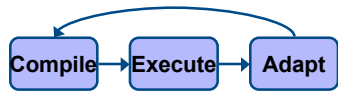


Adaptive Compilation Insights into Compiler Implementation

Jeff Sandoval (jasandov@rice.edu) Keith Cooper (keith@rice.edu)
Department of Computer Science, Rice University, Houston, TX

Introduction

By ignoring assumptions about compiler optimizations, adaptive compilation research has offered unexpected insight into the improvement of our research compiler. This poster presents the addition of a loop-unroller to our compiler, prompted by surprising behavior. This new pass changes the search space, enabling the adaptive compiler to find better solutions more quickly.



Adaptive Compilation

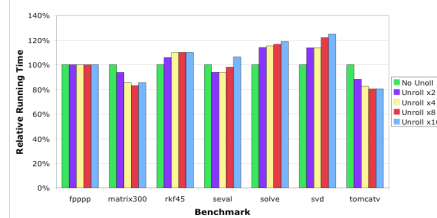
- Selects and orders compiler transformations using AI search techniques, such as hill climbers and genetic algorithms
- Maximizes an objective function (often execution time) with a feedback-directed compile-execute-adapt cycle
- Exploits the specific context of each input code individually
- Benefits from interaction between compiler transformations; some create and destroy opportunity, while others exploit existing opportunities
- Makes no assumptions about the effects of each pass; freely explores all profitable transformation sequences

Methodology

- The loop-unroller is written as independent pass, applicable at any point in the compilation process
- Experiments apply the loop-unroller in two contexts, a fixed optimization sequence and an adaptive hill climber
- The fixed sequence, loop-unrolling followed by a standard optimization sequence, is applied to several benchmarks
- The hill climber is applied to `tomcatv` only, with 40 random restarts and 20% patience

Results

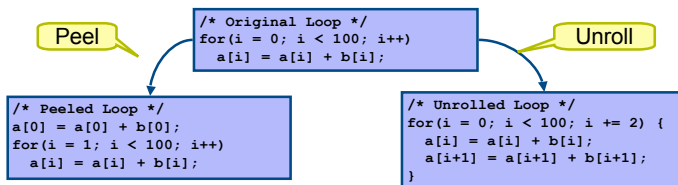
Loop Unrolling Applied in a Fixed Optimization Sequence



- Loop-unrolling works well on some benchmarks and poorly on others; indicates loop-unrolling is a good candidate for adaptation
- Poor performance on `solve` and `svd` are due to low loop iteration counts; adaptive compiler will only apply when profitable

The Problem

- Adaptive compiler exhibited unexpected behavior, loop-peel selected multiple times in a sequence
- Loop-peel originally intended as conditioning pass for loop-unswitching and Cytron-Lowry-Zadeck code motion
- Adapter found unintended use, reducing loop overhead
- Each application of loop-peel can reduce loop overhead for a single iteration of a loop
- This behavior suggests that the compiler needs a more powerful transformation



Loop-Peeling:

- Clones the body of a loop outside the loop
- Moves first iteration outside the loop body
- Does not modify loop body
- Reduces loop overhead for a single iteration
- Often selected multiple times by the adaptive compiler

The Solution

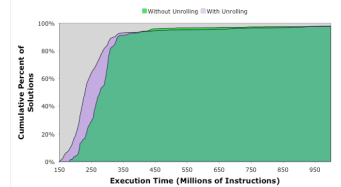
- Loop-unrolling is better suited than loop-peeling for reducing loop overhead
- Each application of loop-unroll can reduce the loop overhead for the entire loop
- Loop-unrolling is capable of producing a much larger improvement in the code
- Adaptive search can use loop-unrolling to find better solutions more quickly
- Loop-unrolling is not effective on all programs, so the adaptive compiler will discover when it is profitable to apply

Loop-Unrolling:

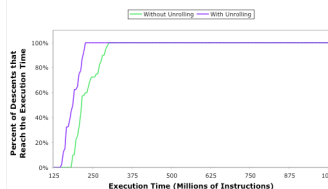
- Clones the body of a loop inside the loop
- Doubles the size of loop body
- Halves the iteration count
- Reduces loop overhead for the entire loop
- Creates new opportunities for optimization in the body of the loop

- The graph to the right shows the distribution of solutions found while running the hill climber with and without loop-unrolling
- The shaded region shows that with loop-unrolling the hill climber explores a subset of the space that contains better solutions

Cumulative Distribution of Search Space Explored with Hill Climber



Percent of Hill Climber Descents that Reach a Given Execution Time



Average Evaluations Needed for a Hill Climber to Reach an Execution Time



- The percentage of hill climber descents that reach a given solution threshold are graphed above
- Loop-unrolling allows individual descents to reach better solutions more often than without loop-unrolling

- The average evaluations a hill climber descent needs to reach a given solution threshold are graphed above
- Loop-unrolling allows a hill climber to reach similar quality solutions faster than without loop-unrolling

Difficulties

- Loop upper bound may not be known
- To fit into our adaptive compiler framework, the loop-unroller must operate on low-level (assembly-like) intermediate representation
- Loop-unroller must be applicable at any point in the compilation sequence; irregular control flow is likely

```

/* Unknown upper bound */
for(i = 0; i < (n-1); i += 2) {
    a[i] = a[i] + b[i];
    a[i+1] = a[i+1] + b[i+1];
}
/* Cleanup loop */
while(i < n)
    a[i] = a[i] + b[i];
    
```

Conclusion

- Unexpected behavior of the adaptive compiler applied loop-peel multiple times in a row
- This suggested the addition of a loop-unrolling pass
- Loop-unrolling is well suited for adaptive compilation because effectiveness is determined by input program

- Adding a pass changes the search space
- Results show that the hill climber performs better in this new space
 - The hill climber can find better solutions with loop-unrolling
 - The hill climber can reach a similar quality solution faster with loop-unrolling