HOLA: Heuristics for Optimizing Loop Unroll Factors using Machine Learning Algorithm

ABSTRACT

Compilers rely on efficiency of the intermediate passes to optimize the machine code. Owing to emergence of high performance architectures today, there are many opportunities available to the compiler for optimizing the machine code. We focus on loop unrolling, one such compiler optimization, to extract parallelism and enable better instruction scheduling. Owing to a huge search space in machine code, machine learning algorithms can account for dependencies between several optimization characteristics at once with the same runtime cost. In order to evaluate the effectiveness of machine learning techniques for optimizing compiler passes, we develop heuristics for loop unrolling using random forest algorithm. We use LLVM Compiler v3.3 to implement and evaluate our heuristics using SPEC 2006 benchmarks. We obtain a speedup of XX percent on our testing set of benchmarks over a baseline LLVM compiler. In addition, even with our limited training set, we obtain performance comparable to the loop unroll pass that is inbuilt in LLVM.

1. INTRODUCTION

Researchers, for generations, have found efficient solutions to optimize machine code generated by a compiler. Multiple traditional optimizations such as loop inlining, loop unrolling, speculative loop invariant code motion, code block ordering, etc. enable vast opportunities to optimize machine code. Majority of these optimizations, exploit parallelism in the machine code so that the instructions can be scheduled efficiently in the modern superscalar processors. However, these optimizations are computationally intensive and can often result in degraded performance when done aggressively. As a result, it is important to regulate these optimizations by implementing intelligent heuristics to determine the sections of the code for which the optimizations can be profitable.

Emerging machine learning techniques are known to analyze a huge search space of data and implement intelligent predicting models. These techniques can evaluate multiple decision variables at once and can account for dependencies between several such variables. The goal of our project is to explore a machine learning technique as an efficient alternative to develop heuristics for compiler optimization passes. We use supervised learning technique to obtain the heuristics. Supervised learning techniques train heuristics over a labeled data consisting of a vector of heuristics and desired output value. As a case study, we develop heuristics for loop unrolling using a random forest machine learning technique. Random forest technique works on the principle of decisions trees; it generates multiple decision trees, obtains classification from each of them, and chooses the classification that gets majority of the votes. Unlike decision trees, random forests are not prone to overfitting and hence a better candidate while deploying a heuristic working on a wide variety of data set.

We train a random forest heuristic on a subset of loops in SPEC 2006 benchmarks. SPEC 2006 benchmarks are used widely by the research community to evaluate performance of both, the compiler optimizations and the underlying hardware. We divide a set of applications in SPEC 2006 benchmark in two categories training set and testing set. We generate a set of features in a loop and unroll factor that enables the best runtime of that loop over the training set to obtain a labeled data. We use this labeled data to train the heuristic and finally deploy it on testing set for evaluation.

2. LOOP UNROLLING

Loop unrolling is a widely used compiler optimization pass targeting simpler loops in the program. Programs spend a significant duration of their execution time in the loops and hence, it is important to efficiently schedule instructions in a loop to exploit parallelism in them. Due to the iterative nature of the loop, control flow has to branch to the header every time the loop condition is satisfied. These branches are expensive and result in underutilization of hardware optimizations such as superscalar processors, load-store coalescing, cache locality etc. Loop unrolling technique expands the loops several times while reducing the number of iterations and consequently, diminishing the number of branches. The advantages of loop unrolling are several as listed below:

- Better instruction scheduling as the window to schedule consecutive instruction increases. Hardware can exploit higher ILP.
- Most loops load and store values to an array of memory locations. Loop unrolling can enable the coalescing of such memory operations in the hardware.
- Reduction in the number of iterations results in fewer number of branches to the loop header. The hardware can load consecutive instructions in the memory without change in the control flow.

Loop unrolling, however, results in some of the obvious side-effects that can result in degraded performance:

- A code bloat leading to higher memory pressure.
- Increased code size and limited hardware register resulting in spilling of variables on the stack. Stack is accessed slower than register resulting in degraded program performance.

• A loop can exit at multiple places; compiler has to insert complex control flow for early exits from the program.

Aggressive loop unrolling can lead to degraded performance of an application. We develop heuristics to determine the profitable loop candidates for unrolling at the compile time.

3. EXPERIMENTAL METHODOLOGY

We develop a random forest trained heuristic for Open Source LLVM Compiler version 3.3. We implement all the custom passes to unroll and instrument loops and to collect features in LLVM. In addition, we use SPEC 2006 integer and floating point C++ applications to train and test our heuristics. Due to the limited resources available to us, we train our heuristics on the set of applications having lower number of loops. We train our heuristics on a training set including 470.lbm, 462.libquantum, 458.sjeng, 456.hmmer, 429.mcf, 401.bzip2, and 445.gobmk applications. Our testing set involves rest of the SPEC 2006 applications. We train and test our heuristics on Intel Xeon 64-bit 2.2GHz 12-core server with 32KB L1 cache and 256KB L2 cache.

4. CONCLUSION

5. REFERENCES