Effect Profiling:
15-745 Project Proposal

Katrina Ligett and Noam Zeilberger

April 5, 2006

Our email addresses are `katrina@cs` and `noam@cs`. The web page for this project is `http://www.cs.cmu.edu/~noam/745`.

1 Goal

Many compiler optimizations are prevented by the possibility of side-effects in code. For example, dead-code elimination may only be performed if it is known that the eliminated code would not affect the rest of the computation, while partial redundancy elimination is only sound if the reused value could not have depended on intervening instructions. To deal with this problem, many compilers take a very conservative approach—for instance, all function calls are assumed to be effectful, and therefore will not be eliminated as dead-code nor their results reused. Other compilers may attempt a static effect analysis [1] [3]. However, static effect analysis is complicated for several reasons. For one, it depends on solving other hard problems, such as pointer aliasing and program termination. Additionally, it is often difficult to distinguish between real effects and "benign" effects (such as caching) which do not modify the input/output behavior of the program (though may affect performance).

Our project begins with the observation that a simple alternative to static effect analysis is dynamic *effect profiling*. The idea of effect profiling is to keep track of all effects (e.g. system calls, memory accesses, etc.) encountered during the lifetime of a program. The effectfulness of a function on a given input is then inferred by simply observing the segment of the trace between a call and its return; there is no need for complicated interprocedural analysis—the effects are in the trace. In particular, note the significance of negative information: if no effects are raised during a function call on some set of inputs, then that function will *always* be effect-free on those inputs—by definition its behavior on that domain is deterministic. Hence this information can be used to enable sound code optimizations. This characteristic of effect profiling places it in contrast with other profiling techniques; typically those make no guarantees (not even statistical ones) that the program's behavior in general will resemble its profiled behavior. The goal of our project is then to to design and implement an effect profiling system, and measure its effectiveness.

2 Background

There is an extensive literature on static effect analysis. Most approaches are based on flow analysis [1] [3], though there are also approaches based on effect type-systems [5].

Program profiling has a long history, from simple profiling of time spent inside of functions [2] to recording entire execution paths through a program [4]. The profiling technique perhaps most related to ours is data breakpointing [9], which keeps track of whenever a variable is modified. As far as we know, this technique has only been used for debugging purposes and not for optimization.

Finally, in order to increase the usefulness of effect profiles we must use the behavior of a function on a few inputs to infer its behavior on more general inputs (see section "Input dependency"). This is closely related to the problem of "test adequacy" [7] [6].

3 Approach

Our project will consist of the five stages described below.

3.1 Initial design and implementation

In the first stage we will implement very coarse-grained effect profiling as an instrumentation pass in our L3 compiler. The instrumented code will produce traces of program executions that mark function calls and returns, as well as anywhere an effectful instruction is executed—in L3 these include accessing a global variable or pointer, and making an external call. From these traces we can determine for each function a subset of inputs on which it is guaranteed to execute effect-free.

3.2 Refined analysis of effects

Not all effects should be treated equally. For example, we can distinguish between read effects and write effects. If the only effect incurred by a function on some input is a read of a global variable, then the result may still *depend* on global state and hence cannot be reused via common subexpression elimination—however, it will not *modify* global state and thus if the result is dead the function call could be safely removed.

It is therefore potentially very profitable to distinguish between different kinds of effects in our effect profiles. At the least we will distinguish between read effects and write effects. We may also try to distinguish between accesses of different parts of global state. This is easy for global variables but harder for pointers—we may attempt an approach based on regions [8]. The representation format of refined effect traces will be a question explored during the design phase.

3.3 Input dependency

We will also explore methods for inferring that a larger set of function inputs is effect-free, despite only performing profiling on a small set of inputs. For example, it may be possible to determine that a function $f(x, y)$ depends only on the value of x, and not on the value of y. In this case, knowing that f is effect-free on some particular input (x_0, y_0) may allow us to assert that f is also effect free on *all* inputs (x_0, y_i), for any y_i. If successful, these more advanced analyses could dramatically improve the number of optimizations enabled by our technique.

3.4 Enabling optimizations

In the fourth stage of our project, we will consider a wide variety of classical compiler optimizations to determine which optimizations are enabled by effect profiling. We will consider the impact of our more refined effects analyses on these optimizations. Time permitting, we may also implement some of the optimizations enabled by effect profiling.

3.5 Analysis, evaluation, and future work

We will evaluate our project by analyzing the performance hit incurred by effect profiling, the frequency with which optimizations are enabled on hot paths, and, if data are available, the performance improvement in code generated using optimizations enabled by effect profiling. We will also provide a qualitative discussion of the benefits and challenges of our technique. It may also be interesting to consider modifications of our technique to enable effect profiling for higher-order functions; however, detailed exploration of this issue is beyond the scope of our project.

4 Plan

Our approach to this problem is staged, allowing us to adjust our plan as necessary. In particular, we have flexibility in the amount of time we spend on refinements of our technique and on implementing enabled optimizations.

We do not anticipate needing any special resources for this project.

References

[1] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 29–41, New York, NY, USA, 1979. ACM Press.

[2] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN*

symposium on Compiler construction, pages 120–126, New York, NY, USA, 1982. ACM Press.

[3] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 56–67, New York, NY, USA, 1993. ACM Press.

[4] James R. Larus. Whole program paths. In *Proceedings of the SIGPLAN '99 Conference on Programming Languages Design and Implementation*, Atlanta, GA, May 1999.

[5] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57, New York, NY, USA, 1988. ACM Press.

[6] Thomas J. Ostrand and Elaine J. Weyuker. Data flow-based test adequacy analysis for languages with pointers. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 74–86, New York, NY, USA, 1991. ACM Press.

[7] Sandra Rapps and Elaine J. Weyuker. Data flow analysis techniques for test data selection. In *ICSE '82: Proceedings of the 6th international conference on Software engineering*, pages 272–278, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[8] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.

[9] Robert Wahbe, Steven Lucco, and Susan L. Graham. Practical data breakpoints: design and implementation. *SIGPLAN Not.*, 28(6):1–12, 1993.