Implementation of Automated Loop-bound Analysis for Static Timing Analysis Framework

Balaji V. Iyer and Won So
{bviyer, wso}@ncsu.edu
http://www4.ncsu.edu/~wso/csc714

Fall 2005 CSC714 Project
Final Report
30 November 2005

1. Introduction

Many existing timing analyzers require that user specify the number of iterations for each loop in the program [Healy98]. The bounds of every loop must be specifically provided by the user. This can give room too many possible errors. For example, the user might have modified the code and have forgotten to modify the annotations. Errors in these annotations can drastically affect the Worst Case Execution Time (WCET) or the Best Case Execution Time (BCET). Compilers today can implement many optimizations that are not implemented before. For example, compilers can software pipeline loops, which can affect the given annotations.

It would be very beneficial if the compiler analyze the optimized code during compile time and provide the appropriate annotations about loop bounds, induction variables and so forth. Some timing analysis tools have the capability to perform these automations [Byhlin05]. However, the available timing analysis framework does not have the ability to extract this information from the given code.

The main purpose of this work is to incorporate this feature in our timing analysis system. We propose to analyze loops in the code and let the compiler provide the appropriate annotations [Ermedahl97]. These loop bounds are passed to the timing analyzer along with the user's annotations.

In the next section, we explain the goals of this project. Section 3 presents implementation details. Section 4 describes experimental results. Section 5 concludes the report and discusses some future work.

2. Project Goals

These are the major goals we propose to do:

- We propose to find the loop bounds for single-entry, single-exit loops with one induction variable, which only determines the loop exit condition. For the rest of this document, we call this type of loops, "simple loops".

- We incorporate this feature in the current existing framework. The current tool chain does not support automatic loop-bound analysis hence requires manual specification of loop bounds for all loops by .loop file as it is shown in Figure 1. Our tool 'loopbound' will automatically generate .loop file from the assembly file as shown in Figure 2.

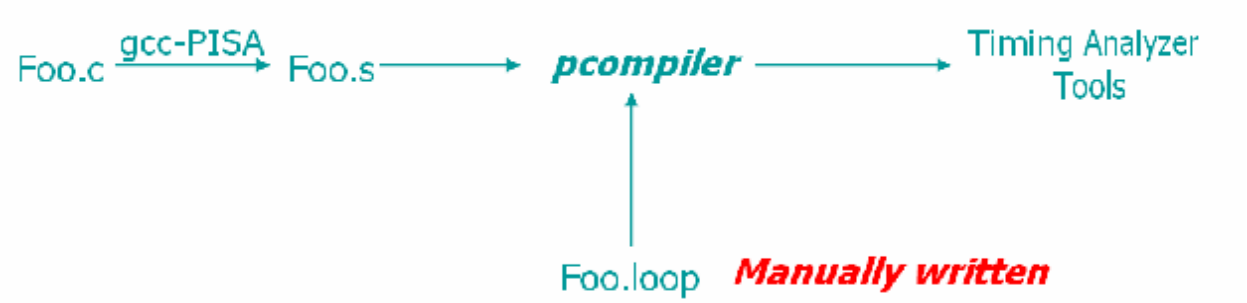

Figure 1 Current tool chain requires manual specification of loop bounds.

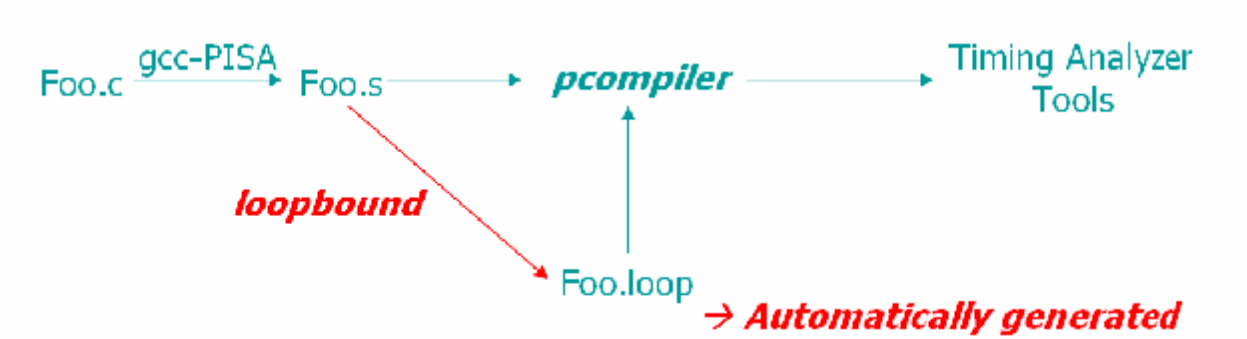

Figure 2 Our tool 'loopbound' will automatically generate the .loop file.

3. Implementation Details

To implement automated loop-bound analysis, we have decided to start from optimization back-end compiler 'opt' which supports CFG construction and loop detection from SPARC assembly code.
Implementation is divided into three parts:

- Front-end: The 'opt' compiler takes SPARC assembly as an input file while the timing analysis tool uses PISA, ISA for Simplescalar [Burger97]. Therefore, the front-end must be modified to process a PISA assembly file.

- Loop-bound analysis: Loop-bound analysis algorithm must be implemented as one of the compiler phases. This phase is added after loop detection phase.

- Back-end: After performing loop-bound analysis, the appropriate .loop file must be generated.

Each subsection below describes each part.

3.1 Front-end

3.1.1 Basic structure

The ‘opt’ program is scalable to add or remove existing analysis and optimization phases. We modified ‘opt’ so that it only enables CFG construction (function: setupcontrolflow) and loop detection (function: findloops) and added one phase ‘loop_bound_analysis’ after ‘findloops’. We used our ‘findloops’ function instead of the one provided because the order of linked list ‘loops’ constructed by the original ‘findloops’ is not easy to handle. Our ‘findloops’ function constructs in-order linked list as it appears in the assembly code.

3.1.2 PISA assembly support

Since existing timing analysis framework uses PISA assembly, we have modified ‘opt’ program to accept PISA assembly and process PISA instructions. Reading an assembly file is done by the function (readfunc () in io.c) and it was modified so that it could process PISA assembly. The structure for defining instruction types (insttypes in io.c) was also redefined for PISA instructions.

The function which constructs CFG (setupcontrolflow in flow.c) is modified because of the differences in branch instruction formats. Besides, other miscellaneous functions for output (E.g. dumpblk in io.c) were also modified to generate valid PISA assembly as an output. The function which sets bit vector for uses and set (setsuses in vars.c) is also dependent on the instruction format and it must be modified if it is ever used. However, we did not modify this function but disable this function because we do not need bit vectors.

3.1.3 Global and local symbol tables

We added symbol table support for both global and local variables in order to use those for parametric analysis. We added the function (readinglobals) in io.c for reading integer global variables from a assembly file and construct a symbol table storing a global variable name and a initial value as a linked list (struct globalinfo* globals in io.c) If the source code is compiled with ‘-g’ option, it also constructs a linked list of locals. (struct localinfo* locals in io.c) The ‘value’ field in ‘localinfo’ stores either stack offset (if it is compiled with –O0) or register number (with higher optimizations).

Despite these efforts, we have decided not to use symbols for parametric analysis. For global variables, it is hard to match a register with a global variable symbol if the source code is compiled with higher optimizations. With higher optimizations, a global value is loaded once into a register and used as a register at the rest of the code. In this case, it is hard to track which global symbol corresponds to a specific register. For local variables, it is also hard to match a register with a local variable symbol with higher optimizations because a register is reused.

3.2 Loop-bound analysis

In this section, we discuss how we find and analyze loops in the Opt Compiler. We start this section by explaining the background work that was done to perform this loop-bound analysis. Section 3.2.2 explains detection of loops. We detail finding the initial and final values in section 3.2.3. Discovery of induction variables are also explained in section 3.2.3. In section 3.2.4, we show how we find the comparison type and the increment value. We explain how we determine nested loops in section 3.2.5. We conclude this section by explaining how we determine if a certain loop is predictable or not.

3.2.1 Preliminary Work and Assumptions.

In order to accurately access the code, we analyzed the gcc-2.6.3 compiler and the PISA (or simplescalar) machine description that generates code for the PISA architecture. The only compare instruction (for fixed point) we encountered in the simplescalar toolset was SLT (Set Less Than), SLTI (SLT Immediate), SLTU (SLT Unsigned) and SLTIU (SLTI Unsigned) [Burger97]. Other than this, we also found many predicated branches such as: beq, bne, blez, bgez, bgtz and bltz. Our first task was to see how these instructions were emitted in the GCC machine description. In this section we try to explain briefly how branches are emitted by the GCC compiler.

```
(define_expand "cmpsi"
  [(set (cc0)
        (compare:CC (match_operand:SI 0 "register_operand" "")
                    (match_operand:SI 1 "arith_operand" "")))]
  ""
  "
{
  if (operands[0])              /* avoid unused code message */
    {
      branch_cmp[0] = operands[0];
      branch_cmp[1] = operands[1];
      branch_type = CMP_SI;
      DONE;
    }
}")
```

Figure 3: Machine Description to Handle "CMPSI" instruction.

Figure 3 explains the output of compare (single integer) instruction. When a compare instruction is to be output by the compiler, the two variables that are being compared to are saved in an array (branch_cmp). The type of comparison necessary is also stored in the variable branch_type.

```
switch (branch_type)
  {
  default:
    goto fail;

  case CMP_SI:
    reg = gen_int_relational (test_code, (rtx)0, cmp0, cmp1, &invert);
    if (reg != (rtx)0)
      {
        cmp0 = reg;
        cmp1 = const0_rtx;
        test_code = NE;
      }

    /* Make sure not non-zero constant if ==/!= */
    else if (GET_CODE (cmp1) == CONST_INT && INTVAL (cmp1) != 0)
```

Figure 4: Emitting the Relational operation

In the file ss.c, there exists a function called "gen_conditional_branch" which reads these values and then creates a relational RTL (SLT). Figure 4 displays the code (shaded part) for this step. This instruction emits a SLT and an appropriate branch instruction. After these two instructions are emitted, the compiler emits a jump instruction that is used to jump into the loop block (this is the code that is executed inside the loop.

```
    }

  emit_jump_insn (gen_rtx (SET, VOIDmode,
                           pc_rtx,
                           gen_rtx (IF_THEN_ELSE, VOIDmode,
                                    gen_rtx (test_code, mode, cmp0, cmp1),
                                    label1,
                                    label2)));

  return;
```

Figure 5: Function responsible for outputting the Jump Instruction.

Figure 5 explains the function used (inside the "gen_conditional_branch") function that is used to emit the jump instruction. In Figure 6(a) and (b), we show a simple for loop along with its translated assembly code respectively.

(a)

```
int main(void)
{
  int ii = 0;
  int jj = 5;

  for (ii = 0; ii < 10; ii++)
    jj=5;

  return 0;
}
```

(b)

```
$L2:
        lw      $2,16($fp)
        slt     $3,$2,10        # <== COMPARE INSTRUCTION
        bne     $3,$0,$L5       # <== BRANCH OUT WHEN THE CONDITION IS NOT MET
        j       $L3             # <== "TAKE" THE LOOP
```

Figure 6: (a) Example C Code and (b) Translated PISA Assembly code (just part of the for-loop) along with our added annotations for clarity.

Using this understanding of the GCC compiler, we began to create a plan of action to complete this project. In order to make this project feasible for the time allotted, we have made the following assumptions that we feel are realistic:

1) This implementation of loop bound analysis assumes that the code is generated using gcc-2.6.3 with the machine description provided in the simplescalar website.
2) We currently handle only single entry and single exit loops. The only exception to this is a loop that contains function calls. However, our algorithm can be extended to multiple exit loops with minimal modification.
3) For loop increments, we only support the following methods: add, subtract, multiply, divide, and shifts.
4) We also assume that multiple increments of loop induction variable, with different arithmetic are not present.
5) We also assume that the for-loop does not contain an infinite loop scenario.

3.2.2 Detecting a Loop

The loops are detected using the findloops method in the Opt Compiler. The loops are then saved in a singly-linked-list structure called loopnode. The head of the loop is stored in a variable called "loops." Each element in the linked list holds one loop. The pointers to all the basic-blocks in the loop are stored under another singly linked list called "blocks."

The singly-linked-list loopnode holds all of the results from our analysis. For the rest of the document, when we mention a member name, we imply that the member belong to this data structure. In further sections we describe how we populate our results in the linked list.

3.2.3 Finding the Initial, Final bound, and induction variable values of a Loop

In order to detect these values, we have to first find the root node of the loop. Figure 6(b) is an example of the root node. For clarity, we have shown the control flow graph of the code depicted in figure 6(a) and marked the root node in Figure 7.

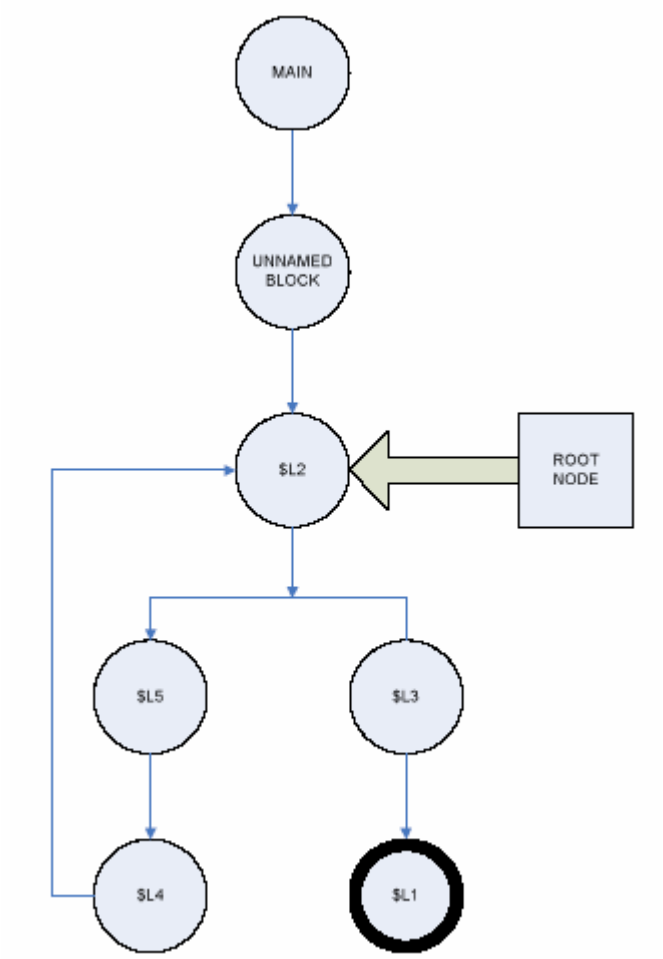

Figure 7: Control flow graph of the Example code given in Figure 6(a).

It is easy to see that the loop nodes for this program are $L2, $L5 and $L4. To detect the root node in the graph, we go through every block in the loop and find the block whose successors are not part of the loop. From figure 7, it is very obvious that the only block that falls into this category is $L2. In this basic block, we search for the compare instruction.

In our analysis of both GCC and our test cases, we find that there are mainly two cases that are possible. First, the compiler emits an SLT instruction and a “BEQ” or “BNE” instruction. Second, it doesn’t emit an SLT instruction, but emits a BEQZ, BNEZ, BLTZ, BGEZ, BGTZ or BLEZ instructions.

In the first case, the first source register hold the induction variable (or the induction variable value) and the second source variable holds the value of the final bound. This is stored under the member name, “final_val.” If this is value is a constant value then we store the “final_val_type” as LOOP_CONSTANT. If this value is a register (or stack location) then we call this variable LOOP_REGISTER.

When we catch the induction variable, we search to see if this is stored as a stack value (spill code) or a dedicated register. If the induction variable is saved in a stack location, the compiler will load this value into a temporary register (please see first instruction in figure 6(b)). We traverse the rest of the instructions (backwards) in the basic block and try to see if any loads are done into the induction variable we caught. If we find one, then we know that the induction variable is in the stack and we save this location.

If the induction variable is a register value, we examine instructions inside the predecessors of the root-node (backwards). If we encounter a load immediate instruction whose destination is the induction variable, then we know the initial value. We save this value under the member name "init_val" and the type under "init_val_type" as LOOP_CONSTANT. On our quest to the load immediate instruction, if we find any arithmetic done to the induction variable, we store the register the induction variable is dependent upon (as init_val_reg) and store the type as LOOP_REGISTER.

If the induction variable is a stack location, we do the same analysis as above, but instead of search for a load immediate, we search for the first store (SW) instruction. If the register that the store instruction is storing is $0, then we know the initial value is 0, and this is stored in the appropriate location with the appropriate type mentioned above.

If the store register is not $0, then we mark that register value and traverse up till we find a load word into that register. We examine all the instructions between this load and store and search all the instruction that load an immediate value into this marked register. This is our initial value. We save this immediate value into init_val and set the init_val_type as LOOP_CONSTANT.

Another possibility is that the initial-value is a global variable. If this is so, then the global value is stored into a register using a Load instruction. We search for this. If we find such a pattern, then we store the name of the global value into the init_val_reg location and mark the type as LOOP_REGISTER.

3.2.4 Finding Comparison Types and Increment values.

In the C code, there are six possible values for comparison: greater than, greater than or equal to, less than, less than or equal to, equal to and not equal to. Since the only comparison value the architecture provides is less than, it was interesting for us how they handled these six cases.

For less than or equal to, they incremented the comparison value by one and branched into the loop blocks if it was less than. The SLT instruction sets its destination register to one if the first source register is less than the 2^{nd} register (or immediate value). The BNE instruction will examine the destination register of SLT instruction and branch to the appropriate location if the register is not zero.

Similarly, greater than or equal to is done by using a SLT instruction, but branching into the loop blocks when the destination register of SLT instruction is zero (using BEQ). For greater than

instruction, they are handled the same way as less than or equal to, but as with the greater-than case, the branching is done when the destination register if SLT is equal to zero.

For the equal-to and not-equal case, the final value is subtracted from the induction variable, and the branching is done if the result is zero or non-zero, respectively.

Another case of for-loops occur when we have the final value equal to zero. For this, the compiler does not insert any SLT instructions. They use the following instructions to compare with the induction variable:

1. BEQZ = Branch if Equal to Zero
2. BLTZ = Branch if Less than Zero
3. BLEZ = Branch if Less than or Equal to Zero
4. BGTZ = Branch if Greater than Zero
5. BGEZ = Branch if Greater than or Equal to Zero.
6. BNEZ (or BNE) = Branch if Not Equal to Zero

The appropriate branch condition is stored in the variable called "compare_type."

The next task was to find the increment value. For the case where the induction variable is a dedicated register, we search for a self destructive instruction with the induction variable as the destination register and we store the immediate value as "inc_value." The type of operations (or instruction name) is stored in inc_type.

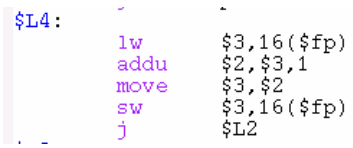

Figure 8: Incrementing the induction variable that is located in a stack

For the case when a stack location is used, we look for the load and store pairs of the address and we look at instructions between the load and store that writes to the register that is stored in the memory location of the induction variable.

Figure 8, provides the L4 basic block from the assembly code of Figure 6(a). Recall that the induction variable is stored in location 16($fp). Please notice that $3 stores its value into the induction variable. We traverse the instructions backwards till we find a load instruction and we can easily see that $3 is taking values from $2 and $2 holds the value from the add instruction above. Right above this instruction, $2 is stored with the sum of $3 and 1. $3 holds the previous value of the induction variable (denoted by the load word instruction). In this example, we store "addu" into "inc_type", and 1 into "inc_value."

3.2.5 Finding Nested Loops and Non-rectangular loops

Figure 9 shows the function we used to check if two blocks are nested or not. Loop A is not nested inside Loop B if there exists a node in A that is not contained in B. We use this to see if two loops are nested. For rest of this section, we use Loop A to explain the outer loop and Loop B to explain the inner loop.

```
int is_nested(struct loopnode *node, struct loopnode *node2)
{
  struct blist *temp_blk = NULL;

  for (temp_blk = node->blocks; temp_blk != NULL;
       temp_blk = temp_blk->next)
  {
    if (!inblist(node2->blocks, temp_blk->ptr))
    {
      return FALSE;
    }
  }
  return TRUE;
}
```

Figure 9: Function to see if two loops are nested or not.

If Loop B is dependent on the induction variable of Loop A, then we call it a non-rectangular loop pair. Figure 10 gives a simple example of a rectangular and non-rectangular loop.

```
int main(void)
{
  int ii = 0;
  int jj = 0;

  for (ii = 0; ii < 10; ii++)
  {
    for (jj = 0; jj < 10; jj++)
    {
      /* Rectangular Loop Pair */
    }
  }

  for (ii = 0; ii < 10; ii++)
  {
    for (jj = 0; jj < ii; jj++)
    {
      /* Non-rectangular Loop Pair */
    }
  }
}
```

Figure 10: Examples of Rectangular and Non Rectangular Loops

We determine these loop pairs and mark them appropriately. The type of loop is stored using a bool value called “rectangular” that is set to TRUE or FALSE as appropriate. These loop pairs have a different notation when generating the loop macros compared the rectangular loops.

3.2.4 Predicable Loops

We determine if a loop is predictable or not, if the final bound of the loop is changed inside the loop. This is not an issue if final value is of type LOOP_CONSTANT. This is determined by walking through all the instructions inside a loop and seeing if the register value of the final

value is written to by any of the instructions. If so, we set the member name “predictable” to FALSE in the loopnode structure.

3.3 Back-end

3.3.1 Command-line Interface

The command-line interface is modified to take an assembly file as the first argument. The same name with .loop extension is automatically created and written as an appropriate format. For example, the command line “loopbound foo.s” will automatically generate ‘foo.loop’ given ‘foo.s’ assembly file.

3.3.2 Debug output

Debug information is printed into stdout. It prints out loop information obtained by the loop-bound analysis followed by loop number and nested levels. Table 1 summarizes it.

Field name	Meanings
predictable	1 = predictable / 0 = non-predictable
rectangular	1 = rectangular / 0 = non-rectangular (either inner or outer loop)
Init	integer value for constant / $x for register / y($fp) for stack offset
Final	integer value for constant / $x for register / y($fp) for stack offset
Bound	Actual loop iteration both init and final are constant
Comp_type	Compare operator: EQ, NE, GT, LT, GE, LE
inc_type	Incremental operator: actual instruction
inc_val	Incremental value
inc_reg	Induction register: $x for register / y($fp) for stack offset

Table 1: Information for each loop shown in debug output

3.3.3 .loop file generation

The .loop file is the format used for ‘pcompiler’ to specify loop bounds. Mainly there are 3 different types of loops: 1) A loop with fixed number of iterations (5 in this case) is represented with number of iterations as shown on line 4 in Figure 11(b). 2) A loop with variable number of iterations is represented with a parameter (.iter2 in this case) as shown on line 6 in Figure 11(b). 3) An outer loop and an inner loop which form non-rectangular loops are represented with a special format as shown on line 8 and 10 in Figure 11(b).

```
(a) C source code
Main() {
  int i,j,x=5;
  for (i=0; i<5; i++) { /* loop 1: fixed */ }
  for (i=0; i<x; i++) { /* loop 2: variable */ }
  for (i=0; i<15; i++) /* loop 3: non-rectangular outer */
    for (j=0; j<i; j++) { /* loop 4: non-rectangular inner */ }
```

```
(b) .loop file
 1:     -3
 2:     main
 3:     ! loop 1
 4:     5 5 -1 -1
 5:     ! loop 2
 6:     -4 r[11] 0 r[1] 1 s -2 .iter2 .iter2 -1 -1
 7:     ! loop 3
 8:     -4 r[12] 0 15 1 s 15 15 -1 -1
 9:     ! loop 4
10:     -3 0 r[13]+0 15 4 1 6 c0 c15 1 -1 -1
```

Figure 11: Example source file (a) and corresponding .loop file (b)

Because of the reasons mentioned in 3.1.3, we use a variable name '.iter<loop_no>' for parametric analysis though we initially planned to use real symbol names. For register numbers for loops, we use increasing numbers stating from r[11] for each loop. For parametric variables, we use increasing numbers starting from r[1] depending on loop nest levels.

It could generate "Can not analyze: …" message instead of valid .loop file line for the cases where current .loop file generation routine (dumploops_dotloop in io.c) can not handle. For example, the message "Can not analyze: loop init, final both are variables" is generated when both init/final value types are variables. However, these limits are not the ones from the loop-bound analysis algorithm but from the .loop file format. Therefore, we can carefully say that integration of pcompiler and loopbound can provide more information applicable for the timing analyzer tools.

4. Experimental Results

In order to test our results, we created about 72 different for-loops. Of these 72 loops, 25 of them were simple loops with both initial and final values are constants. 19 out of 72 were distinct loops that tested all the possibilities applicable for for-loops. 10 out of the 19 distinct ones were rectangular and 9 out of 19 were non-rectangular. 5 out of 72 had final values as global values. 5 out of 72 had initial values as global values. Nine out of 72 had multiple nested loops where some were non-rectangular. Similarly, nine out of 72 had multiple nested loops with all of them rectangular. These programs were compiled using –O0, -O1 and –O2 optimizations. The loop files were generated along with debug outputs and they were checked manually for accuracy.

In order to demonstrate that our program is working as proposed, we have written two files: test_mixed.c and test_all_fors.c. These files are representative of the 72 loops. We have also included a Makefile that will compile the files using a PISA compiler, and then run the assembly

files through our code when typed “make all test” at the command line. This will output the debug output along with the loop files.

Our loopbound function is tested using GCC 2.8.1, 2.95.2, 3.2.3, 3.4.2 and 4.0.0. The outputs seem to match as wanted.

5. Conclusions and Future Work

We created a loop-bound analyzer that analyzes loops in an assembly code for the PISA architecture and generate the appropriate .loop files for the loops that are analyzable. This will greatly help many computer scientist and engineers to automate their timing analysis research in this architecture, and can greatly help reduce human errors.

For future work, the loop-bound analysis algorithm can be extended to more complex loops such as loops with multiple induction variables, loops with multiple exists [Healy98] and conditionally executed loops. As far as the tool chain is concerned, the existing tool ‘pcompiler’ and our new tool ‘loopbound’ can be successfully integrated and provide more useful information to timing analyzer tools.

6. Individual Contributions

Balaji V. Iyer was manly responsible for implementing the loop-bound analysis. (loop_bound_anlaysis.c) Won So was mainly responsible for front-end and back-end implementation. We both contributed equally in testing and debugging. We believe this was a fair division of labor.

7. References

[Byhlin05] D. Byhlin, A. Ermedahl, J. Gustafsson, B. Lisper. "Applying Static WCET Analysis to Automotive Communication Software,” ECRTS’05.

[Healy98] C. Healy, M. Sjodin, V. Rustagi, D. Walley, “Bound loop iterations for Timing Analysis,” RTAS’98

[Ermedahl97] A. Ermedahl and J. Gustafsson “Deriving Annotations for Tight calculation of Execution-Time”, Proceedings of the European Conference on Parallel Processing, 1994

[Burger97] D. Burger and T. Austin, “Simplescalar Toolset, Version 2.0,” www.simplescalar.com, 1997

8. Disclaimer

This paper and associated software changes are intended for **informational purposes only**. Therefore, any use of the information presented in this student work is at your own risk. Balaji V. Iyer and Won So provide **no warranties of any kind** surrounding the use of this material.