Reflections on the Role of Static Analysis in Cooperative Bug Isolation*

Ben Liblit

Computer Sciences Department
University of Wisconsin–Madison
liblit@cs.wisc.edu

Abstract. Cooperative Bug Isolation (CBI) is a feedback-directed approach to improving software quality. Developers provide instrumented applications to the general public, and then use statistical methods to mine returned data for information about the root causes of failure. Thus, users and developers form a feedback loop of continuous software improvement. Given CBI's focus on statistical methods and dynamic data collection, it is not clear how static program analysis can most profitably be employed. We discuss current uses of static analysis during CBI instrumentation and failure modeling. We propose novel ways in which static analysis could be applied at various points along the CBI feedback loop, from fairly concrete low-level optimization opportunities to hybrid failure-modeling approaches that may cut across current static/dynamic/statistical boundaries.

1 Introduction

A complete Cooperative Bug Isolation (CBI) system constitutes a feedback loop between developers and users. Developers provide software to users, and users respond with data about that software's behavior in the deployed environment. Developers then use this data to improve the software in future releases, guided largely by sophisticated statistical models of program misbehavior. The goal is not to produce perfect first releases, but rather to improve software continuously over time guided by the needs of real user communities [1]. It is non-obvious how formal static analysis should interact with CBI's dynamic/statistical approach to software quality. This paper explores how static analysis is currently used within CBI, and how it could most beneficially be used in the future.

1.1 Overview of Cooperative Bug Isolation

We start with a conceptual overview of CBI's feedback loop in Fig. 1. The process begins at the top left, with program source and a small set of configuration choices (made by the developer) that will steer a CBI instrumenting compiler. Configuration choices

* Supported in part by AFOSR Grant FA9550-07-1-0210 and NSF Grants CCF-0621487, CCF-0701957, and CNS-0720565. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of AFOSR, NSF, or other institutions.

M. Alpuente and G. Vidal (Eds.): SAS 2008, LNCS 5079, pp. 18–31, 2008.

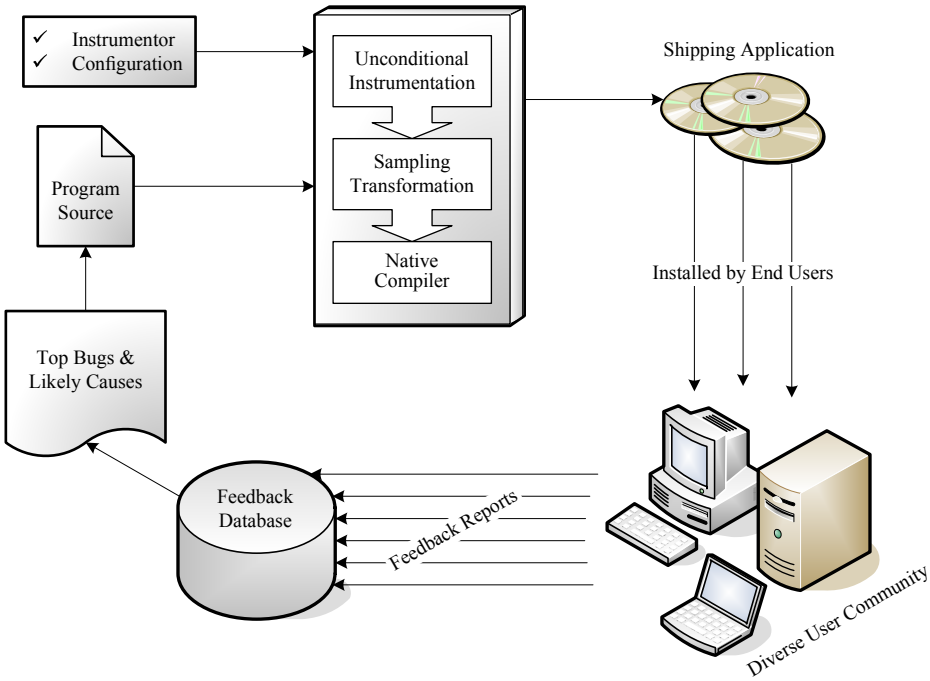


Fig. 1. Conceptual overview of Cooperative Bug Isolation system

are simple and few in number, while program source is absolutely free of any manual, CBI-specific annotation. This minimizes the up-front cost to developers wishing to adopt CBI.

The boxed tool chain at the top center of Fig. 1 represents a CBI instrumenting compiler. This appears to the developer to be a standard compiler augmented with a few additional high-level configuration options, and therefore is easily incorporated into existing build environments. Internally, CBI-instrumented compilation consists of three distinct steps: (1) insertion of unconditional instrumentation; (2) transformation of instrumentation to be sampled instead of unconditional; and (3) native compilation to binaries ready for distribution. The third step is treated as a black box: we accept whatever the native compiler gives us. But each of the first two steps represents a key point at which static analysis could be used to good effect. We review unconditional instrumentation in Sect. 2, with a discussion of the ways in which static analysis is currently used and could potentially be used in the future. Section 3 recaps the sampling transformation, and again considers the current and potential roles for static analysis.

Instrumented compilation yields executable binaries (both applications and shared libraries) capable of monitoring and reporting aspects of their own behavior at run time. We make this instrumented software available to the general public: the "diverse user community" in the lower right of Fig. 1. Each run of an instrumented program produces one feedback report with information about run time behaviors instrumented during compilation. Each feedback report also includes a binary label marking that run as

good (successful) or bad (failed). The challenges at this stage largely fall under the broad umbrella of "computer systems": transmitting feedback data over the network, warehousing it in databases, keeping user data secure throughout its lifetime, and so on. Static program analysis plays no role here.

As runs accumulate, certain trends emerge. If we put all successful runs in one pile and all failed runs in another, we can look for instrumented behaviors that vary between the two piles. In particular, we can direct developers' attention toward suspicious run time behaviors that are strongly associated with failure. Absolute assignment of blame ("the program crashes if and only if `x` is negative on line 140") is impossible due to the many sources of uncertainty in our feedback data. Instead, we have developed a variety of *statistical debugging* techniques to identify informative trends in the difference between successful and failed runs. Statistical debugging is another key point in the feedback loop where static program analysis can play a major role, although many open questions remain. In Sect. 4 we review selected statistical debugging models developed both by my collaborators and by others. We consider the relatively limited use of static program analysis in statistical debugging work thus far, and suggest future directions in which program analysis could play a more prominent role.

Certain recurring challenges span all of the contexts in which we consider adding static analysis to CBI. Section 5 vents some steam about problems that have put many attractive static analyses out of CBI's reach. For the stout-hearted reader who is not so offended as to discard the paper following this rant, Sect. 6 concludes.

2 Unconditional Instrumentation

The first step in compiling an application for use with CBI is to inject extra monitoring code at selected program points of interest. We call this code *unconditional instrumentation* to contrast it with the sampled instrumentation that results from later transformations (see Sect. 3). Unconditional instrumentation casts a broad net across the program, adding code to record a wide variety of run time behaviors that could potentially be of interest later when tracking down a bug. This is not the place to be stingy: we regularly add many hundreds of thousands of monitoring points to medium-sized programs.

Obviously we do not expect the programmer to insert this code by hand. Rather, the developer selects among a small collection of broad *instrumentation schemes*. Each scheme matches specific fragments of program syntax at specific locations in code. We call such matches *instrumentation sites*. For example, the *branches scheme* places one instrumentation site at each `if` statement, while the *returns scheme* creates one site at each function call. Possible behaviors at each site are strictly partitioned into a small set of *predicates* on program state. For example, a branch site tests two predicates: the `if` condition is either true or false. We create one global *predicate counter* for each predicate, which records how often that predicate was true during a run. When execution reaches the location of an instrumentation site, a single *observation* is made: exactly one of the predicates at the site must be true, and the injected code makes this determination and increments the corresponding predicate counter. The feedback report for an entire run, then, consists primarily of the final counter values for all predicates.

`sampler-cc` is our CBI instrumenting compiler for C. It is not the only instrumentor that has been created [2,3,4], but it is the most mature and widely-used. `sampler-cc` currently offers seven instrumentation schemes; developers may activate as many or as few as desired [5]. Some schemes, such as the branches scheme, are quite broad-based and stand a decent chance of detecting something of interest for a wide variety of coding mistakes. Other schemes are more specialized, such as the *g-object-unref scheme* which focuses on reference count mismanagement bugs in a specific widely-used open source library. Specialized schemes are less likely to trigger for any given bug, but if they do correlate strongly with failure, the guidance they offer the developer is more specific and therefore more useful. We generally recommend a mixture of broad-spectrum and narrowly-focused schemes.

2.1 Static Analysis as Currently Used

`sampler-cc` performs almost no interesting static analysis during the unconditional instrumentation stage. This is a somewhat embarrassing admission. Unconditional instrumentation relies on the popular CIL C front end for parsing, type checking, and name resolution [6], and arguably that constitutes program analysis of a sort. But beyond these standard front-end tasks, initial instrumentation is essentially a matter of local pattern-matching against syntactic structures (e.g., finding every conditional branch) and insertion of appropriate monitoring code (e.g., to count how often the branch condition is true versus false). We rarely look at more than one small abstract syntax tree fragment at a time.

The one exception is a relatively recent feature that drops instrumentation sites that examine the values of uninitialized variables. The motivation is not that looking at such values is dangerous or forbidden. After all, this is C: everything is dangerous, and nothing is forbidden. Rather, we find that developers cannot easily make sense of instrumented predicates that involve uninitialized variables. Therefore, considering such predicates when hunting for bugs is not ultimately useful, even if some may be strong failure predictors. So `sampler-cc` uses an intraprocedural forward dataflow analysis to identify and avoid definitely-uninitialized variables.

2.2 Static Analysis Potential

Historically, we have claimed that deep analysis of buggy C programs is foolhardy. How many static analyses give truthful results for C programs that overrun buffers or read uninitialized memory? In a world of wild pointers and corrupted heaps, what points-to analysis can possibly be trusted? The execution semantics of such programs are very different from the formal semantics assumed by most analyses, which means that static analyses can no longer offer guarantees across all possible runs.

Furthermore, many of the facts that static analysis could offer are much easier to derive empirically by examination of dynamic feedback reports. Ernst's Daikon work shows that empirical invariant discovery can be quite effective [7]. Why try to *prove* that `z` must always be zero on line 490 when we can simply check whether it was ever *observed* to be nonzero across a hundred thousand user runs? Of course the later does

not provide any guarantees, but given the loose semantics of buggy C programs, it's not clear that any static analysis could offer guarantees either.

However, this may be prematurely dismissive. Static analysis could go a long way toward streamlining instrumentation by eliminating redundancy. A static proof that one predicate implies another means that the later need not be monitored at run time. For example:

```
1  const int result = fetch();
2  if (result) ...
```

The returns instrumentation scheme will check whether `result` is negative, zero, or positive on line 1. The branches scheme will check whether the conditional on line 2 is true or false. Is later redundant with respect to the former? Probably. If we assume that no other thread can change `result` between lines 1 and 2 (including by storing across a corrupted pointer), then instrumenting the branch is redundant. Should we make this assumption? Arguably, yes. It may cause us to miss certain bugs if we are wrong. But CBI never promised perfection. Reducing the instrumentation load by eliminating redundant or invariant predicates would have several benefits:

1. Less instrumentation means less code, for a smaller executable footprint on installation media, hard drives, and memory.
2. If sampling (discussed below) is not changed, then less instrumentation means more streamlined code and therefore better performance.
3. Conversely, if run time performance is held at a constant level, then less time wasted on uninteresting instrumentation sites allows more intensive sampling of other code that may be more informative.
4. Although statistical analysis of feedback reports can discover likely invariants and redundancies in observed data, this is not free. Advanced statistical debugging algorithms can have difficulty scaling up to massive datasets, and eliminating junk instrumentation earlier leaves the statistical methods with smaller problems to solve.

3 From Unconditional Instrumentation to Sampling

Here we detail CBI's instrumentation sampling transformation, which sacrifices feedback completeness for privacy and performance. We review static analyses currently used during the sampling transformation, and consider possible deeper analyses that could be employed in the future.

3.1 The CBI Sampling Transformation

Complete monitoring of all instrumentation predicates may be impractical or undesirable for reasons of performance or user privacy. We have developed a generic instrumentation sampling transformation to address these concerns [8]. CBI's sampling transformation is a statistically rigorous variant on a performance profiling transformation developed by Arnold and Ryder [9]. The sampling transformation reduces instrumentation overhead while maintaining a very strict statistical fairness guarantee: the behaviors

observed and tallied in site counters represent a sparse but statistically unbiased random sample with respect to the complete (but unobserved) dynamic behavior of the program. This fairness guarantee is necessary to ensure that the statistical debugging algorithms to be applied later have a sound mathematical footing.

At run time, each thread in an instrumented application maintains a countdown that represents the number of instrumentation opportunities that should be skipped before the next observation is taken. The countdown is set randomly using a geometric distribution, which is mathematically equivalent to counting how many times a tail-biased coin comes up tails before the next head is seen. A geometric distribution with mean 100 corresponds to counting tails while tossing a coin that has a 1/100 chance of coming up heads. This countdown yields a statistically fair random sample of about 1/100 of complete program behavior, as though the biased coin were being tossed at each instrumentation site.

The sampling transformation leverages this countdown by avoiding most instrumentation until an observation is imminent. The transformation begins by splitting each function's control flow graph into a collection of single-entry acyclic subgraphs. Doing this optimally is NP-hard [10], but placing acyclic subgraph entry points at loop back edges and the tops of functions works well in practice. An acyclic graph contains only a finite number of paths, each of which is of finite length. Therefore, there is a finite maximum number of instrumentation sites that could be crossed along any single execution through each acyclic subgraph. We call this maximum instrumentation site count the *threshold weight* of a given subgraph.

An instrumenting compiler now clones each acyclic subgraph. In the *fast clone*, we replace each instrumentation site with a simple decrement of the global next-sample countdown. In the *slow clone*, we decrement the countdown and check whether it has reached zero. If the countdown is zero, then we make a single observation at the current instrumentation site and reset the counter to a new geometrically-distributed random number. Entry into the fast and slow clones is guarded by a conditional branch that checks whether the global countdown is below the subgraph's threshold weight. If the countdown is larger than this threshold, then the decrements cannot possibly drive it to zero on this pass through the subgraph, and so the branch selects the fast clone. If the countdown is smaller than the threshold, then a sample might be imminent, and the instrumented slow clone is selected instead. If sampling is sparse (1/100 or 1/1000 is typical), then execution will usually proceed in the fast clone of each subgraph, switching to the slow clone only occasionally when a sampled observation is about to be made. In this way we improve performance by exploiting periods between samples as the fast, common case.

Figure 2 shows an example of an acyclic control flow subgraph after cloning and the insertion of the countdown threshold check. This subgraph has a threshold of four, because there is one path through the subgraph that crosses four instrumentation sites.

3.2 Static Analysis as Currently Used

Some static analyses are implied by the description of the sampling transformation given above. We require control flow graphs for each function, with loop back edges identified. The threshold weight of each single-entry acyclic subgraph is computed in a simple bottom-up pass. While this all constitutes analysis, it has nothing beyond the basics that any reasonable compiler would provide.

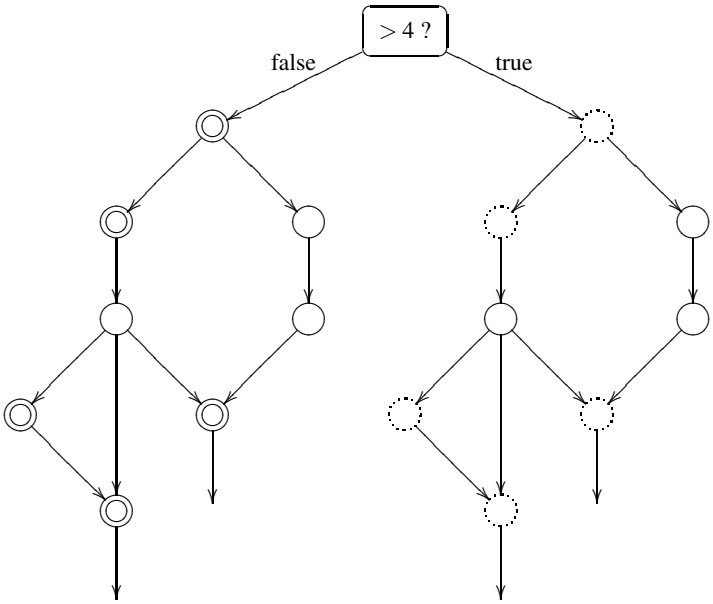


Fig. 2. Example of instrumented code layout. The slow clone is on the left; double-outlined nodes contain countdown decrements and instrumentation site code. The fast clone is on the right; dotted-outline nodes contain only countdown decrements. Single-outlined nodes contain no instrumentation sites.

Several other minor optimizations can be applied within each acyclic subgraph or instrumented function. For example, we add a static branch prediction hints to advise the native code generator that most threshold checks will choose the fast clone. Acyclic subgraphs containing zero or one instrumentation site require no cloning or threshold check at all. The global next-sample countdown is cached in a local variable while an instrumented function executes; this helps the native compiler coalesce decrements into fast register operations for a significant performance boost.

The only analysis that spans procedure boundaries is our identification of weightless functions. We define a *weightless function* as one that contains no instrumentation sites and that only calls other weightless functions. Weightless functions allow several optimizations in global countdown management, so we identify these using a fix-point computation over the call graph with conservative treatment of calls across pointers or that leave the current compilation unit. (A points-to analysis is offered as an option for resolving indirect calls. However, this is considered experimental and not recommended for production use due to its insufficiently-conservative treatment of separate compilation.)

3.3 Static Analysis Potential

Optimization of sampled instrumentation is primitive, with only very modest attempts to analyze beyond the boundary of a single function or a single acyclic subgraph. One

could certainly do better, such as by optimizing across procedure boundaries or by restructuring the fast clones for even greater speed. We propose two analysis-driven optimizations in detail as examples of the sort of improvements that could be made.

Bounded-Weight Function Analysis. With the exception of weightless functions, we currently treat each called function as a black box that might contain arbitrarily-many instrumentation sites. Thus, the next-sample countdown may change arbitrarily across any non-weightless function call. This requires splitting acyclic subgraphs at function calls, which in turn makes the subgraphs smaller. Smaller acyclic subgraphs require more frequent threshold checks, harming performance.

Suppose instead we were to identify a maximum threshold weight for an entire function. For some functions this may not be bounded, but for many (especially small, loop-free leaf functions) a finite bound will exist. This information can be exploited by the caller to compute its own acyclic subgraph thresholds, since a call to a function with threshold weight n can only reduce the next-sample countdown by at most n from the perspective of the caller. Weightless functions are simply a special case of the more general class of bounded-weight functions.

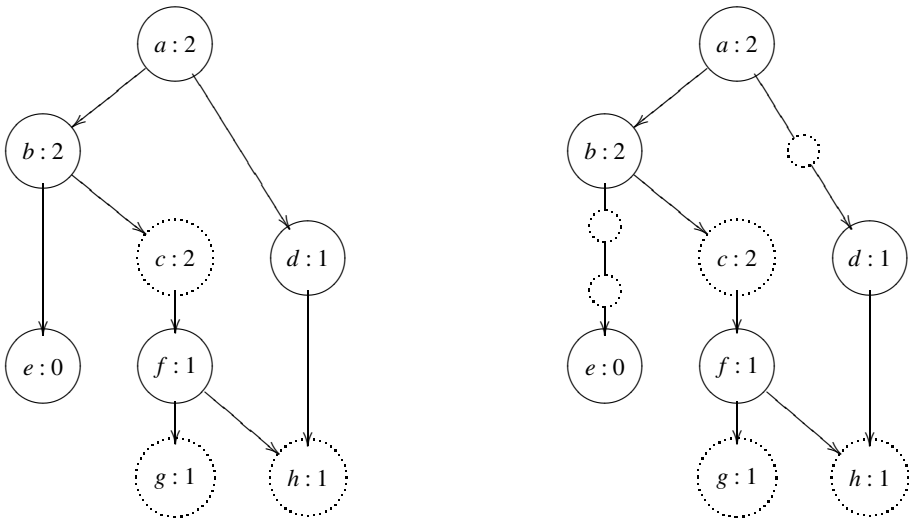
Path Balancing. When the fast clone consists of simple, straight-line code, a native compiler may be able to coalesce multiple countdown decrements into a single larger adjustment. For example, `gcc` performs this optimization provided that the countdown is cached in a local variable per Sect. 3.2. However, decrement coalescing cannot extend across branches, because the multiple forward paths may contain different numbers of instrumentation sites and therefore require different net adjustments to the countdown.

Path balancing generalizes decrement coalescing to arbitrary acyclic subgraphs. The key is to ensure that all forward paths through an acyclic subgraph cross the same number of instrumentation sites. Imbalances occur at branches. When a control flow graph node has multiple successor paths with different weights, extra "dummy" sites are added to the start of those successor paths that have fewer "real" sites than their siblings, thereby creating balance. When all branches in a subgraph are balanced, the entire subgraph is balanced as well.

Figure 3a gives an example of an acyclic subgraph before balancing. Nodes with instrumentation sites have dotted outlines. Notes are lettered for ease of reference, and the number in each node gives the maximum weight of all paths forward from that node. The entire subgraph has threshold weight 2 but individual paths cross 0 (*abe*), 1 (*adh*), or 2 (*abcfg*, *abcfh*) sites. Branch nodes a, b, and f may require balancing. Branch a does have imbalanced successors: one dummy site must be added on the *ad* edge. Branch b is also imbalanced: two dummy sites must be added on the *be* edge. Branch f is already balanced: both successors already have matching weights.

Figure 3b shows the same acyclic subgraph after balancing. Three unlettered dummy sites have been added. The threshold weight for the entire subgraph (2) is now the exact number of sites crossed on each of the four paths through the subgraph starting from entry node a.

Balancing is not an optimization in and of itself. Rather, it actually adds instrumentation in the form of dummy sites. However, once a site is balanced, we can optimize the code as follows. Just before the first node of the fast clone, decrement the next-sample



(a) Before balancing (b) After balancing

Fig. 3. Example of path balancing

countdown by the threshold weight of the entire subgraph (for example, "`countdown -= 2`" just before node *a* in Fig. 3b). This decrement accounts for exactly the number of unary decrements that would have occurred in this subgraph. Elsewhere in the fast clone, wherever a real or dummy instrumentation site would have appeared, do nothing. The decrements have already been accounted for and there is no other work to do.

The slow clone must decrement and check the countdown at each instrumentation site as before, because on the slow clone we do need to know exactly when a site should be sampled. Furthermore, even dummy sites must decrement the countdown and reset it if it reaches zero. This requirement ensures that both the fast and slow clones behave the same with respect to counting down to the next sample, at the expense of making the slow clone even slower. Also, adding dummy instrumentation sites means that the countdown will need to be reset more often, so a slow random number generator will be more of a liability here.

In total, path balancing makes the fast clone faster and the slow clone slower. The idea for the path balancing algorithm arose in discussions between the author and Cormac Flanagan, but has not yet been implemented or evaluated. We offer it here as an example of leveraging the regular structure of sampled instrumentation code using an intraprocedural analysis of quite modest complexity.

4 Statistical Debugging

Some instrumented behaviors may always occur or may never occur; these are invariants in practice (and possibly in theory). Most behaviors vary from run to run. If variation in some instrumented predicate correlates with failure of runs, then we call that

predicate a *bug predictor*. The correlation may be imperfect: nondeterministic bugs can allow apparent success even in runs that ought to have failed. Sparse sampling means that even a completely deterministic failure predictor will not be observed on most runs where it does occur. For this reason, we must look for broad statistical trends in program (mis)behavior across large numbers of runs. A single run tells us virtually nothing, but because the sampling transformation is statistically unbiased, trends over many runs can guide developers to the root causes of recurrent problems.

Statistical debugging refers to the task of finding bug-predictive behaviors among the feedback data collected from large numbers of instrumented runs. Members of the machine learning community have expressed considerable interest in this problem, which can be seen as an unusual example of feature selection (finding bad behaviors) or clustering (grouping failures by the bug that caused them). Statistical models considered, either by me and my collaborators or by completely independent groups, include regularized logistic regression [8,11], probability density function comparison [12], likelihood ratio testing [13,14], iterative bipartite graph voting [15,16] three-valued logic [17] support vector machines [18], random forests [18], Delta Latent Dirichlet Allocation [19], and quite possibly others of which I am shamefully unaware. We refrain from reviewing the details of any of these algorithms here; the interested reader may read the original papers, perhaps with a statistics textbook or colleague nearby for guidance. The approaches vary in their ability to deal with multiple bugs, non-deterministic failures, sparsely sampled data, extremely large datasets, and other qualities. Most share a similar structure of analysis output: a ranked list of instrumented program behaviors that have been identified as bug predictors. Such a list can be presented to a developer to guide further triage, diagnosis, and remediation.

Given the messy, incomplete nature of sampled feedback data, one might think that formal static program analysis has little to contribute to statistical debugging. This is incorrect. Some of CBI's most interesting statistical analysis work takes place at this late stage, *after* the core statistical models have been built and used to identify bug predictors. Indeed, the analysis methods used here far outstrip those found in CBI instrumenting compilers, both in terms of their current sophistication and in their future potential.

4.1 Static Analysis as Currently Used

When hunting down a bug, a ranked list of bug predictors is a good start but it is not the complete story. Developers must still understand why the highlighted misbehaviors can lead to failure, and this is not always easy. Several static program analyses have been used to place bug predictors back into the context of the source program and help developers understand how they relate to failure.

One common goal is to stitch isolated bug-predictive program points together into extended failure paths. Developers can then walk through a doomed run (or an approximate reconstruction thereof) step by step to see where things fell apart. My own early attempts at this [20] have been bested by subsequent work by Lal et al. [21]. Lal's approach uses weighted pushdown systems, a powerful generic formalism for expressing context-sensitive interprocedural dataflow analyses [22,23]. Propelled by this engine, Lal's analysis reconstructs paths that proceed from program entry through high-ranked

bug predictors to a point of failure. Paths obey feasibility constraints imposed by any of a variety of dataflow analyses. Jiang and Su build partial faulty-path segments using an efficient control-flow graph traversal with backtracking [18]. The search is a static analysis, but is heuristically guided by using CBI feedback data to guess likely execution paths at branches.

Static analysis also plays a role in comparing the quality of ranked bug predictor lists. Cleve and Zeller [24] propose a quantitative approach that measures the distance between the code blamed by some tool and the actual location of the bug. Distances are measured in the program dependence graph (PDG), following a model by Renieris and Reiss [25]. This metric is intended to simulate an idealized programmer who first examines code blamed by the tool, then proceeds outward across PDG edges until the true flaw is found. Numerous other researches (myself included) have adopted this PDG-distance metric metric, even though there is no experimental evidence to support the idea that real programmers behave in this manner. Furthermore, the very notion of finding the true flaw is ill-defined when the bug is a sin of omission. A forgotten conditional branch, for example, corresponds to a PDG node that should have been present but is not. How can we measure the distance to a node that is not there? Better models of developer behavior are needed, and unfortunately modeling humans is well outside the domain of static program analysis.

4.2 Static Analysis Potential

In each of the examples given above, static analyses were not being applied in isolation. Rather, they were used in conjunction with statistical models built from dynamic data. Static and dynamic/statistical approaches have complementary strengths and weaknesses. A static analysis may provide strong guarantees (modulo loose C semantics) for a limited set of questions, while dynamic/statistical information can provide best-estimate guesses for nearly any question, but never with absolute certainty. If we combine the two carefully, we may achieve the best of both worlds.

Dynamic data can be used as a hypothesis generator to drive deep static analyses. Nimmer and Ernst's fusion of Daikon and ESC/Java is a classic instance of this style of dynamic-to-static feedback [26]. In the CBI context, bug predictors identified in dynamic data could suggest initial conditions on program state that warrant closer static inspection. If failures are common when `p` is null on line 94, let static analysis explore the antecedent causes or subsequent implications of that condition. Conditioned slicing, for example, may be appropriate for pursuing such leads [27], if it can be made to work for C programs of realistic size and complexity.

Useful information can also flow from the static world to the dynamic/statistical world. We suggested earlier that static analyses could prove some instrumentation redundant, and therefore removable. A related idea would be to use static analysis to reconstruct some of the data omitted by sparse sampling. To take one very simple example, any observation at a given instrumentation site reveals that the control-flow dominators of that site must also have been executed, even if sparse sampling caused this fact to be omitted from the raw feedback data. Deeper static analysis could infer missing information about data values as well as control flow. Recovering missing data effectively increases the sampling rate, resulting in a less noisy dataset for statistical

Table 1. Applications in CBI's public deployment, illustrating the infeasibility of whole-program analysis. The count of plug-ins provided includes only those that are part of the main application distribution. Any number of additional plug-ins could be provided by third parties.

Application	Lines of Code	Shared Libraries Used	Plug-Ins Provided
Evolution	441,644	107	45
GIMP	854,530	50	188
GNOME Panel	69,164	82	0
Gnumeric	351,461	85	36
Nautilus	137,394	89	0
Pidgin	387,962	56	56
Rhythmbox	133,281	95	12
SPIM & XSPIM	28,139	4 & 18	(not extensible)

modeling. Several key questions remain unanswered about this idea. It is not clear that available model checkers and theorem provers can scale up to the problem sizes that arise from this sort of analysis. (Indeed, our own preliminary exploration suggests that they do not.) Additionally, missing-data reconstruction introduces bias, as not all missing data is equally easy to infer from available evidence. Whether this bias fouls the results of statistical models, and how it can be compensated for, remain unknown.

Lastly and most speculatively, perhaps richer statistical models could draw simultaneously from both static and dynamic sources of information, instead of merely feeding one into the other. *Statistical relational learning* describes a broad class of methods for building statistical models over domains with rich internal structure [28]. Programs have rich internal structure, and research on static program analysis offers myriad strategies for extracting that structure. Exposing that static structure in a way that allows principled integration with dynamic feedback data may allow tremendous advances in program understanding and debugging. I will be the first to admit that I do not know how to do this . . . yet. But I intend to find out.

5 A Closing Rant on Analysis Robustness

Invited papers should stir things up a bit. In case this paper has not already done so, I will now indulge in a closing rant likely to agitate (if not offend) many readers.

One practical difficulty in using static analyses with CBI is that many implementations of interesting static analyses don't actually work. They worked at one time, on a few small examples sufficient to write a paper. But give them tens of thousands of lines of real C code and many analysis implementations simply fall apart. The more interesting the analysis in theory, the more brittle its implementation tends to be in practice.

One common and brittle assumption is that whole-program analysis works for mainstream applications. In my subjective experience, it does not. Even if sheer code size were not a problem, I do not have a single real application of interest where all code is available at compilation/analysis time. Table 1 summarizes applications now in CBI's public deployment. Observe that every application uses numerous shared libraries, and

that all but one (SPIM/XSPIM) can be extended at run time through plug-ins. Thus, the idea that whole-program analysis can see all application code is simply a myth.

Even among the code that is present for analysis, real world software is not always pretty. I am no C apologist, and I look forward to C's eventual replacement by stricter languages that are more amenable to analysis. Until that happens, I need analyses that handle the full, horrific glory that is C: pointer casting, threads, stack-unwinding `longjmp`, dynamic code loading ... the whole terrifying bag of C tricks. This is the language as it is used in the real world, and this is the language that a static analysis must handle if it is to be used with CBI in the near term.

6 Conclusion

Cooperative Bug Isolation operates in an messy world of unsafe languages, corrupted heaps, non-deterministic failures, and incomplete data. Faced with such obstacles, I rarely achieve or even seek software perfection. Rather, I describe my research as trying to make software suck less. In this ugly domain, even the findings of a "sound" static analysis may not be entirely trustworthy. Historically, CBI has shied away from deep static analysis in favor of brute-force data collection and statistical modeling. However, static analyses can play an important role if applied wisely. Analysis can make instrumentation more selective and efficient before deployment, and can augment statistical modeling in numerous ways after feedback data arrives. I believe that the most powerful approaches will carefully combine static, dynamic, and statistical methods to leverage the unique strengths of each. If we can do that, then perhaps even software perfection is not to much to hope for.

References

1. Liblit, B.:Cooperative Bug Isolation (Winning Thesis of the 2005 ACM Doctoral Dissertation Competition). LNCS, vol. 4440. Springer, Heidelberg (2007)
2. Driscoll, E., Cooksey, G.: CBI++. CS706 class project, University of Wisconsin–Madison (December 2006)
3. Hunter, J., Kolpin, G., Saeed, U.: CBI instrumentation for Java bytecode. CS706 class project, University of Wisconsin–Madison (December 2005)
4. Kolpin, G.: Jikes CBI implementation details. Independent study project, University of Wisconsin–Madison (May 2006)
5. Liblit, B.: Guide to the bug isolation sampler (January 2008), `http://www.cs.wisc.edu/cbi/developers/guide/`
6. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Horspool, R.N. (ed.) CC 2002 and ETAPS 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
7. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. 69(1-3), 35–45 (2007)
8. Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: PLDI, pp. 141–154. ACM, New York (2003)
9. Arnold, M., Ryder, B.G.: A framework for reducing the cost of instrumented code. In: PLDI, pp. 168–179 (2001)

10. Hirzel, M., Chilimbi, T.: Bursty tracing: A framework for low-overhead temporal profiling (November 24, 2001)
11. Zheng, A.X., Jordan, M.I., Liblit, B., Aiken, A.: Statistical debugging of sampled programs. In: Thrun, S., Saul, L.K., Schölkopf, B. (eds.) NIPS, MIT Press, Cambridge (2003)
12. Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.P.: SOBER: statistical model-based bug localization. In: Wermelinger, M., Gall, H. (eds.) ESEC/SIGSOFT FSE, pp. 286–295. ACM, New York (2005)
13. Jones, J.A., Harrold, M.J.: Empirical evaluation of the Tarantula automatic fault-localization technique. In: Redmiles, D.F., Ellman, T., Zisman, A. (eds.) ASE, pp. 273–282. ACM, New York (2005)
14. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable statistical bug isolation. In: Sarkar, V., Hall, M.W. (eds.) PLDI, pp. 15–26. ACM, New York (2005)
15. Zheng, A.X., Jordan, M.I., Liblit, B., Naik, M., Aiken, A.: Statistical debugging: simultaneous identification of multiple bugs. In: Cohen, W.W., Moore, A. (eds.) ICML. ACM International Conference Proceeding Series, vol. 148, pp. 1105–1112. ACM, New York (2006)
16. Wassel, H.M.G.H.: An enhanced bi-clustering algorithm for automatic multiple software bug isolation. Master's thesis, Alexandria University, Egypt (September 2007)
17. Arumuga Nainar, P., Chen, T., Rosin, J., Liblit, B.: Statistical debugging using compound Boolean predicates. In: Rosenblum, D.S., Elbaum, S.G. (eds.) ISSTA, pp. 5–15. ACM, New York (2007)
18. Jiang, L., Su, Z.: Context-aware statistical debugging: from bug predictors to faulty control flow paths. In: Stirewalt, R.E.K., Egyed, A., Fischer, B. (eds.) ASE, pp. 184–193. ACM, New York (2007)
19. Andrzejewski, D., Mulhern, A., Liblit, B., Zhu, X.: Statistical debugging using latent topic models. In: Kok, J.N., Koronacki, J., de Mántaras, R.L., Matwin, S., Mladenic, D., Skowron, A. (eds.) ECML 2007. LNCS (LNAI), vol. 4701, pp. 6–17. Springer, Heidelberg (2007)
20. Liblit, B., Aiken, A.: Building a better backtrace: Techniques for postmortem program analysis. Technical Report CSD-02-1203, University of California, Berkeley (October 2002)
21. Lal, A., Lim, J., Polishchuk, M., Liblit, B.: Path optimization in programs and its application to debugging. In: Sestoft, P. (ed.) ESOP 2006 and ETAPS 2006. LNCS, vol. 3924, pp. 246–263. Springer, Heidelberg (2006)
22. Reps, T.W., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci. Comput. Program. 58(1-2), 206–263 (2005)
23. Lal, A., Reps, T.W., Balakrishnan, G.: Extended weighted pushdown systems. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 434–448. Springer, Heidelberg (2005)
24. Cleve, H., Zeller, A.: Locating causes of program failures. In: Roman, G.C., Griswold, W.G., Nuseibeh, B. (eds.) ICSE, pp. 342–351. ACM, New York (2005)
25. Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: ASE, pp. 30–39. IEEE Computer Society, Los Alamitos (2003)
26. Nimmer, J.W., Ernst, M.D.: Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. Electr. Notes Theor. Comput. Sci. 55(2) (2001)
27. Canfora, G., Cimitile, A., Lucia, A.D.: Conditioned program slicing. Information & Software Technology 40(11-12), 595–607 (1998)
28. Getoor, L., Taskar, B.: Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning). MIT Press, Cambridge (2007)