, I used it to learn how to implement the optimizations themselves, but I did not have the experience of building the base tool itself to accurately process generated assembly. This project provided that experience of making the base of a tool like this. By starting with the SPARC version of the assembly optimizer and retargeting it to function for MIPS/SCALE assembly, I also gained the experience of porting a tool to function in another context, specifically one assembly language to another.

To complete this project, I had to first solve the problem of determining what registers are live at each point in the assembly program, and this was done without performing interprocedural analysis by using the *_.objdump_ and *_..s.gkd_ files generated by gcc and producing a *_.inf_ file with information for this analysis. Next, I had to expand all pseudo assembly instructions that would generate two or more machine instructions to have a one-to-one correspondence between assembly and machine instructions to allow for effective instruction scheduling. Additionally, I provided the option to generate SCALE assembly, which is slightly different from conventional MIPS assembly, to be used by the SCALE research group that will continue to use this assembly optimizer for further research. After adding these features extensive testing was done by processing all of the MIPS assembly files generated by the gcc compiler when compiling the MiBench and SPEC 2006 benchmark suites. Processing these benchmarks involved successfully translating each assembly instruction through the assembly optimizer with no errors while also verifying that no instructions with unnecessary dead assignments were introduced by the assembly optimizer.

After completing this, the tool was in a state to properly process various complex scenarios involving gcc generated MIPS assembly. Much like how the SPARC assembly optimizer developed by Dr. Whalley was used for learning how to implement various compiler optimizations, this tool can function in a similar context for implementing compiler optimizations against MIPS assembly. It can additionally be used for assisting students learning MIPS assembly for the first time in computer organization and architecture courses by providing details to the user such as what registers are entering/exiting a block, and what registers are used, set, and dead at each line. This tool's most immediate usage will be in future research conducted by the SCALE research group as the SCALE project will have several assembly language instructions that will support a variety of new architectural features, and the MIPS assembly optimizer will be used on future projects involving the various proposed SCALE ISAs.

3.2 Reflection

By completing this project, I learned the skills of how to accurately go about building a complex tool with many moving parts that will be used by others in future projects. More specifically, I learned how to create a tool to process generated assembly in a meticulous manner that is properly tested against benchmarks to ensure accuracy for when it is used by the SCALE research team. I had many assumptions that were incorrect about how to go about a project like this initially, one of which was to go through MIPS documentation and the extensive lists of all MIPS assembly instructions to manually handle each instruction in the assembly optimizer preemptively. The correct approach was to rather tackle each benchmark immediately rather than to try and predict scenarios to handle the generated assembly and to run the benchmarks later. This was evident when I realized the extent of possibilities for generated MIPS assembly when processing each benchmark in MiBench and SPEC 2006.

I also gained extensive debugging skills to isolate sections of assembly blocks using the gdb debugger and invoking functions to print relevant information about the registers being set and used on each line at runtime when a problem was occurring. Along with those debugging skills acquired, I learned how to properly denote where unhandled scenarios occurred in the assembly optimizer so that they can be caught and handled. As the assembly optimizer was run through each benchmark and would encounter unhandled scenarios, we would capture that unhandled scenario in fall through statements and we would log the relevant information about that unhandled scenario to provide the proper information for handling a specific assembly scenario. As this tool will continue to be used and developed upon for future projects for the SCALE research team, I also gained the experience of working with a larger team in a shared code base to develop something that can be used for further research and learning purposes. The skillset gained through this project has proven to be invaluable and I will continue to use the techniques learned in all my future endeavors.