Loop Optimizations for MLton
Matthew Surawski (mjs9585@rit.edu) | Advisor: Dr. Matthew Fluet

Introduction
Loops are one of the fundamental control flow structures for building programs. As such, they are popular targets for optimizing compilers. Loops can be expanded to reduce their iteration counts, or their bodies can be simplified to reduce the amount of work done each iteration. By expanding the program size, the runtime performance of the loops can be improved. This project aims to incorporate two common loop optimizations, unrolling and unswitching, into the MLton Standard ML compiler.

Background
- Standard ML - a functional, statically typed, general purpose language with type inference
- MLton - an open source, whole program, optimizing SML compiler
- MLton aims to produce small, high-performance binaries
- Programs are transformed through several intermediate representations. This project targets the Static Single Assignment (SSA) form
- Many optimization passes are run sequentially on the SSA representation of the program
- Each pass should perform a single optimization well
- Some passes are run multiple times in order to take advantage of optimization opportunities exposed by other passes
- The SML language does not have loops, but the tail-call optimization pass introduces them
- In order to find loops in a program’s arbitrary control flow graph, Steensgaard’s loop forest algorithm is used

Loop Unrolling
Loop Unrolling transforms a loop with a known iteration count into straight line code, or a larger loop of fewer iterations.
- A completely unrolled loop is larger, but has no overhead
- A partially unrolled loop reduces the iteration count by a factor of \( n \) and makes better use of the instruction cache
- The left figure illustrates a loop that runs twice, the right figure illustrates the unrolled loop

Loop Unswitching
Loop Unswitching transforms a loop with a branch on an invariant condition into two loops with no branch.
- Code size grows from copying the loops, but the loops become smaller
- The processor does not have to decide which branch to take during every iteration of the loop
- The left figure illustrates a loop pre-unswitching, the right post-unswitching

Results
- The loop optimizations increase performance with minimal code growth
- Certain benchmarks are heavily optimized by the new passes
- Most loops don’t have sufficient information at compile time for unrolling. Future work should expand the number of optimizable loops

References