How to be an effective profiler: A 15-745 Project

Katrina Ligett and Noam Zeilberger*

May 15, 2006

1 Introduction

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 has no side-effects and does not loop, 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 [3] [6]. 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 began with the observation that if no effects are raised during a function call on a particular set of inputs, then that function will *always* be effect-free on those inputs—by definition its behavior on that domain is deterministic. More generally as an alternative to complete static effect analysis, one can compute effect profiles along particular program paths. This greatly simplifies the analysis, for example by side-stepping the issue of termination. In this paper we describe a technique that combines whole program paths with abstract interpretation to derive refined profiles of the effectful behavior of a program. These profiles take the form of effects produced by function calls on a specific execution, given general conditions met at the call sites.

2 Related Work

There is an extensive literature on static effect analysis. Most approaches are based on flow analysis [3] [6], though there are also approaches based on effect type-systems [8].

*(katrina|noam)@cs. The web page for this project is http://www.cs.cmu.edu/ noam/745/

pure expression

$$
\begin{array}{lll}
e & ::= x \mid n \mid e_1 \oplus e_2 & \\
\text{propositions} & & \\
P & ::= e_1 = e_2 \mid e_1 < e_2 \mid \ldots & \\
\text{command} & & \\
c & ::= x = e & \text{assignment} \\
 & \mid \quad x = [e] & \text{load} \\
 & \mid \quad [e] = e' & \text{store} \\
 & \mid \quad assert(P) & \text{condition} \\
 & \mid \quad call(f(e_1, \ldots, e_n), \tau) & \text{function call} \\
\text{trace} & & \\
\tau & ::= \cdot \mid c \cdot \tau &
\end{array}
$$

Figure 1: Program traces

Program profiling has a long history, from simple profiling of time spent inside of functions [5] to recording entire execution paths through a program [7]. The profiling technique perhaps most related to ours is data breakpointing [12], 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.

3 Effect analysis

3.1 Program traces

The compiler instruments programs to generate traces, whose syntax is given in Figure 1. Every primitive command is logged, along with functional calls/returns and branches taken at conditional jumps. An efficient technique for doing this is described in [7], though our prototype compiler uses a simpler, less efficient implementation.

3.2 Interpretation of traces

Our general approach to effect analysis is an abstract interpretation of the program traces generated by the instrumented code. This allows us to produce a refined analysis of the effects (memory updates and system calls) produced by each function encountered in the trace, in terms of conditions on the function inputs that were met during program execution. This analysis is sound in the sense that whenever a function is called with inputs satisfying the conditions in a profile, it will only have the effects specified in that profile. For example, on a given program execution, all calls to a certain function f may be pure (i.e. have no effects); the compiler can then optimize calls to f in the original source code using the information that f is pure, if it can determine that these calls satisfy the same conditions as the program trace.

abstract value

v	$::= n$	constant
	$\mid \pi_i$	function parameter
	$\mid v_1 \oplus v_2$	operation
	$\mid \mathrm{ld}(\pi_H, v)$	memory load
	$\mid P \,?\, v_1 : v_2$	conditional value
	$\mid \top$	unknown

abstract proposition

$P \ ::= v_1 = v_2 \mid v_1 < v_2 \mid \ldots$

abstract heap

H	$::= \pi_H$	heap parameter
	$\mid [v_1 \mapsto v_2]H$	assignment
	$\mid \top$	unknown heap

Figure 2: Interpretation lattice

We now give a formal description of a simplified version of the abstract interpretation used in our effect analysis. In this simplified account, we ignore external system calls, function return results, and the intricacies of L3's fat pointers—we give some discussion of those issues in later sections. For our purposes in this section, abstract interpretation produces for each function call just the heap update performed on that trace, in terms of the function parameters and input heap.

The interpretation lattice is given in Figure 2. It is mostly straightforward, but the analysis of pointers deserves some discussion. The state of the heap at the beginning of a function call is represented by the abstract parameter π_H, and a memory load of location v from that heap is denoted by $\mathrm{ld}(\pi_H, v)$. Assignments are dealt with by treating the heap as a function—a load of address v in the heap $[v_1 \mapsto v_2]H$, evaluates to v_2 if $v = v_1$, or else to the value determined by the load in H.

Evaluation of a trace τ on an abstract heap H produces a new heap H' assuming that conditions ϕ are met. This is represented by the judgment $\tau, H \Downarrow \phi \supset H'$, which is given in Figure 3. Note that on a function call, we generalize the function parameters and input heap to be $\pi_1, \ldots, \pi_n$ and π_H respectively, so as to obtain the most general profile information for the callee. Given this general profile, we perform a substitution to obtain the effects in terms of the caller's parameters and heap.

3.3 Simplifying effect profiles via theorem proving

The conditions generated along an abstract execution in Figure 3 are often redundant in the sense that they are tautologous given the preceding conditions. For example, suppose that executing the first iteration of a loop generates the condition $2 * \pi_1 < \pi_2$, and the next iteration generates $\pi_1 < \pi_2$—the second condition is strictly redundant. Similarly, symbolic heaps may be overly complicated, if they contain conditional values $P \,?\, v_1 : v_2$ where P is always valid given the preconditions.

$$\cdot, H \Downarrow \top \supset H$$

$$\frac{[v/x]\tau, H \Downarrow \phi \supset H'}{x = v \cdot \tau, H \Downarrow \phi \supset H'} \qquad \frac{\tau, H \Downarrow \phi \supset H'}{assert(P) \cdot \tau, H \Downarrow (P \wedge \phi) \supset H'}$$

$$\frac{[v'/x]\tau, H \Downarrow \phi \supset H' \quad v' = load\ v\ H}{x = [v] \cdot \tau, H \Downarrow \phi \supset H'} \qquad \frac{\tau, [v_1 \mapsto v_2]H \Downarrow \phi \supset H'}{[v_1] = v_2 \cdot \tau, H \Downarrow \phi \supset H'}$$

$$\frac{\begin{array}{c}\sigma = [v_1/\pi_1] \ldots [v_n/\pi_n][H/\pi_H] \\ \tau_f, \pi_H \Downarrow \phi_f \supset H_f \quad \tau, \sigma(H_f) \Downarrow \phi \supset H'\end{array}}{call(f(v_1, \ldots, v_n), \tau_f)\tau, H \Downarrow \sigma(\phi_f) \wedge \phi \supset H'}$$

$$\begin{array}{rcl} load\ v\ \pi_H & = & \mathrm{ld}(\pi_H, v) \\ load\ v\ ([v_1 \mapsto v_2]H) & = & (v = v_1)\ ?\ v_2 : load\ v\ H \end{array}$$

Figure 3: Abstract execution

Now, via a relatively straightforward translation we can convert the symbolic propositions generated by abstract execution into propositions of first-order arithmetic. (The only trickiness involves converting equations containing conditional values.) We can then send the resulting first-order propositions to the Coq proof assistant's `Omega` tactic to determine whether they are always valid [9] [1]. In our experiments this approach resulted in significantly simplified effect profiles.

3.4 Fat pointers

In order to provide safe memory operations, L3 uses a special representation for pointers called "fat pointers." A fat pointer consists of a three-word header containing the base address of a memory block, its size, and an indexing offset. The effective address of a fat pointer is computed as $base + 4 * offset$; the L3 compiler produces code that checks if the effective address is within bounds of the memory block before performing a load/store, and produces a run-time error if it is out of bounds. A naive implementation of the preceding analysis without taking fat pointers into account would result in many constraints that could not be removed, though they are semantically redundant. In particular, one can make use of the fact that after the fields of a fat pointer are initialized, they cannot be overwritten. We therefore extend the grammar of symbolic values with values of the form $\mathrm{base}(v)$, $\mathrm{sz}(v)$, and $\mathrm{ofs}(v)$, denoting loads of these three fields. These load are not affected by memory updates (though we must take some care to deal with initialization).

3.5 More details

Besides memory updates, L3 programs can also be effectful by invoking system calls. Our effect profiler simply records all system calls made on an execution. Arbitrary system calls can have an arbitrary effect on the heap, so we can only represent the returned heap as $\top$. However for particular system calls (e.g. `_alloc`), we can encode a more refined behavior. We can also refine our effect profiles by giving effect/condition pairs for prefixes along an execution path, rather than only for entire function calls. Finally our effect analysis takes into account the fact that L3 functions can return results (the results of external functional calls are $\top$ by default).

4 Future work

Several directions for future research are immediate. First we would like to develop a more practical implementation of the instrumentation phase using techniques from [7]. This is important because it would allow effect profiling to be studied on real-world applications. There is also an interesting question of whether effect profiles can be used as debugging tools by programmers, though in order to realize this we would have to convert effect profiles to a more human-readable format. On the other hand, we would also like to explore the use of effect profiles for enabling compiler optimizations (as other path profiling techniques have been used [2]). Many optimizations could be enabled with the knowledge that a function is pure on a particular program path—for example, partial redundancy elimination, dead-code removal, aggressive inlining—but the practical impact of this is unclear.

The actual effect analysis could be refined, particularly for pointer programs. Since our analysis treats the heap globally whereas real programs tend to manipulate only portions of the heap, it could benefit from ideas from separation logic [10] or region analysis [11]. The approach to separation reasoning in [4] might be immediately useful since it is also based on a functional representation of the heap. A more speculative question is whether our analysis could be extended to higher-order programs, where functions could themselves return effectful functions.

Finally, effect profiling is an interesting combination of a dynamic analysis technique (whole program path profiling) with a static one (abstract interpretation). Could this hybrid approach yield useful solutions to other problems?

References

[1] The Coq proof assistant. http://coq.inria.fr/.

[2] Glenn Ammons and James R. Larus. Improving data-flow analysis with path profiles. *SIGPLAN Not.*, 39(4):568–582, 2004.

[3] 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.

[4] Richard Bornat. Proving pointer programs in hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[5] 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.

[6] 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.

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

[8] 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.

[9] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13, New York, NY, USA, 1991. ACM Press.

[10] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

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

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