Path Splitting: a Technique for Improving Data Flow Analysis

Massimiliano Poletto *
maxp@lcs.mit.edu
MIT Laboratory for Computer Science

Introduction

Path splitting [8] is a new technique for improving the amount of data flow information statically available to the compiler about a fragment of code. Path splitting replicates code in order to provide optimal reaching definitions information within given regions of the control flow graph. This improved information is used to extend the applicability of various "classical" code optimizations, such as copy propagation, common subexpression elimination, dead code elimination, and code hoisting [1, 2].

Path splitting was implemented in the SUIF compiler [4]. Experimental results indicate that path splitting effectively restructures loops, modifying the control flow graph so as to improve data flow information and hence enable further classical optimizations. Path splitting can decrease the cycle count of loops by over a factor of two. Such transformations can result in over 7% run time performance improvements for large C benchmarks. Although path splitting can cause exponential growth in code size, applying it only to regions where it is beneficial limits code growth to below 5% for realistic C programs. Simulations reveal that in most cases this code growth has limited effects on cache performance.

Algorithm

Since loops are the regions of a program executed most often, they are the places where it is usually most profitable to invest compilation time and easiest to justify code size growth. Consequently, path splitting is performed only within loops. The path splitting algorithm can be outlined as follows:

1. Reaching definitions analysis.
2. Removal of joins within the body of a loop so as to create a "tree-like" flow graph structure within the body.
3. "Path specialization," a process by which the path from the head of the loop to each of the leaves of the body is replicated and appended to itself, in place of the original loop back-edge.
4. Removal of unnecessary code.

In order to minimize code growth, code is replicated to remove joins only when doing so will improve reaching definitions information at some point below that join. In other words, a join will be removed in step (2) above only if it is corrupting reaching definitions for some variable below the join in the loop body.

```
for (j=0;j<100;j++) {
  if (j%2)
    a = 3;
  else
    a = 4;
  c += a;
}
```

before

```
for (j=0;j<100;j++) {
  if (j%2)
    c += 3;
  else
    c += 4;
}
```

after

Figure 1. Simple example of path splitting.

This process is best understood by example. Consider the code in Figure 1 (**before**). Traditionally, no copy propagation or other data flow optimization depending on the value of `a` could be performed after the `if` statement, because multiple definitions of `a` reach the assignment to `c`. Path splitting restructures the code in order to make reaching definitions information within the loop as precise as possible. It therefore replicates the assignment to `c`, placing one copy of it after each assignment to `a`. Consequently, copy propagation can occur, so the sequence `a=n; c+=a;` becomes `c+=n`, and we save one assignment on each iteration through the loop. This results in the code in Figure 1 (**after**). By replicating loop tests and eliminating jumps using techniques similar to those presented by Mueller and Whalley [6, 7], we are able to further decrease the number of instructions in the bodies of critical loops. Similarly, path splitting can enable further common subexpression elimination, dead code elimination, and other data flow optimizations in which having precise reaching definitions information is beneficial.

---

*Massimiliano Poletto is advised by Frans Kaashoek. This work was supported in part by the Advanced Research Projects Agency under contracts N00014-94-1-0985 and by a NSF National Young Investigator Award to Professor Kaashoek. Address: Room 521a, 545 Technology Square, Cambridge MA 02139. Web: http://www.pdos.lcs.mit.edu/~maxp

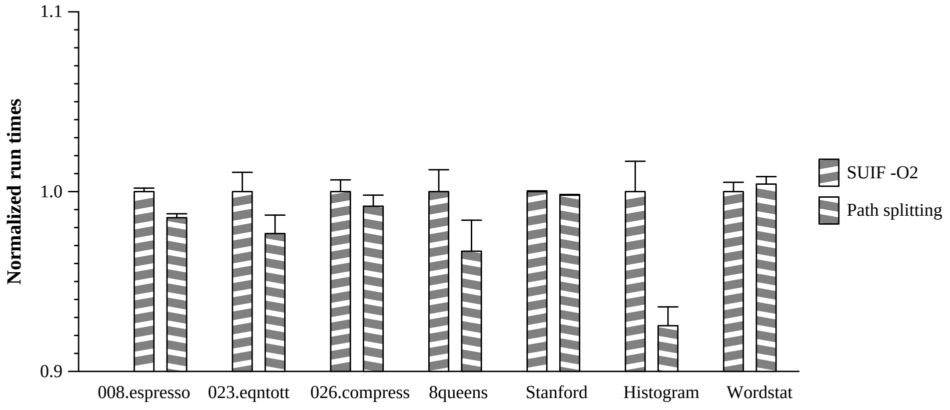

Figure 2. Normalized run times of some large benchmarks.

Only steps (1) and (2) outlined above are necessary to produce the simple example of Figure 1. However, by performing phases (3) and (4) we create a number of specialized loops, potentially as many as one for each distinct path through the body of the loop. Each of these "specialized" loops is tuned to a particular set of data flow values. In addition, each such loop is free of joins. We expect that by increasing code size and decreasing the number of joins in fragments of code, path splitting will improve the performance of trace scheduling for extracting instruction-level parallelism on super-scalar machines.

Results

Figure 2 illustrates normalized run times for a variety of benchmarks on a DECstation 5000/133. Error bars on the graph indicate 95% confidence intervals for the corresponding data. Although some benchmarks did not greatly benefit from it, path splitting significantly improved program performance in several cases (for example `008.espresso`, `histogram`, `8queens`). Only in one case (`wordstat`) did we measure a decrease in performance, but the decrease was very slight, and the confidence intervals overlapped to a large degree. Run time measurements were performed both in multi-user mode, with the machine connected to the network, and in single-user mode, disconnected from the network. In both cases, path splitting improved the performance of most code in a statistically significant manner.

Although removing joins by code replication can exponentially increase the size of a given loop, measurements show that, for large benchmarks, code growth caused by path splitting is generally less than 5% of the original code size. Cache simulations performed with the `dineroIII` cache simulator [3, 5] also indicate that cache effects due to this increased code size are usually minimal, and that the code rearrangement due to path splitting can actually increase performance in the case of direct-mapped instruction caches.

We are currently still evaluating path splitting as a technique for improving the effectiveness of trace scheduling on super-scalar machines.

Conclusion

Path splitting is a method for transforming the control flow graph of a program in order to statically expose additional data flow information to the compiler. Path splitting should be considered a speculative optimization: it has never caused significant decreases in performance, but it may not always provide large performance benefits. Nevertheless, it can be quite useful in several cases: by increasing the effectiveness of classical code optimizations, it can decrease the run time of large C benchmarks by up to 7%, while limiting code size growth to below 5%.

Acknowledgments

Thanks to Dawson Engler, who first had the idea of replicating code and "splitting" joins to improve optimizations, and to Wilson Hsieh, for helpful comments and suggestions.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. Technical report, Computer Science Division, U.C. Berkeley, Berkeley, CA, 1992.

[3] T. Ball and J. R. Larus. Optimally profiling and tracing programs. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, pages 59–70, Albuquerque, NM, January 1992.

[4] Stanford SUIF Compiler Group. SUIF: A parallelizing & optimizing research compiler. CSL-TR-94-620, Computer Systems Laboratory, Stanford University, May 1994.

[5] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software Practice & Experience*, 24(2):197–218, February 1994.

[6] F. Mueller and D. B. Whalley. Avoiding unconditional jumps by code replication. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 322–330, San Francisco, CA, June 1992.

[7] F. Mueller and D. B. Whalley. Avoiding conditional branches in loops by code replication. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995. (To appear).

[8] Massimiliano Poletto. Path splitting: a technique for improving data flow analysis. Master's thesis, Massachusetts Institute of Technology, 1995.