Loop Optimizations for MLton

by

Matthew John Surawski

A Project Report Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in
Computer Science

Supervised by

Dr. Matthew Fluet

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

May 2016

Abstract

Loop Optimizations for MLton

Matthew John Surawski

Supervising Professor: Dr. Matthew Fluet

This report discusses the design and implementation of loop optimizations for the MLton Standard ML compiler. The loops that programmers typically write do not take advantage of the full capabilities of modern processors. Recognizing and transforming these loops into alternate forms that better match the capabilities of the hardware can result in significant performance gains without a significant increase in code size.

The specific optimizations examined are *loop unswitching* and *loop unrolling*. Unswitching is a transformation that moves conditional branches on loop-invariant conditions outside the loop. Unrolling transforms a loop of known iteration count into a larger loop with a reduced iteration count, or eliminates the loop entirely and replaces it with straight-line code. These transformations are implemented as optimization passes on MLton's Static Single Assignment intermediate form. In order to measure the benefits of these optimizations, MLton's benchmark suite is run against the modified compiler. The resulting execution times and binary sizes show that these optimizations can introduce significant performance improvements without unacceptably increasing the size of the program.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Programmers often focus on writing their programs such that they that are easily understood by other programmers. A consequence of this is that these programs are not necessarily in the optimal form for computers to run. As the bridge between the programmer and the machine, compilers have the responsibility of optimizing the source program so that programmers are not penalized for writing less than optimal code. The goal of an optimizing compiler is to reduce the size or improve the running time of the compiled program relative to what would be produced by a straightforward transformation from source to binary.

Loops are fundamental control-flow structures and popular transformation targets for optimizing compilers. Loops contain code that is intended to run more than once and it is often the case that programs spend most of their execution time inside a loop [4]. A compiler that can analyze these loops can perform a variety of transformations on them in order to improve their performance on the target architecture.

Instruction pipelining, SIMD instructions, branch prediction, and other modern processor features can provide significant improvements in execution time for programs that can take advantage of them. Loops in the state that programmers typically write them limit the processor's ability to utilize these features. Consider the loop in the following C program:

```
int myarray[400];
// fill myarray with some values
for (int i = 0; i < 400; i++) {
        myarray[i] = myarray[i] * 2
}
```

This loop performs an arithmetic operation on a sequential set of data. This is a candidate for a SIMD instruction that would perform the multiplication on multiple values at once, but it is not obvious to the compiler that it can use a SIMD instruction. While looking at the loop's body, the compiler can only see one array access and can't guarantee that it should perform the multiplication on the next few elements in the array. If this loop were transformed into the following loop:

```
for (int i = 0; i < 400; i+=4) {
    myarray[i] = myarray[i] * 2
    myarray[i+1] = myarray[i+1] * 2
    myarray[i+2] = myarray[i+2] * 2
    myarray[i+3] = myarray[i+3] * 2
}
```

it is now clear to the compiler that a SIMD instruction can be used. Additionally, the number of times the loop will iterate has been reduced resulting in fewer inequality comparisons made at runtime during the execution of the loop.

This paper looks at how two common loop optimizations, unrolling and unswitching, can be implemented as optimization passes in the MLton compiler. Loop unrolling is an optimization that duplicates the loop's body a number of times resulting in either a completely unrolled loop, or a loop with an expanded body. A completely unrolled loop is a series of copies of the original loop's body where the number of copies is equal to the number of times the loop would have normally ran. An expanded loop has a body n times the size of the original loop body and an iteration count that is reduced by a factor of n. This optimization aims to improve performance by reducing/eliminating loop overhead, providing opportunities for parallelism and SIMD instructions, and improving cache locality[1]. Loop unswitching acts on loops that contain a conditional branch that is invariant to the loop itself. The branch is extracted outside the loop and the rest of the loop's body is replicated on each branch. This avoids branching during every iteration of the loop, eliminating the conditional branch instruction and shrinking the body of the copied loops.

We will be discussing these optimizations in the context of the MLton SML compiler. MLton is a whole-program, optimizing compiler that produces small, high performance, binaries for programs written in the Standard ML programming language. As a whole-program compiler, MLton compiles the entire program at once and makes the maximal amount of information available to the optimization passes. Standard ML (SML) is a functional, statically typed programming language with many powerful features including higher-order functions, polymorphism, and functors.

During compilation, MLton transforms the source program through several different intermediate representations (IRs) that increasingly resemble the outputted assembly rather than the source program. The first of these representations is called the *AST* (Abstract Syntax Tree) and is produced by the front end of the compiler. The last is the *Machine* representation, an untyped register transfer language that acts as the input for the various codegens. The IR we will be focusing on is the *SSA* (Static Single Assignment) representation in which every variable is assigned exactly once and every variable is assigned before it is used. The properties inherent to a program in SSA form are useful for performing optimizations [2].

MLton's SSA IR consists of sets of datatype declarations, global statements, and functions. Functions consist of a collection of basic blocks. Basic blocks consist of a list of statements followed by some control flow transfer. MLton's SSA form implements the SSA ϕ function by allowing basic blocks to take variables as arguments. Instead of ϕ choosing which value to assign to a variable at the destination block, the blocks transferring to said block pass the ϕ variables as arguments. This SSA form is the primary IR transformed by MLton's existing optimization passes.

As a functional programming language, SML programs typically do not feature imperative-style loops. When a loop is needed, SML programmers will typically write the recursive function equivalent of the imperative loop. As an optimizing compiler, MLton performs function inlining and tail-call optimization in order to transform the recursive code into the iterative form, which is typically more performant on modern processors. The majority of

the loops we will be attempting to optimizing will be produced by these optimizations as opposed to being directly written in the source program.

The remainder of this paper discusses the process of implementing loop unswitching and unrolling in MLton. Chapter 2 discusses the benefits and implementations of these transformations in other compilers. Chapter 3 discusses the MLton-specific implementation of the transformations. Chapter 4 contains performance benchmark results of programs transformed by these optimizations. Chapters 5 concludes the paper and discusses future work.

Chapter 2

Related Work

2.1 Implementation Specificity of Optimizations

Before we take a look at how unrolling and unswitching have been implemented in other compilers it is worth noting that the design of a specific optimization is heavily influenced by the design of the compiler itself. There is no set of intermediate forms that all compilers use when transforming their programs from source to target. Even the commonly used forms like SSA have implementation-specific details; for SSA the exact implementation of the ϕ function is a source of ambiguity for compiler writers. MLton allows basic blocks to take arguments in order to provide the functionality of the ϕ function, but other compilers have made different design decisions. Optimizations may also be performed on any of the intermediate forms of the program. The choice of form heavily influences the amount of information available to the compiler; forms closer to the source program have more information about what the programmer actually wrote, while forms closer to the machine code have more information about what operations will be executed to perform the computations. The transformations discussed in this report are the same transformations made by other compilers with the same goals in mind, but the design of the transformations is heavily influenced by the design of MLton's SSA form.

2.2 Unswitching

Bacon [1] describes loop unswitching as a transformation that can be applied to loops that contain conditionals with loop-invariant test conditions. The loop is replicated for each

branch of the conditional, removing any overhead of performing the branch from inside the loops. The new loops are smaller, simpler, and more open to being acted upon by other optimizations.

Deciding if a branch condition is invariant within the loop is not a trivial problem. The conditions themselves can be detected through the analysis performed by loop invariant code motion [1], but actually moving the computation of the condition outside the loop requires care. If evaluating the condition would throw an exception, simply moving it outside the loop could change the meaning of the program.

2.3 Unrolling

Loop unrolling is a transformation that duplicates the loop's body a number of times; reducing the number of iterations the loop must run, or eliminating the loop entirely. The number of times the body is replicated is known as the *unrolling factor* u. Each iteration of an unrolled loop will iterate by u steps instead of one [1]. If the loop's iteration count is not an integral multiple of u, extra iterations are added outside the unrolled loop to compensate.

This optimization improves performance in three key ways. The first is that it reduces the overhead of the loop [1, 3]. A loop that was unrolled by a factor of 5 will only perform $\frac{1}{5}$ of the iterations which reduces the number of branch tests by the same factor of 5. The second is the opportunities created for instruction level parallelism [1, 3]. A larger loop body allows for the processor to take greater advantage of instruction pipelining in order to do more work at once. The third is the opportunities provided for additional optimizations on unrolled loops [3]. A completely unrolled loop has none of the original loop's control flow, making opportunities for optimization more obvious.

An overly aggressive approach to loop unrolling has been shown to have a negative effect on the performance of programs [3]. When the body of a loop is expanded too greatly it may exceed the size of the instruction cache. This causes the program to experience a large number of cache misses that it would not have otherwise encountered with the original loop.

Chapter 3

Implementation

3.1 Background

3.1.1 Adding an Optimization Pass to MLton

Adding an optimization pass that operates on MLton's SSA form is a straightforward process. The optimization pass itself is a functor that produces a structure with a single function that takes a program as an argument and returns the optimized program as a result. Figure 3.1 contains an example of an empty optimization pass. A program is made up of a

```
functor LoopUnswitch (S: SSA_TRANSFORM_STRUCTS): SSA_TRANSFORM =
struct

open S

fun transform (Program.T {datatypes, globals, functions, main}) =
   let
      (* Transform the program *)
   in
      Program.T {datatypes = datatypes,
                 globals = globals,
                 functions = functions,
                 main = main}
   end

end
```

Figure 3.1: An empty optimization pass

collection of datatypes, a collection of global statements, a collection of functions, and the label of the main function. For the purposes of this paper, we only focus on modifying the functions themselves.

Once the optimization pass is written it must be decided where in the optimization pass pipeline it will be inserted. A pass may be run more than once at various points in the pipeline and be more effective at certain points than at others. Figure 3.2 contains a snippet from the SSA optimization pipeline. Note how a pass like *RemoveUnused* is run at different

```
{name = "removeUnused1", doit = RemoveUnused.transform} ::
{name = "introduceLoops1",
 doit = IntroduceLoops.transform} ::
{name = "loopInvariant1", doit = LoopInvariant.transform} ::
{name = "inlineLeaf1", doit = fn p =>
 Inline.inlineLeaf (p, !Control.inlineLeafA)} ::
{name = "inlineLeaf2", doit = fn p =>
 Inline.inlineLeaf (p, !Control.inlineLeafB)} ::
{name = "contify1", doit = Contify.transform} ::
{name = "localFlatten1", doit = LocalFlatten.transform} ::
{name = "constantPropagation",
 doit = ConstantPropagation.transform} ::
{name = "useless", doit = Useless.transform} ::
{name = "removeUnused2", doit = RemoveUnused.transform} ::
```

Figure 3.2: A sample from the optimization pipeline

points in the pipeline. Variables may become unused as optimizations are applied to the program, so it makes sense to clean them up now and then. There are two general criteria for deciding where to insert a new pass in the pipeline:

- An optimization should be run at the earliest point where it will have a significant effect on the program
- The optimization should be run again if it is likely that more optimization opportunities have been uncovered

The first point enables other optimizations to take advantage of opportunities created by the pass that is being introduced. Not all optimization passes have a directly measurable impact on performance; these transformations transform the program into a state that enables other optimization passes to act upon it. This brings us to the second point: an optimization pass should be rerun if more opportunities may have been introduced. While some optimization passes, i.e. *ConstantPropagation*, only provide a significant benefit once, others may find opportunities that they did not discover in the previous state of the program.

For the loop optimization passes introduced in this paper, there is one key optimization pass that they should be run after. This is the *IntroduceLoops* pass, which performs tail-call optimization on recursive functions and transforms them into loops in the control flow graph. In a functional language like SML it is rare to see imperative style loops written in the source program. SML only has syntax for **while** loops and they are unpopular due to the requirement of a mutable variable. Most SML programmers write their loops in tail-call recursive form in order to cleanly express their algorithm while knowing that tail-call optimization will eliminate the overhead of recursive function calls. MLton's SSA form is an arbitrary control flow graph and while loops may appear from other sources, the transformations described in this paper do not distinguish between them and operate on any loop meeting the criteria for optimization.

3.1.2 MLton's SSA form

MLton's SSA form represents the program as the composite of a collection of datatypes, a collection of global statements, a collection of functions, and the label of the main function. Functions are composed of a function name, a vector of arguments, a vector of body blocks, the label of the starting block, and vectors for the return and exception types of the function. For the purposes of this paper we are only interested in the collection of global statements and the blocks that make up the body of each function. The transformations described are intraprocedural and only operate on the CFG of each function's body. The global statements contain the definitions of constant variables, which are important for loop

unrolling.

Blocks are composed of a label, an argument vector, a statement vector, and a transfer to the next block(s). The block's arguments serve as the SSA ϕ function values for that block and are passed in implicitly or explicitly from the transferring block. The statement vector contains a list of statements that belong to the block. A statement is an expression that is optionally bound to a variable. There are 8 different transfers that a block may take:

- Arith - performs an arithmetic operation with overflow checking and implicitly passes the result to the next block, or transfers to an overflow handler
- Bug - a transfer that should never be reached
- Call - performs a function call and transfers to the next block upon returning
- Case - performs pattern matching on a value. Matched values are passed as arguments to the next block. **if ... then ... else ...** expressions are generalized to this transfer
- Goto - unconditional transfer to a block that explicitly passes arguments
- Raise - raises an exception
- Return - returns from a function call
- Runtime - calls a runtime function and transfers to the next block upon returning

For loop unswitching, we are concerned with only the *Case* transfer because it performs a conditional branch on a value. For loop unrolling, we examine *Arith, Case,* and *Goto* transfers in order to compute the iteration counts of loops.

A *Case* transfer t consists of either a list of datatype constructor and label pairs $c_1, c_2, \ldots, c_n$, or a list of word literals and label pairs $w_1, w_2, \ldots, w_n$. It is important to note that datatype cases deconstruct a datatype and pass the values as arguments to the target block. For our purposes we don't care which type of *Case* transfer we are dealing with, so we say a *Case* transfer t has a set of pattern/label pairs $p_1, p_2, \ldots, p_n$.

3.1.3 Detecting Loops

In order to perform optimizations on loops, we first must discover the loops in the control flow graph. In order to do so we take advantage of MLton's existing implementation of Steensgaard's Loop Forest algorithm [5]. The details of the algorithm and how it is implemented are outside the scope of this paper.

Steensgaard's algorithm produces a *loop nesting forest* for the given CFG. A loop forest consists of a list of *loops* and a list of blocks not in any loop. A loop consists of a list of *headers*, and a child loop forest. A header is a block that acts as the entry point into a loop. A header will have one or more transfers from outside the loop's body to itself and it will also have one or more transfers from within the loop's body to itself. If a loop forest contains no nested loops, then we know it is an *innermost loop*. An important property of Steensgaard's algorithm is that every block in the CFG must appear in a loop forests list of blocks that are 'not in a loop' exactly once. For loop forests with no nested loops this list of blocks is equivalent to the list of blocks that make up the loop's body, including the header.

In this paper we only consider performing optimizations on innermost loops with exactly one header. Avoiding tremendous code growth is a priority and of all the loops in a nested forest, the innermost loop is most likely to be the one small enough to transform. In the case of loop unrolling where a loop may be eliminated entirely, additional runs of the unrolling pass will take care of any loops that became innermost loops after the old innermost was eliminated. For loops with multiple headers, it is unclear how to perform the optimizations in a meaningful way.

3.1.4 SSA Restore

The loop optimizations described in this paper transform the program by copying the loop's body a number of times. A consequence of copying blocks is that the variables defined in those blocks become defined twice, violating the SSA principal. In order to solve this problem we make use of MLton's *Restore* function. The Restore function traverses the

CFG and assigns new variable names to variables that have multiple assignments. Every function that we have optimized is run through Restore in order to correct any SSA errors.

3.1.5 Computing Code Size

It is important to have knowledge of the size of the output program in order to decide if some transformations should be made or not. Loop unswitching and unrolling can both result in explosive code growth if they do not heed the size of the loops they transform. An existing optimization pass, *Function Inlining*, provides functionality for estimating the size of a program. That functionality will be used to decide if and how our loop optimizations will be performed.

The difficulty of estimating code size varies depending on how close the intermediate representation is to actual machine code. MLton's SSA representation represents the program after many of the high-level features of the source language have been eliminated, so it is possible to get a usable estimate. The specifics of how function inlining estimates size are omitted, but given a list of blocks it produces a unit-less integer value that represents the approximate size of the blocks.

3.2 Loop Unswitching

Loop unswitching is an optimization that aims to remove conditional branches from loops when the condition of the branch is invariant to the loop itself. In order to do so, the loop's body is replicated down each branch of the conditional. The primary goal of this optimization is to simplify the loop being unswitched, opening up the new loops for further optimization.

3.2.1 Detecting Unswitchable Loops

In order to detect an unswitchable loop, we must find a block within the loop's body that performs a conditional branch on an invariant value. In MLton's SSA, this is a block that performs a *Case* transfer on a test variable that is not defined in one of the loop's body

blocks. Note that, while we do not consider these variables, it is possible that a variable defined within the body of a loop is invariant to the loop itself. Variables that are computed to have the same value for every iteration of the loop can be considered invariant. Moving these variables outside the loop is part of the optimization *Loop Invariant Code Motion* and therefore outside the scope of loop unswitching. Instead, our *LoopUnswitch* pass will leverage other passes to lift invariant variables outside the loop.

We can define the detection of unswitchable loops as follows. Let $L < h, B >$ be an innermost loop with header block h and body blocks B. Note that $h \in B$. For a block $b \in B$, let the set of variables defined by statements be v_s, the set of variables defined by arguments be v_a, and the set of variables defined in that block $V_b = v_a \cup v_s$. The set of all variables defined within the loop is $V_L = V_{b1} \cup V_{b2} \cup \ldots \cup V_{bn}$ for $b_i \in B$. A block with a transfer suitable for unswitching is any block b with a *Case* transfer t on a variable v_t, where $v_t \notin V_L$. If a loop has a block that has a suitable transfer, we can perform the unswitching transformation on the transfer of that block. For loops that have multiple suitable transfers, we choose an arbitrary one to transform upon.

3.2.2 The Unswitching Transformation

For a loop $L < h, B >$ with an unswitchable transfer t that has pattern/label pairs $p_1, \ldots, p_n$ we can perform the unswitching transformation as follows. First, create a new block b_e where b_e's arguments and label are the same as h's. b_e has no statements and has the transfer t. Note that the block labels in p_n are undefined at this point as they will be replaced with blocks that transfer to each of the copied loops in the future. b_e serves as the new entry to the unswitched loop. It is not a loop header itself because it is not within the loop, but it retains the same label as the old header so that all existing transfers to the old loop will now transfer to it.

The next step is to make a copy of the loop for each of the branches $p_i; i \leq n$ in t. The procedure to copy a loop is shown in Figure 3.3.

```
(* Copy an entire loop. *)
fun copyLoop(blocks, blockInfo, setBlockInfo) =
  let
    val labels = Vector.map (blocks, Block.label)
    (* Assign a new label for each block *)
    val newBlocks = Vector.map (blocks, fn b =>
        let
          val oldName = Block.label b
          val oldArgs = Block.args b
          val newName = Label.newNoname()
          val () = setBlockInfo(oldName, (newName, oldArgs))
        in
          Block.T {args = Block.args b,
                   label = newName,
                   statements = Block.statements b,
                   transfer = Block.transfer b}
        end)
    (* Rewrite the transfers of each block *)
    val fixedBlocks = Vector.map (newBlocks,
      fn Block.T {args, label, statements, transfer} =>
        let
          val f = fn l => fixLabel(blockInfo, l, labels)
          val newTransfer = Transfer.replaceLabel(transfer, f)
        in
          Block.T {args = args,
                   label = label,
                   statements = statements,
                   transfer = newTransfer}
      end)
  in
    fixedBlocks
  end
```

Figure 3.3: Procedure to copy a loop

In order to handle the mismatch between the arguments passed by the *Case* transfer and the arguments the loop's header expects, an additional block is created. This block b_h takes the arguments produced by the current branch p_i and performs a *Goto* transfer to the header of the copied loop, passing it the arguments it would normally expect. Note that these arguments are in scope because they were passed to b_e. b_h is given a new label l and p_i is set to transfer to l. b_h together with the blocks from the copied loop makes up L'_i.

All of the blocks in B are removed from the CFG. The block b_e and the blocks $L'_1 \cup \ldots \cup L'_n$ are added to the CFG. This completes the work performed by the *LoopUnswitch* pass, but notice that the invariant branches were never eliminated from the copied loops. In order to eliminate those branches we take advantage of one of MLton's existing optimizations; the *KnownCase* pass. KnownCase recognizes when a *Case* transfer has been performed on a variable v_t at a previous point in the CFG and eliminates all subsequent transfers on the same v_t, choosing the branch that was taken at the previous point in the CFG. By inserting a KnownCase pass after our LoopUnswitch pass in the optimization pipeline, the duplicated branches are automatically cleaned up.

3.2.3 Example

Consider the example program in Figure 3.4. Figure 3.5 illustrates the portion of a CFG that represents the **loop** function from the source program, before a *LoopUnswitch* pass has been run. Note how block **L_281** performs a conditional branch on variable **x_451**, which is defined outside the loop. This transfer is a candidate for unswitching. Figure 3.6 illustrates the same CFG after a *LoopUnswitch* and *KnownCase* have been run. Notice how the original loop has been replaced by two loops; one that takes the branch that adds `0x2329` to the total, and another that takes the branch that adds `0x539` to the total. The program has been made slightly larger than it originally was, but the size of the loop bodies has been reduced.

```
fun demo (count, b) =
  let
    val () = if count = 0 then ()
             else (print (Int.toString count) ; demo (0, b))
    fun loop n i s =
      if i < n then
        let
          val a = if b then 1337
                  else 9001
        in
          loop n (i+1) (s+a)
        end
      else s
  in
    print (Int.toString (loop count 0 0))
  end
val x = Option.valOf (Int.fromString
        (Option.valOf (TextIO.inputLine TextIO.stdIn)))
val text = TextIO.inputLine TextIO.stdIn
val gotText = case text of NONE => false | _ => true
val () = demo (x, gotText)
```

Figure 3.4: A sample program with an unswitchable loop

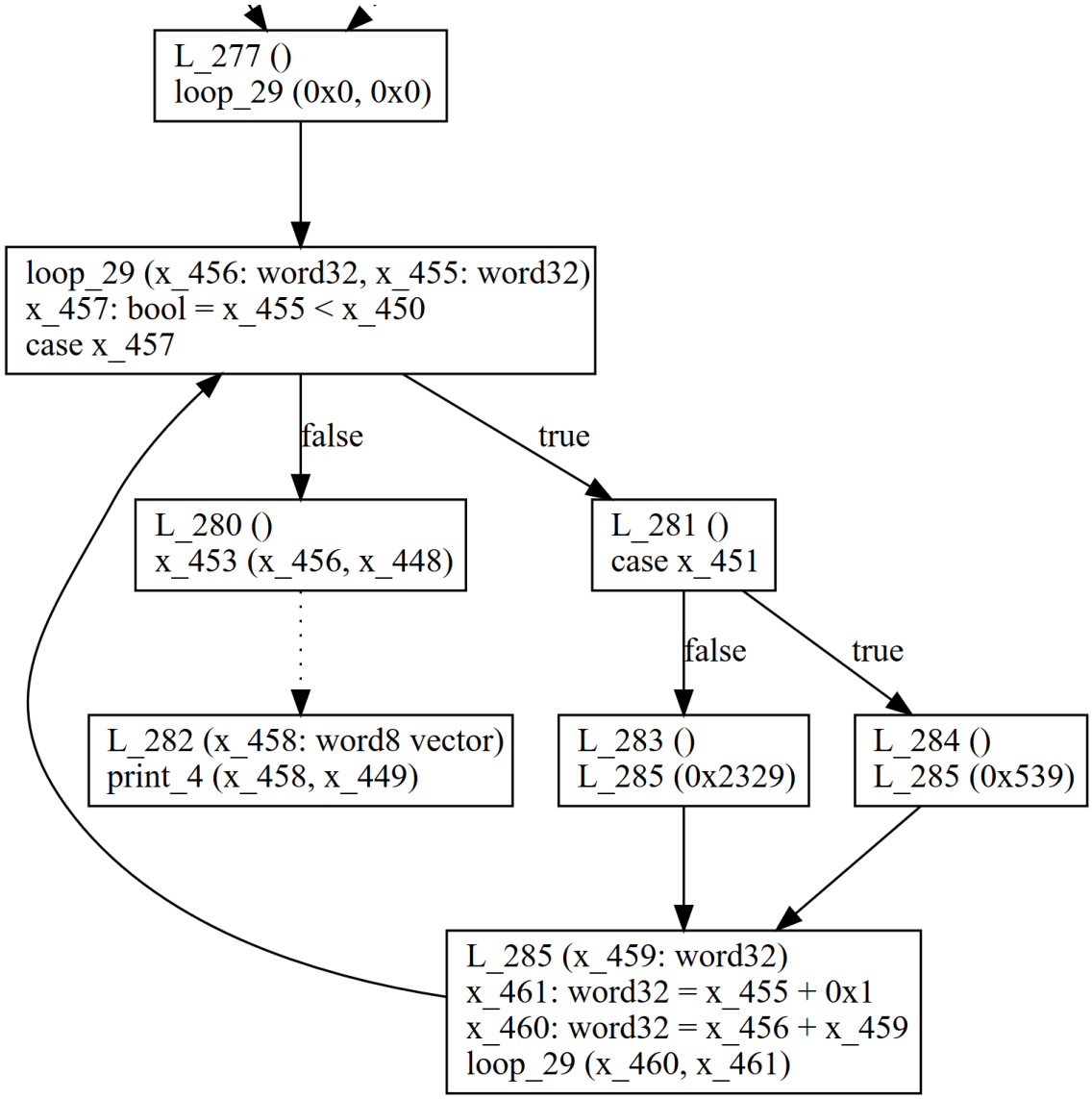

Figure 3.5: An example CFG, before Unswitching

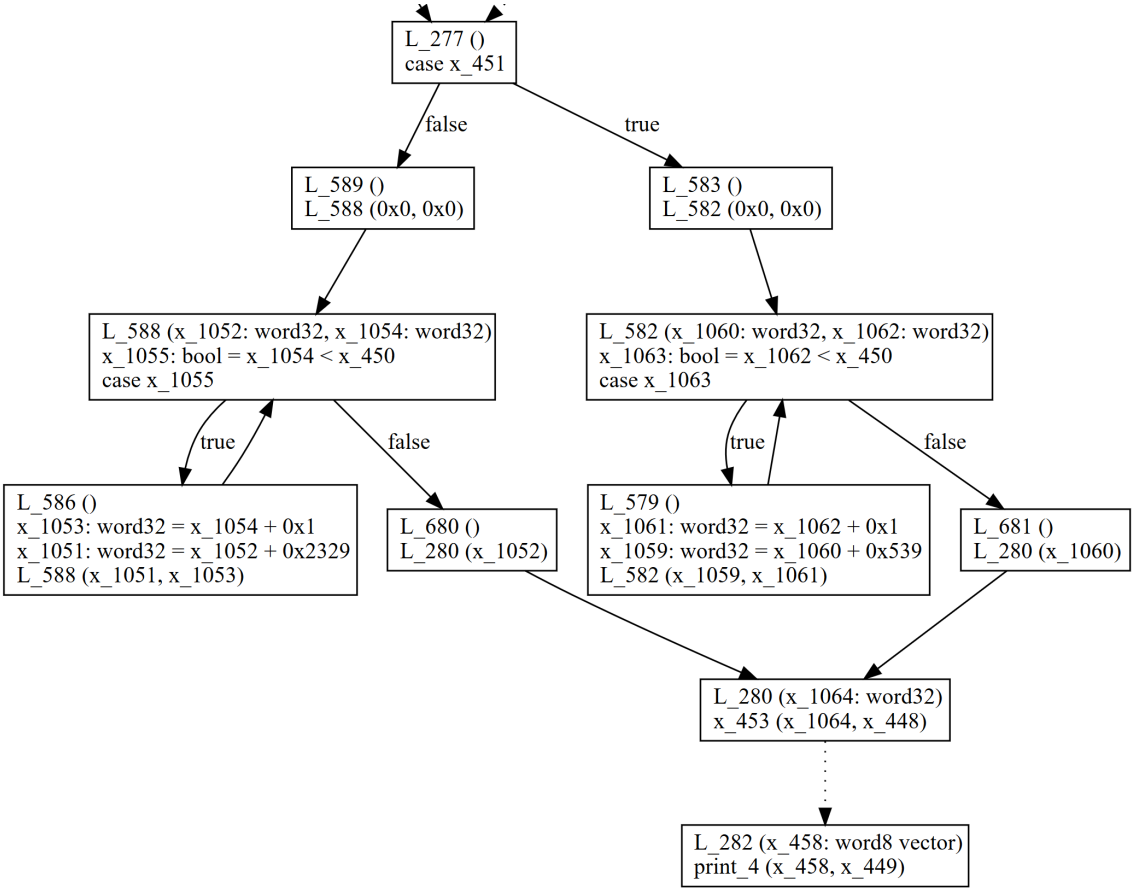

Figure 3.6: An example CFG, after Unswitching

3.2.4 Deciding to Unswitch

In order to decide if a loop $L < h, B >$ with a unswitchable transfer t should be optimized, we must compute the growth of the unswitched loops. We calculate the loop size S_L of L using the same method as *Function Inlining*. The number of branches n is equal to the number of pattern/label pairs $p_1, \ldots, p_n$ for the transfer t. If the relation $S_L \times n < X$, where X is an integer value representing the maximum acceptable code growth, is true, then the loop will be unswitched. If not, the loop will not be transformed.

The value for the constant X was determined experimentally. MLton's benchmark

suite was run for several different values of X. The value for X that produced the greatest execution time gains relative to the size of the output binary was chosen as the default value of X. A command line flag was added to MLton to support changing this value at runtime. By invoking the compiler with the flag `-loop-unswitch-limit X` the value of X can be changed.

This is not a perfect method of estimating the code growth that would result from unswitching. It fails to account for the branches in each loop copy that are eliminated by the *KnownCase* pass. Exploring better ways to determine the code growth by loop transformations is an area for future work.

3.3 Loop Unrolling

Loop unrolling is an optimization that takes advantage of compile time information about the iteration counts of loops in order to reduce their iteration count or eliminate them entirely. The optimization presented in this paper specifically targets *counting loops*; loops that begin with a known starting value, increment by a constant amount each iteration, and stop after a relation with a constant integer is no longer true. Unrolling can be applied to more advanced counting loops, such as loops with non-linear increments or loops that are missing some exact information at compile time. These loops are discussed in the Future Work section of this paper.

3.3.1 Induction Variables

In order to analyze a loop and determine if it is a counting loop suitable for unrolling, we must look at the *induction variables* of the loop. An induction variable is defined as a variable whose value is derived from the number of iterations that the loop has run [1]. In MLton's SSA form, induction variables are ϕ variables of the loop header that are transformed by some constant transformation each iteration of the loop. For the counting loops we want to unroll, the induction variables we look for are incremented by a constant integer value each iteration. The value of an induction variable v at iteration n of a loop is

$v_{initial} + n \times s$, where s is the constant step value and $v_{initial}$ is the value of v before the first iteration of the loop.

3.3.2 Detecting Unrollable loops

In order to unroll a counting loop $L < h, B >$ we need to look for an induction variable that can be unrolled on. To do so, we look for variables that have a constant starting value, a constant step between iterations, and a comparison with a constant value that either continues or exits the loop.

Starting values

In order to determine if a variable has a constant starting value, we search the CFG for all blocks b that transfer to h such that $b \notin B$. Let O be the set of the argument vectors that each transfer passes to the loop header. A variable at index i in h's argument vector can be said to have a constant starting value if $\forall o \in O(o[i] = K_s)$, where K_s is some constant value.

Steps

A variable v at index i of the loop header's argument vector can be said to have a constant step if for all transfers from a block in the loop body B to the loop header h, the value at index i of the transfer's argument vector is equal to $v+K_t$, where K_t is some constant value. In order to determine if K_t exists, we use the procedure shown in Figure 3.7. It is important to note that the procedure does not look at the ϕ variables of any block when trying to compute the step, unless the block was transferred to by a single arithmetic transfer. This ensures that the step will be consistent for every path through the loop's body.

```
(* Given:
    - a variable in the loop
    - another variable in the loop
    - the loop body
    - a function from variables to their constant values
    - a starting value, if the transfer to the header
      is an arith transfer
   Returns:
    - Some x such that the value of origVar in loop
      iteration i+1 is equal to
      (the value of origVar in iteration i) + x,
      or None if the step couldn't be computed *)
fun varChain (origVar, endVar, blocks, loadVar, total) =
   case Var.equals (origVar, endVar) of
     true => SOME (total)
   | false =>
      let
        val endVarAssign = Vector.peekMap (blocks, fn b =>
          let
              val assignments = (* assignments to endvar *)
              val arithTransfers = (* arithmetic transfers
                that get assigned to endvar *)
              val assignments' = Vector.concat
                            [assignments, arithTransfers]
          in
              case Vector.length assignments' of
                 0 => NONE
               | 1 => SOME (Vector.sub (assignments', 0))
               | _ => raise Fail (* Multiple assignments
                                    in SSA form! *)
          end)
      in
         case endVarAssign of
           NONE => NONE
         | SOME (nextVar, x) =>
            varChain(origVar, nextVar, blocks,
                     loadVar, x + total)
      end
```

Figure 3.7: Procedure to compute the step of a variable

Bounds

A **bound** is defined as some conditional transfer t on a variable v_t where one or more branches of t transfer to a block outside of B and one or more branches transfer to a block $b \in B$. v_t is the result of a comparison of an induction variable and a constant value. Additionally, the block b containing the transfer t must dominate all blocks in B that transfer to the header h. This ensures that all paths from the header through the loop and back to the header must pass through the bounding transfer.

For a counting loop, the bounding condition will be a comparison between an induction variable and a constant value using one of the following operators: $<, >, =, \not<, \not>, \neq$. MLton's SSA form only features less-than and equality operators; the other operators are derived based on the order of the arguments and whether or not the loop exits if the condition is true. For example, if the loop continues when the comparison $v < 5$ is true, then the bound is < 5. If the loop exited when $v < 5$ was true, then the bound is $\not< 5$.

In order to determine the bound of an induction variable v, examine all the transfers of blocks in B that satisfy the properties of dominating all transfers to the loop header and having one branch exit the loop while the other continues. Let T be the set of boolean variables examined by these transfers. Let $T_i \subset T$ be the set of variables that are defined as a comparison c between v and a constant value K_b. If $T_i \neq \emptyset$ then the bound of v is the comparison c against the value K_b. If $|T_i| > 1$, then the loop has multiple bounds on the same induction variable. Any one is suitable for unrolling because each bound places a hard limit on the number of iterations that a loop will run. Consider the two bounds A of < 5 and B of < 10 on variable v, where v starts at 0 and increments by 1. If we chose to completely unroll the loop on A, we would copy the body 5 times and eliminate all the comparisons of v with the condition A, because we know they will always be true. All comparisons with B would be eliminated by constant folding and propagation, as every number less than 5 is also less than 10. If we chose to unroll on B instead, we would copy the body 10 times and eliminate all comparisons of v with the condition B. As constant folding and propagation eliminate the comparisons with A they would see that eventually

A becomes false and the loop would exit. The conditional exit branch would be replaced by a goto, and the 5 additional copies of the loop's body would be eliminated as dead code.

3.3.3 The Unrolling Transformation

For a counting loop $L < h, B >$ with an induction variable v at index i in h's argument vector, start and step constants K_s and K_t, a bound comparison c and a bound constant K_b we can compute the iteration count of the loop using the procedure in Figure 3.8. The iteration count is equal to the number of steps that must be taken in order to make the bounding condition false. For loops in which the bounding condition is never true, the iteration count is 0. For a loop in which the bounding condition is true once but becomes invalidated after one step, the iteration count is 1.

It is important to allow the bounding condition to be anywhere within the loop's body so that all types of loops may be optimized. A consequence of this is that the iteration count does not completely reflect how many times a part of the loop's body will be run. For a 0 iteration loop, if the bounding condition is in the last block of the loop's body before transferring back to the header then all of the body's code will still be run once. If the bounding condition is somewhere in the middle, any code before it will be run once and any code after will not be run at all. In order to solve this problem we append an additional copy of the loop's body to the end of the unrolled loop, with the final value of the induction variable inserted in the copied loop's header. Constant propagation will replace the bounding condition's branch with a goto and any code that would normally not be run is eliminated by dead code elimination. This ensures that the final iteration of the loop will always behave normally.

```
fun iters (start, step, max) =
      let
        val range = max - start
        val iters = range div step
        val adds = range mod step
      in
        if step > range then
          1
        else
          iters + adds
      end

(* Assumes isInfiniteLoop is false,
otherwise the result is undefined. *)
fun iterCount (T {start, step, bound, invert}) =
  case bound of
    Eq b =>
      if invert then
        (b - start) div step
      else
        1
  | Lt b =>
    (case (start >= b, invert) of
      (true, false) => 0
    | (true, true) => iters (b - 1, ~step, start)
    | (false, true) => 0
    | (false, false) => iters (start, step, b))
  | Gt b =>
    (case (start <= b, invert) of
      (true, false) => 0
    | (true, true) => iters (start, step, b + 1)
    | (false, true) => 0
    | (false, false) => iters (b, ~step, start))
```

Figure 3.8: Procedure to compute the number of loop iterations

Deciding Between Partial and Total Unrolling

If a loop's iteration count is 0 or 1, we always totally unroll the loop. The overhead of the loop is unnecessary and the code size almost always goes down. For other iteration counts, in order to decide if a loop should be partially unrolled or totally unrolled we must first compute the size of the completely unrolled loop. We calculate the loop size S_L of L using the same method as ***Function Inlining***. If the relation $S_L \times iteration\_count < Y$, where Y is an integer value representing the maximum acceptable code growth, is true, then then loop will be completely unrolled. If not, an unrolling factor u will be determined such that $S_L \times u < X$, where X is an integer approximating the size of the instruction cache. For the purposes of this paper $Y = X$; it is not a requirement for these values to be equal but the value of X acts as a conservative default value for Y. We find constants A and B such that $A \times u + B = iteration\_count$ and $B < u$. A is the number of times the partially unrolled loop will run and B is the number of extra iterations that will be completely unrolled and appended to the front of the partially unrolled loop.

The value for the constant X was determined experimentally. MLton's benchmark suite was run for several different values of X. The value for X that produced significant performance gains in the most programs without causing performance drops in any benchmark was chosen as the default value of X. X is an approximation of the instruction cache size, so increasing X may lead to partially unrolled loops overflowing the cache. Significant performance decreases were seen in some benchmarks with larger values of X due to cache misses in partially unrolled loops. A command line flag was added to MLton to support changing this value at runtime. By invoking the compiler with the flag `-loop-unroll-limit X` the value of X can be changed.

This is not a perfect method of deciding how loop unrolling should be performed. Notice that the same X is used for both deciding if a loop should be totally unrolled and how large the body of a partially unrolled loop should be. These values do not have to be the same and it is reasonable to think that each use case would have a distinct optimal value.

Additionally, the target architecture is not considered when trying to limit the size of a partially unrolled loop. Different architectures may have different instruction cache sizes, so it would be wise to consider the target architecture when deciding.

Total Unrolling

In order to completely unroll a loop we begin by first computing the values of the induction variable v for each iteration of the loop. Let V be the set of values v takes at each iteration of the loop, where $|V| = iteration\_count + 1$. From V we can produce a list of statements S that each perform an assignment of the constant value $k_i \in V$ to a new variable s_i. Next, create a block b_e with the same argument vector as h, statements S, the same label as h, and a *Goto* transfer to an undefined label. This block will serve as the entry point to the unrolled loop. Finally, create a block b_x with the same argument vector as h, no statements, a new label, and a *Bug* transfer. This block will be eliminated by dead code elimination, but it is necessary for the final copy of the loop's block to transfer to somewhere.

For each $s_i \in S$, copy the loop's body using a similar procedure to the one shown in Figure 3.3. There are two key modifications to the previous algorithm. The first is that in the copy's header, a statement is inserted that assigns the variable s_i to the induction variable v. This allows constant propagation to replace all instances of the induction variable in the copied loop with its actual value. The restore function described in Section 3.1.4 will handle the violation of the SSA invariants. The second is that any transfers to h are rewritten as transfers to the header of the *next* copy of the loop. In the last copy of the loop, transfers to the header are directed to b_x. The transfer of b_e is rewritten to the header of the first copy of the loop.

The copied loops, b_e, and b_x form the unrolled loop. The blocks B from the original loop are removed from the CFG and the unrolled loop is inserted. The SSA restore function fixes variables with more than one definition and MLton's *Shrink* function performs constant folding and propagation. This completes the work done by the *LoopUnroll* pass.

Partial Unrolling

If we have determined that a loop should be partially unrolled with an unrolling factor of u, a partial iteration count of A, and an extra number of iterations B, we can partially unroll the loop as follows. First we perform the total unrolling transformation once using only the final value of the induction variable v. This produces a set of blocks B_x that perform any work done by the last iteration of the loop and follows the bound's exit path. Next we produce an expanded loop body by performing the total unrolling transformation using the induction variable values $v + K_t \times i$ for $0 \leq i < u$. Within the expanded loop we do not know the values of the induction variable ahead of time, so constant propagation will not be able to eliminate the bound's conditional branch. However, we do know that they will always satisfy the bounding condition so we can rewrite the bound's conditional branch ourselves to always take the path that continues the loop.

The expanded loop is nested between three new blocks b_{Ph}, b_{Pt}, and b_{Px}. b_{Ph} serves as the header for the expanded loop and has the same argument vector as h with the addition of an additional argument a. b_{Ph} has no statements and simply transfers to the head of the expanded loop. b_{Pt} increments a by 1 and transfers to b_{Ph}. b_{Px} performs a conditional branch on $a < A$ and transfers to b_{Pt} if true and the header of B_x if false. b_{Ph}, b_{Pt}, b_{Px}, and the expanded loop blocks form the set of blocks B_e.

The total unrolling transformation is performed once more using the first B induction variable values in order to form the set of blocks B_p. The final transfer in B_p is rewritten to transfer to b_{Ph}. $B_p \cup B_e \cup B_x$ forms the unrolled loop. The blocks B from the original loop are removed from the CFG and the unrolled loop is inserted. The SSA restore function fixes any SSA violations and the shrink function performs constant propagation. This completes the work done by the *LoopUnroll* pass.

Partially unrolling a loop introduces opportunities for the *Common subexpression elimination* optimization to simplify the body of the partially unrolled loop. The extra iterations of the loop that were completely unrolled may reveal some loop invariant expressions that can be eliminated in the partially unrolled loop. In order to take advantage of these potential

opportunities, the *CommonSubexp* pass is run after loop unrolling. In order to ensure that these opportunities are made available, we always completely unroll at least one iteration of a partially unrolled loop. If $B = 0$, we decrement A by 1 and increase B by u so that some copies of the loop's body are always completely unrolled.

3.3.4 Example

Consider the example program in Figure 3.9. Figure 3.10 illustrates the portion of a CFG

```
fun demo count =
  let
    val () = if count = 0 then ()
             else (print (Int.toString count) ; demo 0)
    fun loop n i =
      if i < n then
        (print (Int.toString i) ; loop n (i+1))
      else ()
  in
    loop 2 0
  end
val () = demo 1
```

Figure 3.9: A sample program with an unrollable loop

that represents the **loop** function from the source program, before a *LoopUnroll* pass has been run. Notice the variable `x_100`; it is an induction variable of the loop with header `loop_7` that begins at 0, steps by 1, and has a bound of < 2. Figure 3.11 illustrates the result of unrolling the loop on `x_100`. Notice how the loop has been completely unrolled and replaced by a linear set of blocks. The values of the induction variable have been inserted as constants in block `L_59`.

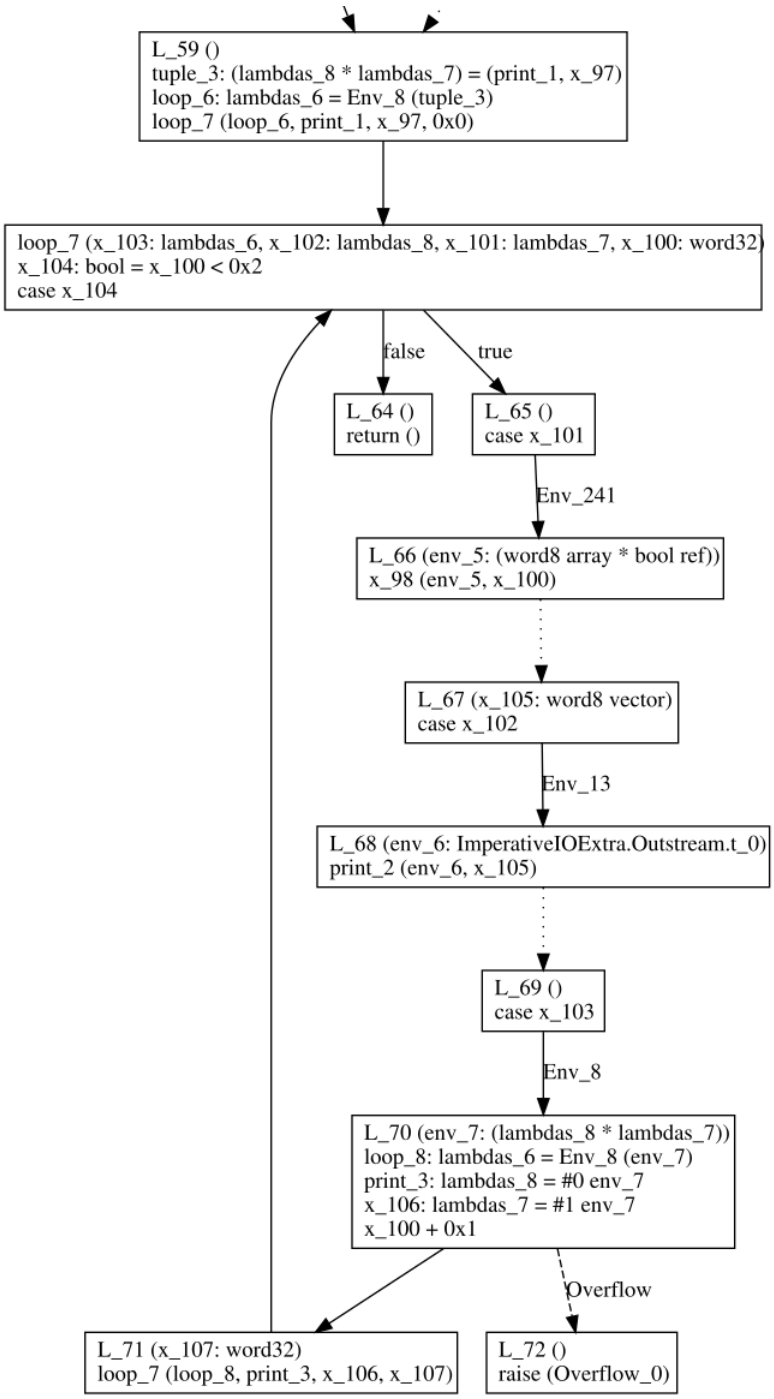

Figure 3.10: An example CFG, before Unrolling

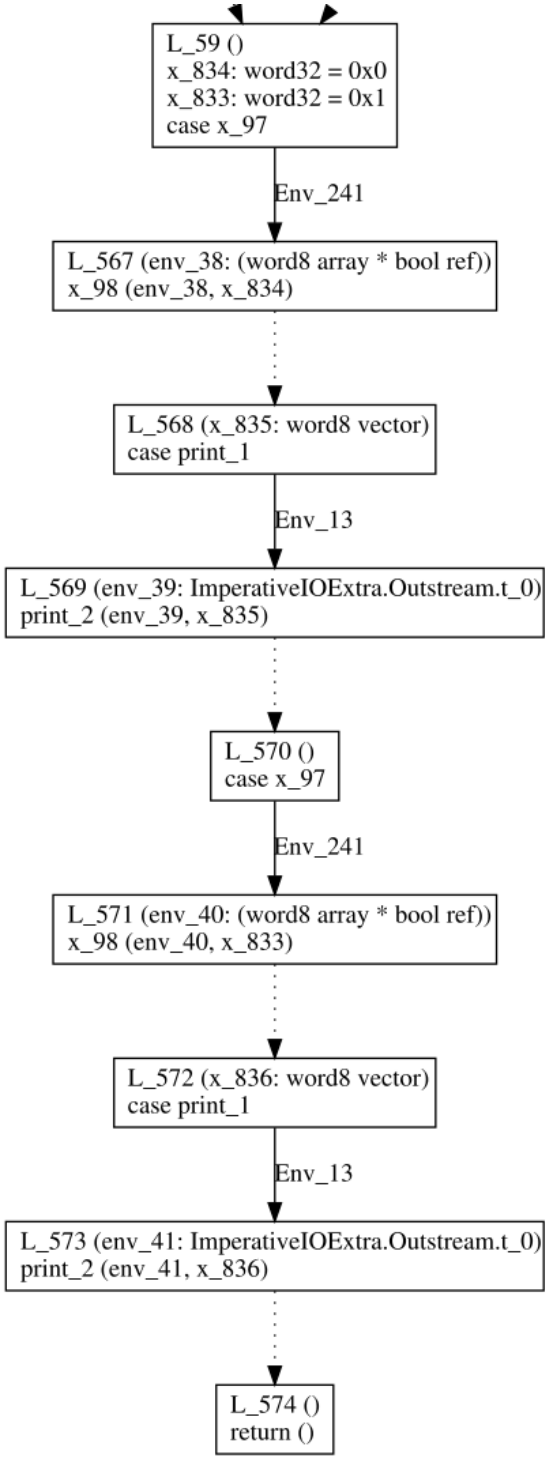

Figure 3.11: An example CFG, after Unrolling

Chapter 4

Analysis

This chapter evaluates the benefits of loop unrolling and loop unswitching by evaluating the performance of programs compiled with these optimizations enabled. The two metrics we evaluate are execution time and binary size; recall that our loop optimizations sought to increase performance at the cost of code size. Results are gathered from MLton's existing suite of benchmark tests. We also examine statistics gathered by the optimization passes as they compiled MLton itself.

4.1 Benchmarks

MLton's benchmark suite consists of 43 different benchmarks. The suite measures performance in three areas; run time, binary size, and compile time. For the purposes of this project we are only interested in the run time and binary size. Results were gathered on the Computer Science department's server `glados`. `glados`'s specs can be seen in Figure 4.1.

4.1.1 Run Time Performance

Figure 4.2 shows the run time ratios of the benchmarks that had significant changes in run time with unrolling, unswitching, and both optimization passes enabled. The baseline represents the compiler with unrolling and unswitching disabled. 11 of the benchmarks produced significant results ($\geq 5\%$ change in run time); the other 32 showed no significant improvements. The full set of results is shown in Table 4.1. The results show that loop

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                16
On-line CPU(s) list:   0-15
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 26
Stepping:              5
CPU MHz:               2801.000
BogoMIPS:              5585.73
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0-3,8-11
NUMA node1 CPU(s):     4-7,12-15
Memory:                36199M
```

Figure 4.1: Output of `lscpu` on `glados` and memory information

unrolling produced very good improvements on several tests, while loop unswitching only produced marginal improvements on its own.

One possible reason for loop unswitching's low performance increases is the effectiveness of a modern processor's branch prediction. If a loop contains a branch on an invariant condition, the processor will likely predict that branch correctly for every iteration of the loop. The benefits of unswitching are therefore more likely to come from the optimizations it enables by simplifying loops rather than just the removal of the branch.

Loop unrolling produced significant improvements on 11 of the 43 benchmarks and

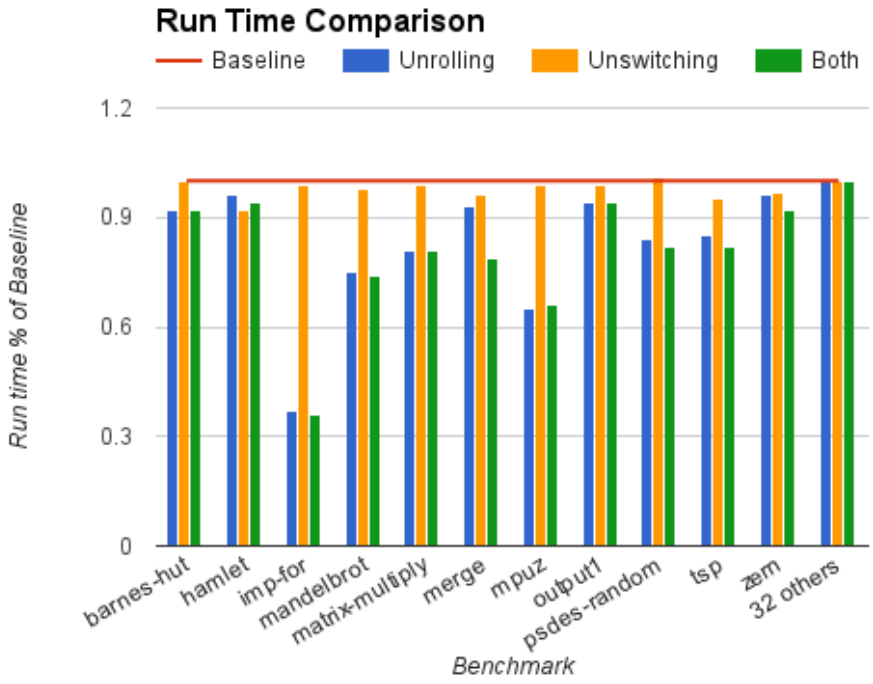

Figure 4.2: Graph of benchmark run time ratios

improvements of 25% or greater on 3 of them, the greatest of which being a 63% improvement on `imp-for`. `imp-for` tests nested imperative style for-loops with known iteration counts; these loops are prime candidates for unrolling and illustrate the significant benefits associated with unrolling an innermost nested loop.

4.1.2 Binary Size Performance

Figure 4.3 shows the binary size ratios of the benchmarks that showed significant changes in performance. The full results are shown in Table 4.2. As you can see, the increases in binary size are very small. The largest increase in binary size was 5% and occurred in a single benchmark, `vector-rev`. Overall, the benchmarks saw an average of a 1% increase in binary size when compiled with both optimizations enabled. These results show that significant performance gains were made at the cost of small increases in code size.

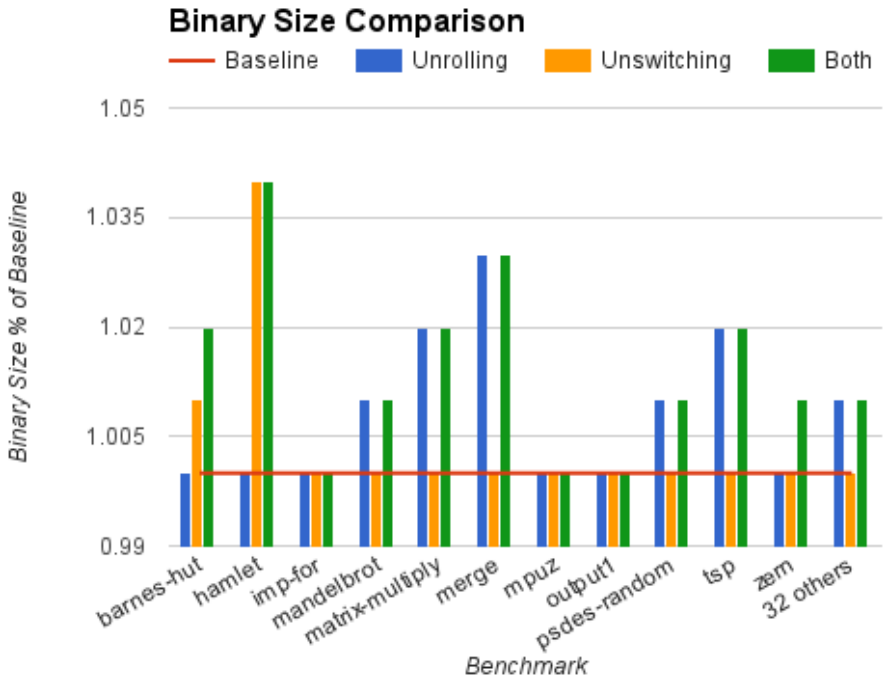

Figure 4.3: Graph of benchmark binary size ratios

4.2 Loop Statistics

In order to understand how many loops within a program are able to be unrolled or unswitched, the optimization passes collect diagnostic info about the loops they encounter. In order to understand what types of loops are found in a large, real-world SML program, we look at the diagnostic output generated from compiling MLton itself.

A single loop unswitching/unrolling pass was run at the end of MLton's SSA pipeline for the purpose of gathering statistics. Of the 11394 innermost loops:

- 946 loops had multiple headers
- 332 loops were totally unrolled
- 24 loops were partially unrolled
- 19578 loop arguments were not constant integers

- 1697 loops with constant starts had non-computable steps
- 3077 loops with constant starts and steps had non-computable bounds
- 130 loops were unswitched
- 55 loops were too big to unswitch
- Within these 11394 loops there were 21241 branches on loop-variant conditions

We can see that not many loops can be unswitched despite the fact that loops have almost 2 conditional branches per loop on average. This highlights the weakness of unswitching in the context of MLton, which does not have a powerful loop invariant code motion pass. MLton's current *LoopInvariant* pass only eliminates ϕ values that are invariant. In order to expose more unswitchable loops, more powerful loop invariant code motion is needed.

About 3% of the loops in MLton met the requirements for loop unrolling to optimize them. Most of the arguments passed to loops are not constant integers. Of those that are, many are not incremented by a constant value each iteration of the loop. Many more do not have a compile time limit on how many times the loop will iterate. We can see that most loops are not simple counting loops, therefore a more general method of determining loop iteration counts is needed to expose more unrollable loops.

Of the loops we could unroll, their iteration counts were also recorded. The results are shown in Table 4.3. Interestingly, loops of iteration count 1 appear very frequently. One would not expect programmers to place code that only runs once inside a loop, so this may be the result of MLton's various transformations and optimization passes producing CFGs that happen to follow this pattern.

Table 4.1: Run Time Ratios

benchmark	Baseline	Unrolling	Unswitching	Both
barnes-hut	1.00	0.92	1.00	0.92
boyer	1.00	1.03	1.00	1.03
checksum	1.00	1.00	1.00	1.00
count-graphs	1.00	0.99	0.99	0.99
DLXSimulator	1.00	1.00	0.99	1.00
even-odd	1.00	0.99	0.99	0.99
fft	1.00	1.00	0.98	0.97
fib	1.00	0.98	0.97	1.02
flat-array	1.00	1.01	1.01	0.99
hamlet	1.00	0.96	0.92	0.94
imp-for	1.00	0.37	0.99	0.36
knuth-bendix	1.00	0.99	1.00	0.99
lexgen	1.00	0.99	0.99	1.04
life	1.00	1.00	1.01	1.00
logic	1.00	0.98	0.99	1.00
mandelbrot	1.00	0.75	0.98	0.74
matrix-multiply	1.00	0.81	0.99	0.81
md5	1.00	1.00	1.01	1.00
merge	1.00	0.93	0.96	0.79
mlyacc	1.00	1.00	0.99	1.01
model-elimination	1.00	1.01	0.99	1.00
mpuz	1.00	0.65	0.99	0.66
nucleic	1.00	1.01	1.02	1.00
output1	1.00	0.94	0.99	0.94
peek	1.00	1.00	1.00	1.00
psdes-random	1.00	0.84	1.01	0.82
ratio-regions	1.00	1.00	0.99	1.00
ray	1.00	1.00	1.00	1.00
raytrace	1.00	1.00	1.00	1.00
simple	1.00	1.00	0.98	0.99
smith-normal-form	1.00	1.02	1.03	1.03
tailfib	1.00	1.00	1.00	1.00
tak	1.00	1.01	1.10	1.10
tensor	1.00	1.00	1.00	1.00
tsp	1.00	0.85	0.95	0.82
tyan	1.00	0.99	0.99	1.01
vector-concat	1.00	1.00	1.00	1.00
vector-rev	1.00	1.00	1.00	0.99
vliw	1.00	1.00	1.02	1.00
wc-input1	1.00	1.00	0.99	1.02
wc-scanStream	1.00	1.01	1.00	1.00
zebra	1.00	1.02	1.01	1.01
zern	1.00	0.96	0.97	0.92

Table 4.2: Binary Size Ratios

benchmark	Baseline	Unrolling	Unswitching	Both
barnes-hut	1.00	1.00	1.01	1.02
boyer	1.00	1.02	1.00	1.02
checksum	1.00	1.00	1.00	1.00
count-graphs	1.00	1.00	1.00	1.00
DLXSimulator	1.00	1.01	1.00	1.01
even-odd	1.00	1.02	1.00	1.02
fft	1.00	1.00	1.00	1.00
fib	1.00	1.01	1.00	1.02
flat-array	1.00	1.00	1.00	1.00
hamlet	1.00	1.00	1.04	1.04
imp-for	1.00	1.00	1.00	1.00
knuth-bendix	1.00	1.00	1.00	1.00
lexgen	1.00	1.00	1.01	1.01
life	1.00	1.00	1.00	1.00
logic	1.00	1.00	1.00	1.00
mandelbrot	1.00	1.01	1.00	1.01
matrix-multiply	1.00	1.02	1.00	1.02
md5	1.00	1.00	1.00	1.01
merge	1.00	1.03	1.00	1.03
mlyacc	1.00	1.00	1.01	1.01
model-elimination	1.00	1.00	1.00	1.00
mpuz	1.00	1.00	1.00	1.00
nucleic	1.00	1.00	1.00	1.00
output1	1.00	1.00	1.00	1.00
peek	1.00	1.00	1.00	1.00
psdes-random	1.00	1.01	1.00	1.01
ratio-regions	1.00	1.01	1.00	1.01
ray	1.00	1.00	1.00	1.01
raytrace	1.00	1.00	1.01	1.01
simple	1.00	1.00	1.00	1.00
smith-normal-form	1.00	1.01	1.00	1.01
tailfib	1.00	1.00	1.00	1.00
tak	1.00	1.00	1.00	1.00
tensor	1.00	1.00	1.00	1.00
tsp	1.00	1.02	1.00	1.02
tyan	1.00	1.02	1.00	1.03
vector-concat	1.00	1.00	1.00	1.00
vector-rev	1.00	1.05	1.00	1.05
vliw	1.00	1.00	1.01	1.01
wc-input1	1.00	1.01	1.01	1.01
wc-scanStream	1.00	1.00	1.01	1.01
zebra	1.00	1.00	1.00	1.01
zern	1.00	1.00	1.00	1.01

Table 4.3: Loop Iteration Count Frequencies

Iteration Count	Frequency
1	209
2	83
3	20
4	13
5	4
8	3
10	1
14	1
22	1
64	1
65	9
84	1
85	1
256	7
622	1
1024	1

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This report described two loop optimizations, loop unrolling and loop unswitching, and how they can be implemented in the MLton Standard ML compiler. The goal of this project was to implement both optimizations as optimization passes on MLton's SSA intermediate form and measure how they affect the performance of programs compiled with these optimizations enabled.

Loop unswitching aimed to remove conditional branches on invariant values from within loops. By duplicating the loop's body for each branch, the branches were eliminated at the cost of a small increase in code size. While loop unswitching alone does not result in significant performance increases, the duplicated loops are simpler than the original, allowing other optimization passes to act upon them. Loop unswitching requires the variable being branched upon be defined outside the loop, yet many loops do not meet this requirement. This severely limits the number of loops that can be optimized.

Loop unrolling tried to figure out how many times a loop would iterate at compile time and transform the loop based on that knowledge. If a loop was small enough and ran for a small number of iterations, the loop could be eliminated completely and be replaced by straight line code. This allows constant propagation and folding to insert the correct values for the induction variable everywhere, significantly increasing performance. Loops that are too large to unroll completely are unrolled partially instead. By duplicating the loop's body and forming a larger loop the number of iterations that the loop performs is significantly

reduced. Completely unrolling a few of these iterations gives the common subexpression elimination pass additional opportunities to simplify the body of the expanded loop. The results show that loop unrolling achieved significant performance increases at the cost of a small increase in binary size.

5.2 Future Work

We have shown that the loop unrolling and loop unswitching transformations can significantly improve program performance without a significant increase in binary size. Now that the transformations themselves are implemented, the focus turns to making as many loops as possible available for transformation and taking advantage of the opportunities exposed by the transformations.

In order to make as many loops unswitchable as possible, a more powerful form of loop invariant code motion is needed than what already exists in MLton. Currently MLton only eliminates the ϕ values of loops that stay constant for all loop iterations. A more powerful optimization would target variables defined within the loop's body that are computed to have the same value for every iteration of the loop. This is not a trivial problem to solve; performing the computation outside the loop may have side-effects and it doesn't make sense to perform computations for a loop that never runs. However, this type of pass would benefit most loop optimizations.

Loop unrolling as it is described in this paper only transforms counting loops, but the transformation is valid for any loop where knowledge of the iteration count is available. For example, if the exact number of iterations is unknown but we know it is a multiple of 5, then we could perform a modified partial unrolling transformation; duplicating the loop's body 5 times and placing it inside an infinite loop. In a language like SML, it is common for programmers to write recursive functions that deconstruct lists and other finite data structures. These functions become loops that could be unrolled if the size of the data structure could be determined at compile time. In an existing paper, Lokuciejewski describes an approach for gaining knowledge about the iteration counts of loops through

the use of an analysis based off of *Abstract Interpretation* and utilizes that knowledge to perform loop unrolling [4]. A more powerful analysis of loop iteration counts would benefit many loop optimizations, such as *loop fusion*, which combines two loops with equal iteration counts into one[1].

Loop Peeling is an optimization that *peels* a number of iterations off of a loop[1]. This is a similar to what is done to the extra iterations of a partially unrolled loop. Peeling off an iteration or two can help expose opportunities for common subexpression elimination to simplify the body of the loop. Additionally, if the large number of counting loops with an iteration count of 1 is not an occurrence unique to counting loops, then it may be beneficial to peel off two iterations of every loop. If many loops exit after a single iteration, then this transformation may be able to eliminate many loops from the program.

In order to avoid hurting performance by performing a transformation like loop unrolling, accurate knowledge about the size of the program is needed. However, the exact size of a program isn't known until the program is being processed by the back-end of the compiler. Lokuciejewski proposes a solution to this problem called **Back-Annotation**[3]. Back-Annotation establishes a link between the high-level and low-level intermediate representations, making information from the assembly level available to optimization passes working at a higher level. Having this knowledge available in MLton's SSA form would allow optimizations like loop unrolling and function inlining to make better decisions.

A partially unrolled loop exposes several opportunities for optimizations that improve memory efficiency. In particular, optimizations that improve memory parallelism can target the larger bodies of partially unrolled loops. Modern vector and superscalar machines feature instructions that allow for multiple values to be loaded from memory into registers at once[1]. An optimization pass that reordered memory accesses into groups and then transformed those multiple operations into a single one could result in significant performance improvements. Once those values are loaded into registers, SIMD operations could be introduced to operate on all of those values at once. These optimizations would not be possible if only a single memory access was performed per loop iteration.

Bibliography

[1] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994.

[2] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[3] P. Lokuciejewski and P. Marwedel. Combining worst-case timing models, loop unrolling, and static loop analysis for wcet minimization. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 35–44, July 2009.

[4] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 136–146, Washington, DC, USA, 2009. IEEE Computer Society.

[5] G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Trans. Program. Lang. Syst.*, 24(5):455–490, September 2002.