An Overview of the Saturn Project

Alex Aiken
Stanford University
aiken@cs.stanford.edu

Suhabe Bugrara
Stanford University
suhabe@cs.stanford.edu

Isil Dillig
Stanford University
isil@cs.stanford.edu

Thomas Dillig
Stanford University
tdillig@cs.stanford.edu

Brian Hackett
Stanford University
bhackett@cs.stanford.edu

Peter Hawkins
Stanford University
hawkinsp@cs.stanford.edu

Abstract

We present an overview of the Saturn program analysis system, including a rationale for three major design decisions: the use of function-at-a-time, or summary-based, analysis, the use of constraints, and the use of a logic programming language to express program analysis algorithms. We argue that the combination of summaries and constraints allows Saturn to achieve both great scalability and great precision, while the use of a logic programming language with constraints allows for succinct, high-level expression of program analyses.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

General Terms Design, Experimentation, Languages

Keywords program analysis, verification, boolean satisfiability

1. Introduction

Saturn [1] is a system for the static analysis of programs. Saturn aims to be both highly scalable and precise, with the goal of eventually being able to verify the absence of certain kinds of bugs in real systems. Saturn is based on three main ideas:

- Saturn is summary-based: each function f is analyzed separately, producing a summary of f's behavior. At call sites for f, only f's summary is used. Summary information may also be attached to types, global variables and loops.
- Saturn is also constraint-based: an analysis is expressed as a system of constraints describing how the state at one program point is related to the state at adjacent program points. The primary constraint language used in Saturn is boolean satisfiability, with each bit accessed by a procedure or loop represented by a distinct boolean variable.
- Program analyses in Saturn are expressed in a logic programming language with support for manipulating constraints and accessing summaries.

PASTE'07 June 13–14, 2007, San Diego, California, USA.

In combination, these ideas give Saturn the ability to succinctly express precise analyses while also providing the ability to scale to very large programs. The use of constraints and logic programs allows succinct analyses, which are easier to understand and verify correct than analyses written at a lower level of abstraction. Bit-level path-sensitive analysis gives precision, while analyzing a single function at a time and summarization give scalability—Saturn is routinely used to run analyses on the entire Linux kernel (with more than 6MLOC) and other large open source projects.

Currently, the Saturn project is pursuing two related goals. The first goal is to understand how programmers structure large systems in practice; that is, to study and describe software as it actually exists in the wild. Distilling and describing useful structure in large systems requires automatic assistance (i.e., program analysis) to digest and systematize the huge amount of raw data (i.e., programs). The second goal is to build tools that can prove useful properties of programs, either finding bugs or proving the absence of bugs. The Saturn analyses that have been developed to date have found thousands of previously unknown bugs in widely used open source systems [21, 20, 8, 6]. We have also found that discovering the patterns that programmers use to structure code for their own understanding is often a crucial step in designing practical bug-finding or verification tools.

Figure 1 gives a block diagram of the Saturn toolchain. The C frontend (currently CIL [15]) encodes the abstract syntax trees of the program as relations, storing them all in a few syntax databases. A program analysis, written in Saturn's Calypso programming language,[1] is then run on each function in the syntax database by the Calypso interpreter, constructing constraints and querying constraint solvers, constructing summary information and producing error reports. These reports can then be viewed either as plaintext or via an XML-based UI, depending on the analysis.

This paper gives a brief tour of the major components of Saturn: the use of summaries (Section 2), constraints (Section 3), the logic programming language (Section 4), and how they fit together.

2. Summary-Based Analysis

One of the defining characteristics of Saturn is that it is *summary-based*: the unit of analysis is the individual function, and the only way for function f to refer to the results of analysis of function g is through g's *summary*. While summary-based analysis has been known for a long time (dating back at least to early work on type inference [14]), it is at odds with many recent efforts in fully automatic program analysis, including other projects involving mem-

[1] Calypso is a moon of Saturn.

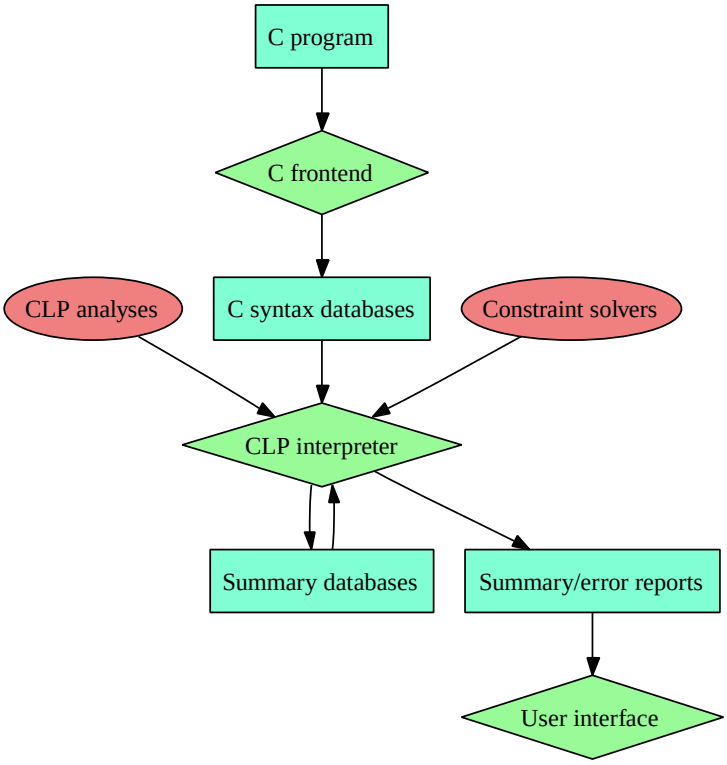

Figure 1. Saturn toolchain structure

bers of our own group. It is worthwhile, then, to examine the arguments for and against a summary-based approach.

On a purely semantic level, there are two appealing aspects of summary-based analysis. First, it naturally supports context sensitivity, since any polymorphism in the summary is easily exploited when the summary is instantiated at different call sites. Second, it is also natural in Saturn to write *compositional* summary-based analyses that are applicable to *open* as well as *closed* programs. (An open program is one with some free identifiers, such as a library in isolation; a closed program is complete and can be executed.) By compositional, we mean that the analysis of a function makes no assumptions about the possible environments in which that function is called, and thus the callers need not be present to compute useful information about the function, which is just the case of analyzing a library in isolation. The alternative is *whole-program* analysis, where a representation of the entire program is constructed and analyzed at once. Some whole-program analyses (e.g., monomorphic forms of receiver class analysis for object-oriented languages [16]) cannot be understood in a compositional way because the analysis presumes knowledge of the specific contexts in which each function is used in the larger program. It is worth noting, however, that many other whole-program analysis systems (most notably those based on solving systems of constraints [10]) could in principle be adapted fairly easily to a compositional style of analysis, but this has not yet been done. The fact is that compositional analyses are more work to write than whole-program analyses. At a minimum, one must design and then construct polymorphic summaries for each function, and often the analysis goes to extra effort to construct a representation of all possible environments in which the function may be used. While compositional analyses are arguably the most natural to write in Saturn, the system is capable of expressing whole-program analyses as well (including demand-driven interprocedural analyses), and in practice, at least so far, many complete Saturn analyses are made up of components of both styles.

The real benefit of summary-based analysis, in our opinion, lies in its systems engineering advantages. The limiting factor for most analysis systems is not time, but space: for large programs it is difficult to construct a representation that fits entirely in main memory. Rather than expend effort finding ways to compress program representations for space efficiency, a natural alternative is to do what databases and large scientific applications do when faced with very large datasets: use out-of-core algorithms that stage the computation in small pieces between disk and main memory. Summary-based analysis fits this paradigm well, with the individual function being the unit of work. At each point in time, only one function f is represented in main memory, together with the information about its current summary and the summaries of any functions, globals, and types f refers to. The rest of the functions and their summaries are stored on disk. As the portion of the program being manipulated is always small, memory pressure is low and constant (at least in well-designed analyses) and it is possible to write analyses that can, in principle, scale to arbitrarily large programs. Because the cost of analyzing a function is proportional to the size of the function and the size of the summaries it depends on (e.g., the summaries for any callees of the function), two assumptions underly this design:

- The size of function summaries is bounded by a constant—summaries do not grow with program size. It is the analysis designer's responsibility to ensure this property holds or to accept limited scalability. We discuss this requirement further in Section 3.
- The size of a function is bounded by a constant. Clearly the analysis designer has no control over function size, but normal programming practice ensures that this assumption is well matched to real programs. For example, a recent version of the Linux kernel has over six million lines of code when all device drivers are included. So far as we know, there has been no successful in-memory analysis of the entire kernel, though subsets of millions of lines have been analyzed [11]. From the point of view of a summary-based analysis the situation is not so difficult. The average function in the kernel has only 29 lines of code, and the median function length is just 16 lines. There are some very large functions, but they are rare: the largest function has 2,249 lines, there are six functions with 1,000 lines or more, and only 91 functions (less than 0.1% of all functions) have at least 500 lines of code.

A conscious trade-off in Saturn is that we have designed for scalability rather than speed. The analysis of a single function incurs expensive operations (e.g., reading and writing disk), and the function being analyzed is usually small, suggesting that the ratio of useful work to system overhead may be a problem. However, another advantage of analyzing functions separately is that the process is easily parallelized, with parallelism limited only by analysis dependencies between different functions. We use compute clusters of 40-100 cores to run Saturn analyses in parallel and normally achieve 80-90% efficiency. As a result, Saturn's raw performance, at least on a sufficiently large cluster, appears competitive with other systems of which we are aware; most Saturn analyses complete in a few hours on programs the size of Linux.

3. Constraints

A standard methodology in program analysis is to treat analysis as a constraint satisfaction problem: a pass (or passes) over the program generate constraints that capture the conditions under which the property of interest holds, and a separate constraint solver or decision procedure reports whether those constraints have a solution.

There are many constraint theories that have application in program analysis, but as the name suggests, the one used most in Saturn is boolean satisfiability (SAT).

Saturn attaches summaries not only to functions but also to loops; conceptually, each loop is treated as a tail recursive function. This decision establishes a useful invariant, namely that every function/loop body is iteration free—every program point in the function/loop body is executed at most once per call. Since each program statement accesses a fixed, known number of memory locations, this implies that every memory location accessed by a function body can be statically named.

We illustrate Saturn's use of constraints and the interaction with function summaries using Saturn's alias analysis [8]. We represent aliasing indirectly using *guarded points-to graphs*. In a *points-to graph* nodes are *labels* (names of memory locations, which we do not further define here) and edges (l_1, l_2) mean that pointers at locations in l_1 may point to locations in l_2. Aliasing information is recovered easily from points-to graphs; for example, edges (x, l) and (y, l) show that x and y may be aliased, as they both may point to locations in label l. Guarded points-to graphs generalize points-to graphs by associating each edge with a *guard*, a constraint stating under what condition the points-to relationship holds. Guards contribute to the precision of the points-to analysis; for example, if x and y may both point to l, but with guards that cannot simultaneously hold, they are not aliased. We use formulas over boolean variables b for guards:

$$\begin{array}{rcl} g \in \text{Guard} & ::= & \mathsf{true} \mid \mathsf{false} \mid b \mid g_0 \wedge g_1 \mid g_0 \vee g_1 \mid \neg g \\ \rho \in \text{PTGraph} & = & (\text{Label} \times \text{Label}) \rightarrow \text{Guard}_\perp \end{array}$$

The Saturn alias analysis is intraprocedurally path-sensitive. A relatively simple way to achieve path-sensitivity is to use fresh boolean variables to separate points-to information along branches. Even without branch condition information (i.e., ignoring the actual predicate of the conditional), path information can track correlations between side effects. For example, path information is sufficient to statically determine that the following C fragment that performs a conditional swap cannot introduce aliasing between `a` and `b`:

```
a = x; b = y; // *x != *y
if (...) { t = a;  a = b;  b = t; }
f(a,b);
```

It is easy to see that `a` and `b` are never aliased at the call `f(a,b)`. As an example, we give a simplified version of the rule for analyzing conditionals.

$$\begin{array}{l} [\![\textbf{if}\ ?\ c_0\ c_1]\!]\ \rho\ (l, l') \ = \\ \quad \textbf{let}\ \rho_0 = [\![c_0]\!]\rho\ \textbf{in} \\ \quad \textbf{let}\ \rho_1 = [\![c_1]\!]\rho\ \textbf{in} \\ \quad \textbf{let}\ b \text{ be a fresh boolean variable } \textbf{in} \\ \qquad (b \wedge \rho_0(l, l')) \vee (\neg b \wedge \rho_1(l, l')) \end{array}$$

This rule illustrates how control information is incorporated into guards: the guards for the two branches are disjoined but still distinguished by b in the final points-to graph for the statement. If the behavior of the two branches is the same with respect to l, then $\rho_0(l, l') \equiv \rho_1(l, l')$ and we can simplify the resulting guard by removing the reference to b with no loss of information.

While this rule computes only path information, most Saturn analyses use a more sophisticated analysis that gives a bit-level model of the branch condition itself [21], which is crucial for correlating different branches within a procedure. For example, in the following code skeleton, which is idiomatic in many low-level programs, the correlation, if any, between the branches can only be understood by examining the effect of the operators on each individual bit:

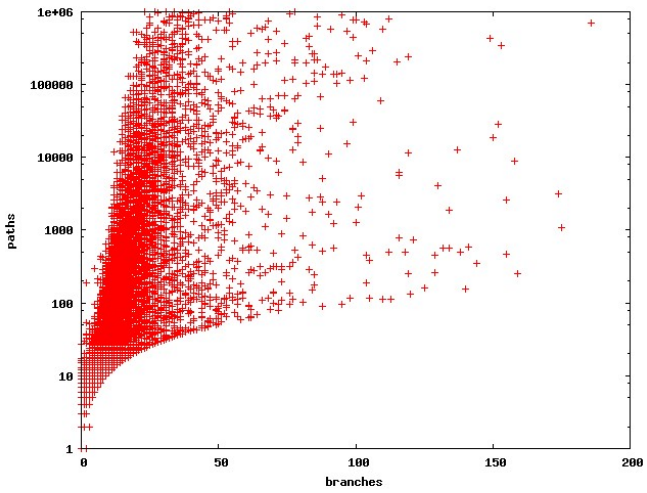

Figure 2. Branches vs. number of paths in Linux.

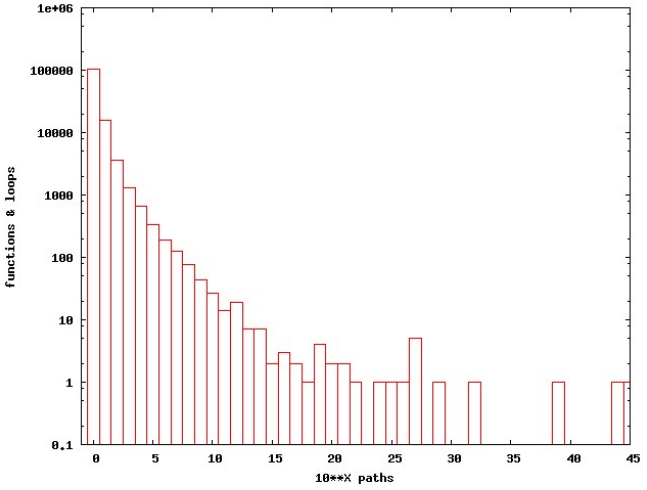

Figure 3. Functions/loops with a given number of paths in Linux.

```
if (x & MASK1) . . .
. . .
if (x & MASK2) . . .
```

Returning to the discussion of the overall architecture of Saturn, there are important interactions between the summary-based approach and Saturn's use of constraints. Bit-level path sensitivity is difficult to scale to large programs; bounded model checking systems that use a similar approach have been limited to programs with hundreds of lines of code [3, 4]. As discussed above, the requirement for summaries at every loop and function boundary makes it reasonable to assume that each function/loop body is a small, loop-free piece of code, and that is why Saturn can make use of something as expensive and precise as full path-sensitive modeling of every bit manipulated in the function body. Figures 2 and 3 quantify the difficulty of path-sensitive intraprocedural analysis in the Linux operating system. Figure 2 plots the number paths in a procedure (or loop body) as a function of the number of branches. Note that the y-axis is on a log scale. This figure only displays the dense part of the data; there are a few procedures not shown,

one with about 300 branches and 10^{45} paths, and one with about 1500 branches and fewer than 100,000 paths. Nevertheless, the data shown in Figure 2 is discouraging enough: not only do some procedures have a lot of branches, but those branches are arranged in a way that clearly often realizes the worst-case exponential growth in the number of paths. What Figure 2 does not show, however, is that almost all of the points in the plot are concentrated near the origin. Figure 3 plots the number of functions/loops that have between 10^x and 10^{x+1} paths for each value of x; note that both axes are on a log scale. As can be seen, the overwhelming majority of functions have very few paths. Indeed, the median number of paths in a function or loop body in Linux is 3, and 95% of functions/loops have fewer than 100 paths. Also not depicted in the graphs is that more often than not it is possible to do the lossless merging of information where paths join as outlined above. Thus, only about the .1% of largest functions/loops (those with 10^{10} or more paths) are difficult to build a complete model for in less than a minute.

Full path-sensitivity is a level of precision undoubtedly unneeded for some applications, but it has proven very convenient in all of the applications we have explored. The boundary between the precise intraprocedural modeling and the scalable interprocedural analysis is the function summary. It is the task of the analysis designer to choose an abstraction at function/loop boundaries that captures the important features of the application while being sufficiently compact to ensure scalability. In Saturn, analysis design is summary design.

Returning to the points-to example, the final result of the intraprocedural analysis is a pair of guarded points-to graphs: the assumed guarded points-to graph on input to the function, and the final guarded points-to graph that results on exit from the function. In considering an appropriate summary for this information, it is important that whatever the summary is, it be bounded in size to ensure termination. The guards, then, present a problem, because if guards appear in summaries then guards can propagate from callees to callers at function call sites, and no bound on the size of the guards can be guaranteed. Our current solution is to test all guards in the initial and final points-to graphs for satisfiability and to promote all satisfiable guards to `true`, converting the guarded points-to graphs to unguarded points-to graphs in summaries. This approximation is sound, as it can only overestimate the conditions under which one location points to another. Thus, path-sensitive points-to information is only intraprocedural in our current implementation of the alias analysis. Observing that the number of locations accessed directly by a loop-free function is fixed by the program and that the number of nodes in a function's summary points-to graphs is bounded by the number of locations, it is clear that discarding path-sensitive guards in function summaries guarantees that summaries are bounded in size.

It turns out, however, that this approach to summarization is not enough for scalability and that alias analysis of many programs, especially large ones, consumes a great deal of memory, which has also been the common experience with previous flow-sensitive, context-sensitive points-to analyses. We have found that the expense is almost entirely due to recomputing alias relationships among user-defined data types and global variables, which, once introduced by any program statement, tend to propagate throughout the program. The solution we have adopted in the points-to analysis and in other applications is to also have *type* and *global summaries*, which are flow-insensitive facts associated with user-defined types and global variables. Recording points-to relationships that are solely within user-defined types just once with that type (e.g., one cell points to another cell of the same type) dramatically reduces the amount of information included in function summaries. Previous work in path-sensitive, context-sensitive points-to analysis scaled to the low tens of thousands of lines of code [13]. Using these different levels of precision (flow- and path-sensitive within functions, flow-sensitive interprocedurally, flow-insensitive for facts about data structures and global variables), enables us to scale our context-, flow-, and partially path-sensitive analysis to the entire Linux kernel. The false aliasing rate for Saturn's points-to analysis is 26% (about one in four points-to relationships does not correspond to potential aliasing at run-time) [8]; so far as we know, it is currently both the most precise and most scalable points-to analysis in existence.

4. The Analysis Language

All program analysis frameworks provide a language in which users can express analyses. In Saturn we wished to avoid decisions that would commit us to a particular formalism or preclude experimentation with different approaches for different analysis problems, so expressiveness was more important than algorithmic efficiency. For this reason we opted to use a logic programming language as the medium in which analyses are written. Saturn's language, Calypso, is a general-purpose logic programming language with extensions to support both constraints and function summaries.

Using logic programming to express dataflow analyses dates to Ullman [19] and subsequently Reps explored using logic programs to express demand-driven analysis algorithms [17]. At Microsoft Research in the late 1990's, a logic programming language for querying abstract syntax trees was used to find thousands of bugs in production code by searching for largely syntactic (but potentially complex) erroneous coding practices [5]. Two recent projects have used logic programming as the notation for large scale program analysis [12, 9], and we have borrowed heavily from their experience.

The motivation for logic programming in program analysis is best given by example. Consider the problem of evaluating dereference expressions under a flow-sensitive points-to analysis; particular dereferences may have many different targets depending on the point at which they are evaluated. In a published paper one might find inference rules such as:

$$\frac{\texttt{pointsto}(\texttt{P},\texttt{X},\texttt{Y})\ \ \texttt{eval}(\texttt{P},\texttt{E},\texttt{X})}{\texttt{eval}(\texttt{P},*\texttt{E},\texttt{Y})}$$

where `pointsto(P,X,Y)` indicates that at program point `P`, `X` may point to `Y`, and `eval(P,E,X)` indicates that at program point `P`, expression `E` may evaluate to `X`. The inference rule notation (or small variations of it) is standard for presenting program analysis systems.

In Saturn we would write the rule in the following way. We need one more predicate `exp_deref(ED,E)`, which indicates that `ED` is the dereference of `E`; this simply gives a name to the expression $*$`E` in the notation above. We can then write the dereference operation in a Prolog-like syntax as:

```
eval(P,ED,Y) :- exp_deref(ED,E), eval(P,E,X),
    pointsto(P,X,Y)
```

Logic programming is the natural implementation of inference rules; with a logic programming language, the gap between the formal description of a program analysis algorithm and its implementation is nearly erased.

Calypso's implementation is based on a bottom-up interpreter for pure, first-order logic programs with a unique combination of features. Most analyses perform a mixture of bottom-up and demand driven computation, so in order to allow demand-driven behavior with a bottom-up interpreter we use a variant of the magic-sets transformation [2]. To help analysis-writers locate bugs in their code quickly, Saturn performs strong static type and mode

checking, and optional dynamic determinism checking, features which are closely related to those in Mercury [18].

Calypso differs from previous efforts primarily in its support for constraints and for summary-based analysis. Several constraint solvers can be plugged into Saturn, and used by individual analyses. Each constraint solver has its own Calypso interface, a set of primitive predicates for creating, manipulating, and querying constraints. For example, the predicate `#and(G0,G1,G)` can be used by an analysis to construct a boolean formula `G` by taking the conjunction of formulas `G0` and `G1`. Querying the predicate `#sat(G)` invokes a SAT solver and tests whether `G` is satisfiable. Extending our example from above, if we change the `eval` predicate to `eval(P,E,X,G)`, where `G` indicates the guard under which `E` evaluates to `X` at `P`, we can test whether two expressions `E0` and `E1` at point `P` could alias — that is, there is some `X` such that `E0` and `E1` may simultaneously evaluate to `X` — with the following Calypso code:

```
may_alias(P,E0,E1) :-
    eval(P,E0,X,G0), eval(P,E1,X,G1),
    #and(G0,G1,G), #sat(G).
```

Again, this code closely matches an inference rule that might be used to formally specify the analysis. Note that Calypso itself is implemented using a conventional interpreter, and the constraint solver is only involved when querying predicates such as `#and` and `#sat`. The main reason for keeping the constraints well-separated from the core language is to allow experimentation with different constraint systems and to build applications that use multiple and mixed [7] constraint systems. While SAT is used most often, we have also used integer constraint solvers for some analyses and are experimenting with yet additional solvers. A contrasting design can be seen in [12], where a single constraint formalism is built into the logic programming language.

In addition to using constraint solvers, Calypso analyses must be able to generate and query summary information for performing interprocedural analysis. Summary-based analysis is given first-class status in Calypso through the use of *sessions*. A session is a persistent set of relations representing facts about a program object, such as the abstract-syntax tree for a function or a summary for a function or a type. An analysis consists of a logic program that is executed separately in the context of each session in a database (typically a database of function and loop bodies), and can freely query facts from any session, and compute new facts and add them to any session. Typically, an analysis is executed for every function or loop body in the program, generating a summary session for that function or loop and querying the summary sessions of any of that function or loop's callees. An interprocedural scheduler keeps track of dependencies between sessions, and so if an analysis updates a session, then all analyses that had previously queried that session are rerun until a fixpoint is reached.

By keeping track of these dependencies and allowing reanalysis of functions, we free the scheduler from having to analyze functions in a fixed order (though analyses can choose to prioritize bottom-up or top-down traversal of the call graph, to minimize reanalysis). This allows parallel computation using a central scheduler/server, and any Calypso analysis can be converted from a single core analysis to a distributed analysis with a few extra command line arguments (and some extra machines!), saving the analysis writer considerable effort in the low-level engineering details of writing a distributed analysis.

More generally, sessions and the Calypso language hide the underlying management of when information is allocated or deallocated or pushed to disk, how to index information for fast retrieval, how to order and memoize the results of the various analysis computations, and so forth. While the analysis writer loses some control in doing so, this turns attention to the actual analysis design rather than low level implementation details, and in our experience has made both writing analyses and, in particular, maintaining analyses easier than in a lower level programming language.

5. Conclusion

We have given an overview of the Saturn program analysis system, focusing on the implications of the combination of separate analysis of functions and the very precise, constraint-based, intraprocedural analysis of function bodies. We have found that analyzing functions separately and summarizing the results have the dual benefit of allowing us to use much more computationally intensive techniques for analyzing individual functions, while simultaneously making it relatively easy to scale Saturn analyses to the largest programs available to us.

References

[1] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. The Saturn program analysis system. `http://saturn.stanford.edu`, 2006.

[2] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15, New York, NY, USA, 1986. ACM Press.

[3] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. In *Formal Methods in System Design*, July 2001.

[4] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[5] R. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, October 1997.

[6] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *Proceedings of the Conference on Programming Language Design and Implementation*, page to appear, June 2007.

[7] M. Fähndrich and A. Aiken. Program analysis using mixed term and set constraints. In *Proceedings of the 4th International Symposium on Static Analysis*, pages 114–126. Springer-Verlag, 1997.

[8] B. Hackett and A. Aiken. How is aliasing used in systems software? In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 69–80, September 2006.

[9] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *Proceeding of the 28th International Conference on Software Engineering*, pages 232–241, 2006.

[10] J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *Proc. of the Conf. on Programming Language Design and Implementation*, pages 207–218, 2004.

[11] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *Proceedings of the 12th International Static Analysis Symposium*, pages 218–234. London, United Kingdom, September 2005.

[12] M. Lam, J. Whaley, B. Livshits, M. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the Conference on Principles of Database Systems*, pages 1–12, 2005.

[13] V.B. Livshits and M.S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the European Software Engineering Conference*, pages 317–326, 2003.

[14] R. Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 1998.

[15] G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction*, March 2002.

[16] J. Palsberg and M.I. Schwartzbach. Object-oriented type inference. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 146–161, 1991.

[17] T. Reps. Demand interprocedural program analysis using logic databases. In *Applications of Logic Databases*, pages 163–196, 1994.

[18] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *JLP*, 29(1–3):17–64, 1996.

[19] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1989.

[20] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 115–125, September 2005.

[21] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 351–363, January 2005.