Partial Redundancy Elimination using Lazy Code Motion

Sandeep Dasgupta[1] Tanmay Gangwani[2]

April 25, 2014

[1]Electronic address: sdasgup3@illinois.edu
[2]Electronic address: gangwan2@illinois.edu

Summary

Having identified the computationally and lifetime optimal placement points previously, we spent the bulk of the time in this phase in the following items:

1. Adding code to insert and replace expressions
2. Handling loop invariant code motion
3. Building up the testing infrastructure and initial testing.

Below we detail each of these steps.

Insert and Replace

To maintain compatibility with SSA, we perform insertion and replacement through memory and re-run the *mem2reg* pass after our PRE pass to convert the newly created load and store instructions to register operations. Following are the major points:

- Assign stack space *(allocas)* at the beginning of the function for all the expressions that need movement
- At insertion point, compute the expression and save the value to the stack slot assigned to the expression
- At replacement point, load from the correct stack slot, replace all uses of the original expression with the load instruction, and delete the original expression
- *mem2reg* converts stack operations to register operations and introduces the necessary Φ instructions

Working on same example (with some added complexity) as that in our phase-1 report, we show in Figure1 and Figure2, the optimizations performed by our PRE pass. The intention here is to show how our version of PRE performed on the computations $a + b$ & $a < b$;

Loop Invariant Code Motion

We first address a question raised in the phase-1 report. We had previously mentioned that to optimize for space and time, we provide a bit vector slot only to value numbers which have more than one expression linked to them. With this, however, we could miss opportunities for loop invariant code motion. As a solution, we extend the bit vector to include value numbers which have only a single expression linked to them but only if the expression is inside a loop. Note that we still exclude the cases where the expression is not part of a loop, and expect reasonable space and time savings.

The second issue we resolved with respect to loop invariant code motion was checking for zero-trip loops. Our PRE algorithm would move the loop invariant computations to the loop pre-header only if placement in the loop pre-header is anticipatible. Such a pre-header is always available for *do-while* loops, but not for *while* and *for* loops. Hence, a modification is required to the structure of *while* and *for* loops which peels off the first iteration of the loop, protected by the loop condition. This alteration provides PRE with a suitable loop pre-header to hoist loop independent computations to. In Figure 3 we show the CFG changes. Fortunately, we were able to achieve this effect using an existing LLVM pass *-loop-rotate* rather than having to write it ourselves.

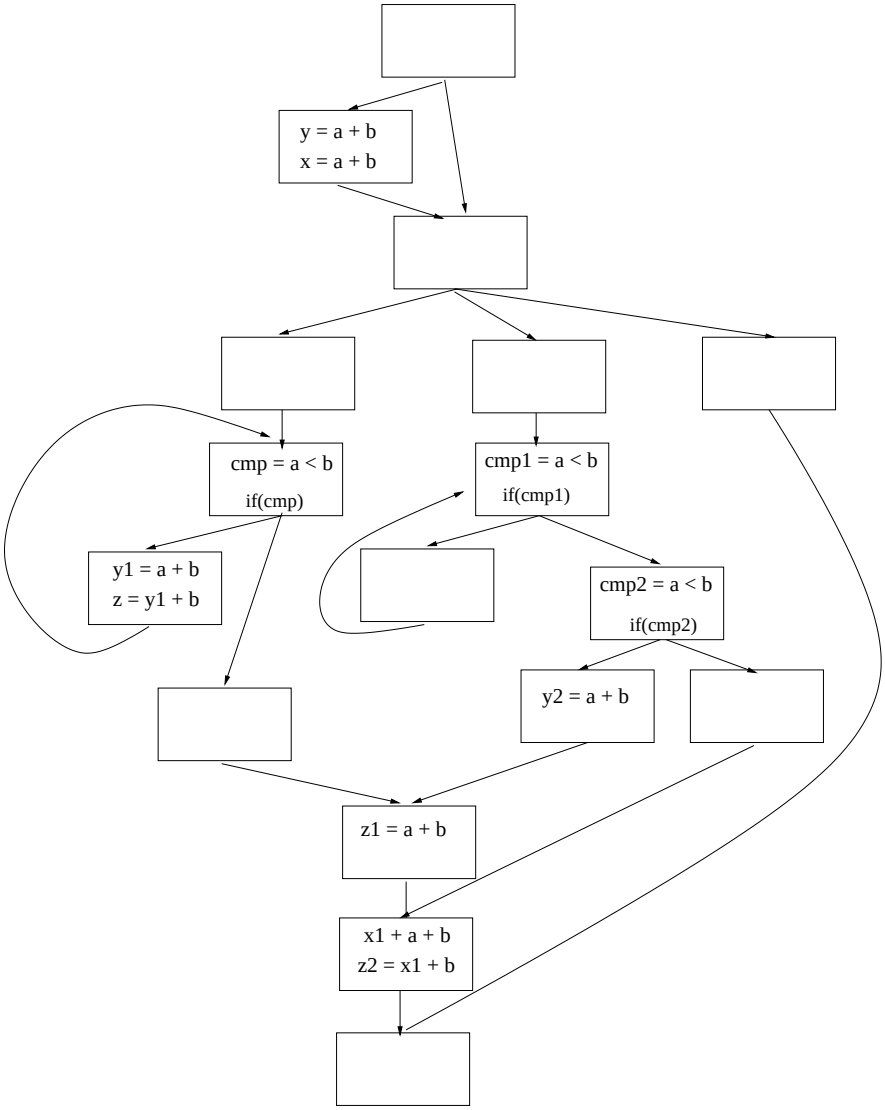

Figure 1: A motivating example

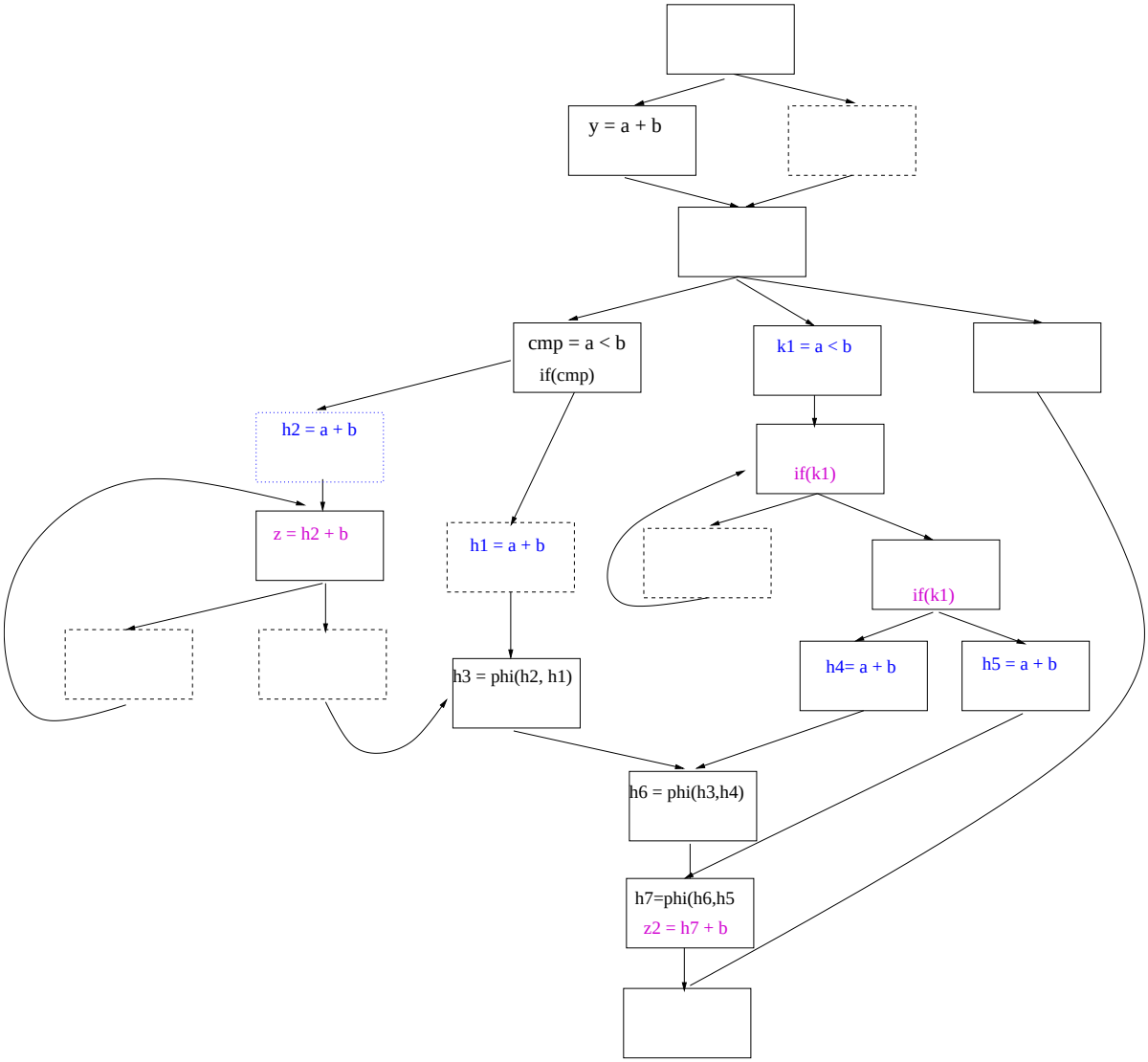

Figure 2: Lazy code motion transformation on computations $a + b$ & $a < b$. Dotted boxed denote critical edges. Blue dotted boxes are the ones inserted by loop rotate. PRE can insert the computations in these places. Inserted statements are marked blue and replaced ones with magenta

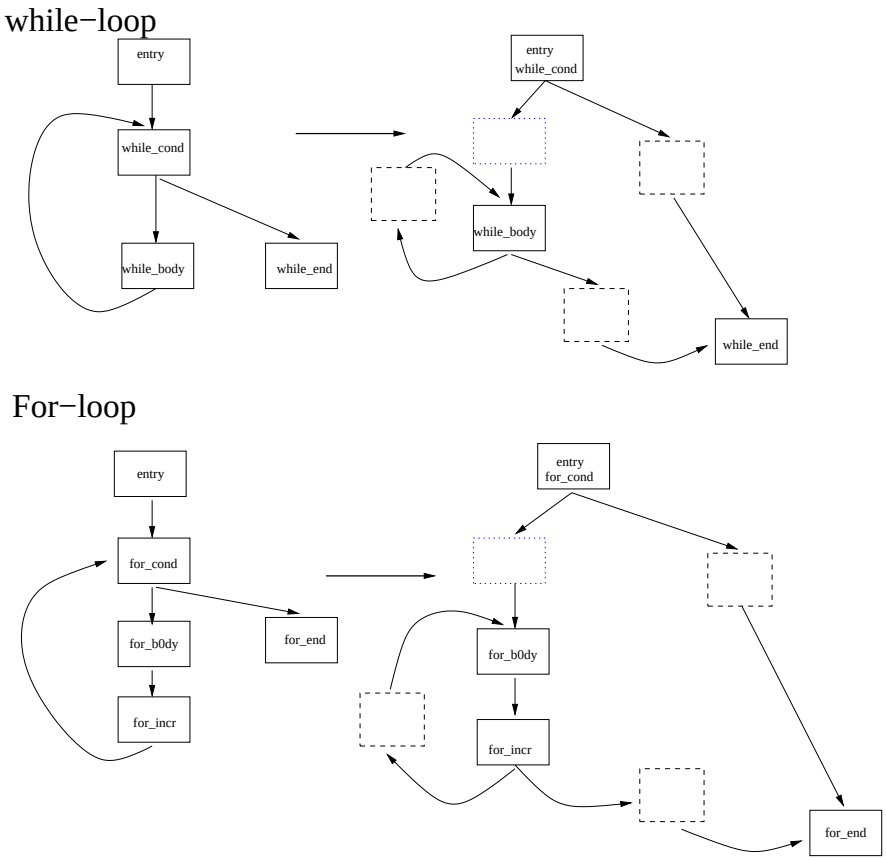

Figure 3: Loop transformations done by *-loop-rotate*. *do-while* loops remain unaffected. Blue dotted boxes are the ones inserted by loop rotate. PRE can insert the computations in these places.

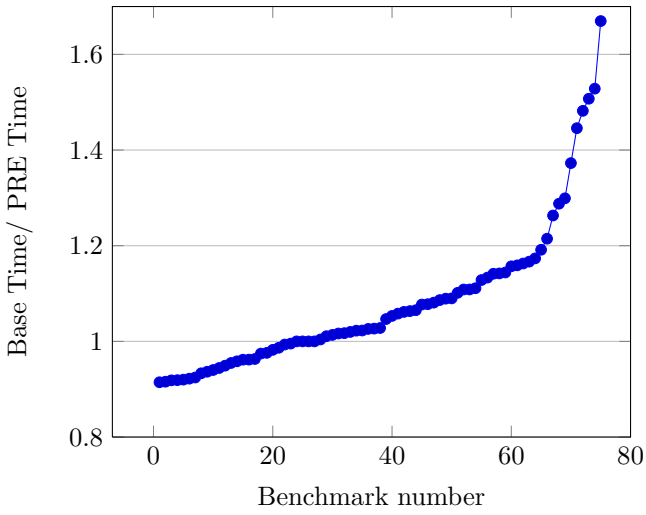

Figure 4: S-curve for performance improvements over Baseline (no PRE)

Testing

Apart from the small test cases which we created in phase-1, we have been able to successfully test our PRE pass on real codes from the LLVM test-suite. Specifically, we have completed the testing for 75 benchmarks from LLVM single-source package *"test-suite/SingleSource/Benchmarks/"* for correctness and performance. Correctness is checked by comparing the output of the binary optimized with our PRE pass with the reference output provided along with benchmarks. All benchmarks pass the correctness test. For performance, we compile the codes with and without the PRE pass and take the ratio of the run-times on the hardware. Figure 4 shows the S-curve for performance. For 55/75 benchmarks, our pass improves the run time by varying amounts (upto 67%). Performance drops for few benchmarks, but the degradation is bound by 8%.

Apart from measured runtime on the hardware, another metric to quantify the effects of our optimization pass is the dynamic instruction count. We use Pin (dynamic binary instrumentation tool) from Intel for this purpose. We have written a very simple Pintool to dump the dynamic instruction count of each type of instruction. We would present supporting data from Pin our final report.

Conclustion and Future Work

In this phase, we were able to code the missing pieces of our PRE algorithm, thereby achieving full functionality of the project. The highlights were the insert-replace algorithm, LICM improvements, and fixes for numerous bugs which surfaced during testing on the LLVM single-source package. We have written scripts to automate the testing, and this would speed up work for the final phase.

The S-curve in this report (Figure 4) presents improvements with respect to the baseline (no PRE pass) for single-source package. We would like to evaluate the cases where our pass degrades performance compared to baseline. In our final report, we plan to include similar curves for benchamarks from the LLVM multi-source package as well as the SPEC 2006 suite. Also, we would include S-curves to compare the effectiveness of our PRE pass with the LLVM GVN-PRE pass. For extreme outliers, we hope to present supporting data to reason about the performance change. Pin tool analysis and the statistics dumped by our PRE pass would be used for this.