Incremental Algorithms for Inter-procedural Analysis of Safety Properties

Christopher L. Conway[1], Kedar S. Namjoshi[2],
Dennis Dams[2], and Stephen A. Edwards[1]

[1] Department of Computer Science, Columbia University
{conway, sedwards}@cs.columbia.edu
[2] Bell Labs, Lucent Technologies
{kedar, dennis}@research.bell-labs.com

Abstract. Automaton-based static program analysis has proved to be an effective tool for bug finding. Current tools generally re-analyze a program from scratch in response to a change in the code, which can result in much duplicated effort. We present an inter-procedural algorithm that analyzes *incrementally* in response to program changes and present experiments for a null-pointer dereference analysis. It shows a substantial speed-up over re-analysis from scratch, with a manageable amount of disk space used to store information between analysis runs.

1 Introduction

Tools based on model checking with automaton specifications have been very effective at finding important bugs such as buffer overflows, memory safety violations, and violations of locking and security policies. Static analysis tools such as MC/Coverity [1] and Uno [2], and model checking tools such as SLAM [3] are based on inter-procedural algorithms for propagating dataflow information [4, 5, 6, 7]. These algorithms perform a reachability analysis that always starts from scratch. For small program changes—which often have only a localized effect on the analysis—this can be inefficient.

Our main contribution is to present the first, to our knowledge, incremental algorithms for safety analysis of recursive state machines. We demonstrate how these algorithms can be used to obtain simple—yet general and precise—incremental automaton-based program analyses. We give two such algorithms: one that operates in the forward direction from the initial states and another that operates "inside-out" from the locations of the program changes. These have different tradeoffs, as is true of forward and backward algorithms for model checking safety properties. The key to both algorithms is a data structure called a *derivation graph*, which records the analysis process. In response to a program change, the algorithms re-check derivations recorded in this graph, pruning those that have been invalidated due to the change and adding new ones. This repair process results in a new derivation graph, which is stored on disk and used for the following increment.

K. Etessami and S.K. Rajamani (Eds.): CAV 2005, LNCS 3576, pp. 449–461, 2005.

A prototype implementation of these algorithms has been made for the Orion static analyzer for `C` and `C++` programs [8]. Our measurements show significant speedup for both algorithms when compared with a non-incremental version. This comes at the expense of a manageable increase in disk usage for storing information between analysis runs. We expect our algorithms to be applicable to many current program analysis tools.

The algorithms we present are incremental forms of a standard model checking algorithm. As such, their verification result is identical to that of the original algorithm. The implementation is part of a static analysis tool that checks an abstraction of `C` or `C++` code. Thus, there is some imprecision in its results: the analysis may report false errors and miss real ones. However, the incremental algorithms produce reports with the same precision as the non-incremental algorithm.

Incremental model checking may have benefits beyond speeding up analysis. One direction is to trade the speed gain from incremental analysis for higher precision in order to reduce the number of false errors reported. Another direction is to integrate a fine-grained incremental model checker into a program development environment, so that program errors are caught immediately, as has been suggested for testing [9]. A third direction is to use an incremental model checker to enable correct-by-construction development, as suggested by Dijkstra [10]. In this scenario, instead of applying model checking after a program is written, an incremental model checker can maintain and update a proof of correctness during program development. Our work is only a first step towards realizing the full potential of these possibilities.

Experimental data and full proofs of theorems are available in an expanded version of this paper [11].

1.1 An Example

The input to the basic algorithm is a program, described by a collection of control-flow graphs (CFGs), and a checking automaton. The nodes of a CFG represent control locations, while edges are labeled either with simple assignment statements, (side-effect free) assertions, or function calls. In the model checking view, the (possibly non-deterministic) checking automaton "runs" over matched call-return paths in this collection of CFGs, flagging a potential program error whenever the current run enters an error state.

The basic model checking algorithm works by building, on-the-fly, a "synchronous product" graph of the collective CFGs with the automaton. At a function call edge, this product is constructed by consulting a summary cache of entry-exit automaton state pairs for the function. Using this cache has two consequences: it prevents infinite looping when following recursive calls and it exploits the hierarchical function call structure, so that function code is not unnecessarily re-examined.

The key to the incremental version of the algorithm is to observe that the process of forming the synchronous product can be recorded as a derivation

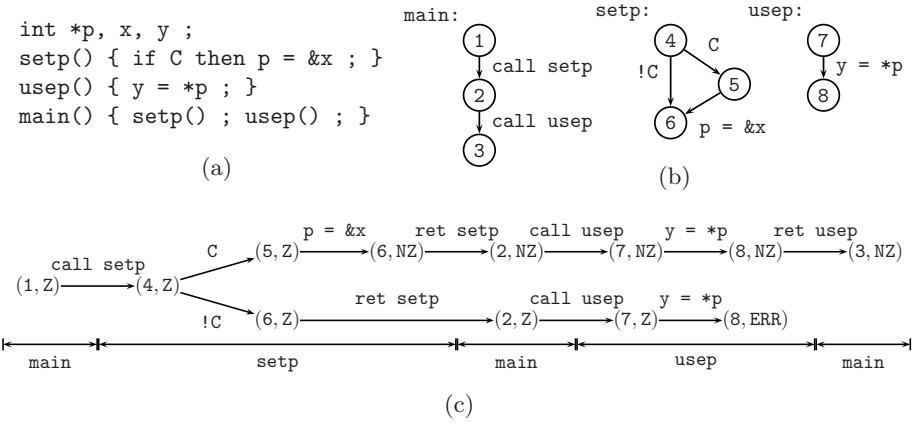

Fig. 1. (a) An example program, (b) its function CFGs, and (c) the derivation graph

graph. After a small change to the CFGs, it is likely that most of the process of forming the synchronous product is a repetition of the earlier effort. By storing the previous graph, this repetitive calculation can be avoided by checking those portions that may have been affected by the change, updating derivations only when necessary.

To illustrate these ideas, consider the program in Fig. 1(a). The correctness property we are interested in is whether the global pointer `p` is initialized to a non-null value before being dereferenced. A simple automaton (not shown) to check for violations of this property has three states: `Z`, indicating `p` may be null; `NZ`, indicating `p` is not null; and the error state `ERR` indicating `p` is dereferenced when it may be null.

Figures 1(b) and 1(c) show, respectively, the CFGs for this program and the resulting derivation graph (in this case a tree). Each derivation graph node is the combination of a CFG node and an automaton state. If condition `C` holds on entry to `setp` (the upper branch from the state $(4, Z)$ in `setp`), the function returns to `main` with the automaton state `NZ`, and execution proceeds normally to termination. If `C` does not hold (the lower branch), `setp` returns to `main` with the automaton state `Z`. On the statement "`y = *p`", the automaton moves to the state `ERR` and an error is reported in `usep`.

The incremental algorithm operates on the derivation graph data structure. Besides this graph, its input consists of additions, deletions, and modifications to CFG edges. The basic idea is simple: inspect each derivation step to determine whether it is affected by a change; if so, remove the derivation and re-check the graph from the affected point until a previously explored state is encountered.

For our example, consider the revision obtained by replacing the body of `setp()` by "`x++; p = &x;`". The new CFGs are shown in Fig. 2(a). Figure 2(b) shows the incremental effect on the derivation graph. The removal of the `if` statement has the effect of removing the conditional branch edges (dashed) from the graph, making the previous error state unreachable. The addition of `x++`

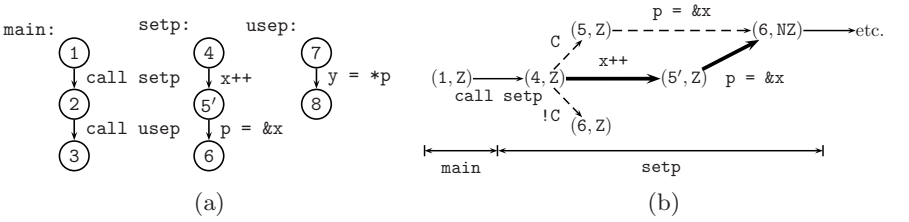

Fig. 2. (a) The revised CFGs and (b) a portion of the incremental derivation graph

has the effect of adding the state $(\mathtt{5}', \mathtt{Z})$ and two edges (bold) to the graph. After processing these edges, we get to state $(\mathtt{6}, \mathtt{NZ})$, which is identical to the corresponding state in the previous analysis. At this point, we should be able to terminate the analysis. This is a simplified picture: our algorithms actually operate somewhat differently. In particular, the backward algorithm will also inspect the derivation graph in `main`, but not that for `usep`.

2 The Full Analysis Algorithm

A program is given as a set $\mathcal{F}$ of functions, with a distinguished initial function, *main*. Each function is represented by a CFG, which is a tuple (N, Σ, E). Here, N is a finite set of *control locations* containing the distinguished locations $\downarrow$ (entry) and $\uparrow$ (exit); Σ is a set of *(simple) program statements* (assignments and assertions); and E is the set of *edges*. Let Σ' be Σ together with call statements $\{call(f) \mid f \in \mathcal{F}\}$. E is a subset of $(N \backslash \{\uparrow\}) \times \Sigma' \times N$. We require that there are no calls to functions outside $\mathcal{F}$. For simplicity of exposition, we do not represent function call arguments and return values, or variables and their scoping rules. The implementation takes each of these features into consideration.

Next we define the executions of a program. A *position* is a pair (f, n), where f is a function and n is a node in (the CFG for) f. A *(global) program state* is a sequence $(f_1, n_1) \cdots (f_k, n_k)$ of positions, representing a point during execution where control resides at position (f_k, n_k) and $(f_1, n_1) \cdots (f_{k-1}, n_{k-1})$ is the stack of return locations that is in effect at this point. We define a labeled transition system on program states, as follows.

1. $(f_1, n_1) \cdots (f_k, n_k) \xrightarrow{a} (f_1, n_1) \cdots (f_k, n'_k)$ iff (n_k, a, n'_k) is an edge in f_k and a is not a call
2. $(f_1, n_1) \cdots (f_k, n_k) \rightarrow (f_1, n_1) \cdots (f_k, n'_k)(f', \downarrow)$ iff $(n_k, call(f'), n'_k)$ is an edge in f_k
3. $(f_1, n_1) \cdots (f_{k-1}, n_{k-1})(f_k, \uparrow) \rightarrow (f_1, n_1) \cdots (f_{k-1}, n_{k-1})$

An *execution* is a finite path in this transition system that begins with the program state $(main, \downarrow)$, consisting of just the initial position. Such an execution generates a *trace* consisting of the sequence of labels (which are program statements) along it. Note that this is the definition of a recursive state machine [6, 7], restricted to the case of finite executions.

```
ADD-TO-WORKSET(c)
1  if c is not marked then
2    workset ← workset ∪ {c}
3    mark c

FOLLOW-EDGE(c, e)
1  // c=(f,n,r,q), e=(n,a,n')
2  if a = call(f') then
3    // use summaries; do book-keeping
4    ADD-TO-WORKSET((f', ↓, q, q))
5    Add c to call-sites(f')
6    for q' : ⟨q, q'⟩ ∈ summary(f') do
7      ADD-TO-WORKSET((f, n', r, q'))
8  else
9    // follow automaton transition
10   for q' : (q, a, q') ∈ Δ do
11     ADD-TO-WORKSET((f, n', r, q'))

STEP(c = (f, n, r, q))
1  if q ∈ F then
2    REPORT-ERROR(c)
3  if n =↑ then
4    // add a summary pair
5    Add ⟨r, q⟩ to summary(f)
6    workset ← workset ∪ call-sites(f)
7  else
8    // follow a CFG edge
9    for e ∈ edges(n) do
10     FOLLOW-EDGE(c, e)

ANALYZE
1  workset ← {(main, ↓, q, q) | q ∈ Q̂}
2  while workset ≠ ∅ do
3    Remove some c ∈ workset
4    STEP(c)
```

Fig. 3. Pseudo-code for the Full Algorithm

Analysis properties are represented by (non-deterministic, error detecting) automata with Σ as input alphabet. An analysis automaton is given by a tuple $(Q, \hat{Q}, \Delta, F)$, where Q is a set of *(automaton) states*, $\hat{Q} \subseteq Q$ is a set of *initial states*, $\Delta \subseteq Q \times \Sigma \times Q$, is a *transition relation*, and $F \subseteq Q$ is a set of *rejecting states*. A *run* of the automaton on a trace is defined in the standard way. A *rejecting run* is a run that includes a rejecting state. Note that in this simplified presentation, the set Σ of program statements does not include function calls and returns, and hence the automata cannot refer to them. In the implementation, transitions that represent function calls and returns (rules 2 and 3 above) carry special labels, and the error detecting automaton can react to them by changing its state, e.g. to perform checks of the arguments passed to a function, or the value returned by it.

We emphasize that an automaton operates on the *syntax* of the program; the relationship with the semantics is up to the automaton writer. For instance, one might define an under-approximate automaton, so that any error reported by the automaton check is a real program error, but it might not catch all real errors. It is more common to define an over-approximate automaton, so that errors reported are not necessarily real ones, but the checked property holds if the automaton does not find any errors.

The pseudo-code for the from-scratch analysis algorithm (Full) is shown in Fig. 3. It keeps *global configurations* in a work-set; each configuration is a tuple (f, n, r, q), where (f, n) is a position and r, q are automaton states. The presence of such a configuration in the work-set indicates that it is possible for a run of the automaton to reach position (f, n) in automaton state q as a result of entering f with automaton state r (the "root" state). In addition, the algorithm keeps a set of summaries for each function, which are entry-exit automaton state pairs,

and a set of known call-sites, which are configurations from which the function is called. ANALYZE repeatedly chooses a configuration from the work-set and calls STEP to generate its successors. In STEP, if the automaton is in an error state, a potential error is reported. (In an implementation, the REPORT-ERROR procedure may also do additional work to check if the error is semantically possible.)

Much of the work is done in the FOLLOW-EDGE procedure. For a non-call statement, the procedure follows the automaton transition relation (Line 10). For a function call, the procedure looks up the summary table to determine successor states (Line 6). If there is no available summary, registering the current configuration in *call-sites*(f') and creating a new entry configuration for f' ensures that a summary entry will be created later, at which point this configuration is re-examined (Line 6 in STEP). We assume that visited configurations are kept in a suitable data structure (e.g., a hash-table).

Theorem 1. *The Full algorithm reports an error at a configuration* (f, n, r, q)*, for some* r, q*, if and only if there is a program execution ending at a position* (f, n)*, labeled with trace* t*, such that the automaton has a rejecting run on* t*.*

3 A First Incremental Algorithm: IncrFwd

Input: A textual program change can be reflected in the CFGs as the addition, deletion, or modification of control-flow edges. It can also result in the redefinition of the number and types of variables. Our incremental algorithms expect as input CFG changes, and repair the derivation graph accordingly. Changes to types and variables correspond to automaton modifications. The algorithm can be easily modified for the situation where the property—and not the program—changes, since we are maintaining their joint (i.e., product) derivation graph.
Data Structure: The incremental algorithm records a derivation relation on configurations. This is done in the procedure FOLLOW-EDGE: whenever a new configuration of the form (f, n', r, q') is added after processing a configuration (f, n, r, q) and an edge a, a derivation edge $(f, n, r, q) \vdash_a (f, n', r, q')$ is recorded. This results in a labeled and directed derivation graph. Notice that the derivation graph can be viewed also as a tableau proof that justifies either the presence or absence of reachable error states.

Given as input a set of changes to the CFGs and a derivation graph, the incremental algorithm first processes all the modifications, then the deletions, and finally the additions. This order avoids excess work where new configurations are added only to be retracted later due to CFG deletions.
Modifications: For an edge $e = (n, a, n')$ modified to $e' = (n, b, n')$ in function f, if each derivation of the form $(f, n, r, q) \vdash_a (f, n', r, q')$ holds also for the new statement b—which is checked by code similar to that in FOLLOW-EDGE— there is no need to adjust the derivation graph. Otherwise, the modification is handled as the deletion of edge e and the addition of e'.
Additions: For a new edge $e = (n, a, n')$ in the CFG of f, FOLLOW-EDGE is applied to all configurations of the form $c = (f, n, r, q)$, for some r, q, that are

```
ADD-TO-WORKSET(c)
1  if c is not marked then
2    workset ← workset ∪{c}
3    mark c

CHECK-EDGE(e = c ⊢_a c')
1  // c = (f,n,r,q), c' = (f,n',r,q')
2  if (n, a, n') is a deleted edge then
3    skip
4  elseif a = call(f') then
5    // use stored summaries
6    ADD-TO-WORKSET((f', ↓, q, q))
7    Add c to call-sites(f')
8    if ⟨q, q'⟩ is marked in f' then
9      ADD-TO-WORKSET(c'); mark e
10 else
11   ADD-TO-WORKSET(c'); mark e

CHECK-STEP(c = (f, n, r, q))
1  if n =↑ then
2    Mark the summary ⟨r, q⟩ in f
3    workset ← workset ∪ call-sites(f)
4  else
5    for each deriv. edge e from c do
6      CHECK-EDGE(e)

CHECK-DERIVATIONS(Fns)
1  for f ∈ Fns do
2    Unmark f's configs, edges,
       summaries, and call sites that
       originate in Fns
3  workset ← EXT-INITS(Fns)
4  while workset ≠ ∅ do
5    Choose and remove c ∈ workset
6    CHECK-STEP(c)
7  Remove unmarked elements
```

Fig. 4. The IncrFwd algorithm for Deletions

present in the current graph. Consequently, any newly generated configurations are processed as in the full algorithm.

Deletions: Deletion is the non-trivial case. Informally, the idea is to check all of the recorded derivation steps, disconnecting those that are based on deleted edges. The forward-traversing deletion algorithm (IncrFwd) is shown in Fig. 4. The entry point is the procedure CHECK-DERIVATIONS, which is called with the full set of functions, $\mathcal{F}$. The auxiliary function EXT-INITS(F) returns the set of entry configurations for functions in F that arise from a call outside F. The initial configurations for *main* are considered to have external call sites. This gives a checking version of the full analysis algorithm. Checking an existing derivation graph can be expected to be faster than regenerating it from scratch with the full algorithm. The savings can be quite significant if the automaton transitions Δ are computed on-the-fly—notice that the algorithm does not re-compute Δ. The similarity between the Full and IncrFwd algorithms can be formalized in the following theorem.

Theorem 2. *The derivation graph resulting from the IncrFwd algorithm is the same as the graph generated by the Full analysis algorithm on the modified CFGs.*

4 A Second Incremental Algorithm: IncrBack

The IncrFwd algorithm checks derivations in a forward traversal. This might result in unnecessary work: if only function g is modified, functions that are not on any call path that includes g are not affected, and do not need to be

```
RETRACE(c = (f, n, r, q))
1  if c is not marked then
2    return false
3  elseif f = main then
4    return true
5  else
6    return (∃c' = (f', n', r', r):
         c' ∈ call-sites(f) ∧ RETRACE(c'))
```

```
INCR-BACK()
1  // bottom-up repair
2  for each SCC C (in reverse
       topological order) do
3    if AFFECTED(C) then
4      CHECK-DERIVATIONS(C)
5  // remove unreachable errors
6  for each error configuration c do
7    if (not RETRACE(c)) then
8      unmark c
```

Fig. 5. The IncrBack algorithm for Deletions

checked. Moreover, if the change to g does not affect its summary information, even its callers do not need to be checked. Such situations can be detected with an "inside-out" algorithm, based on the maximal strongly connected component (SCC) decomposition of the function call graph. (A non-trivial, maximal SCC in the call graph represents a set of mutually recursive functions.)

The effect of a CFG edge deletion from a function f propagates both upward and downward in the call graph. Since some summary pairs for f may no longer be valid, derivations in f's callers might be invalidated. In the other direction, for a function called by f, some of its entry configurations might now be unreachable.

The SCC-based algorithm (IncrBack) is shown in Fig. 5. It works bottom-up on the SCC decomposition, checking first the lowest (in topological order) SCC that is affected. The function AFFECTED(C) checks whether a function in C is modified, or whether summaries for any external function called from C have been invalidated. For each SCC C, one can inductively assume that summaries for functions below C are valid. Hence, it is only necessary to examine functions in C. This is done by the same CHECK-DERIVATIONS procedure as in Fig. 4, only now applied to a single SCC instead of the full program. Note that CHECK-DERIVATIONS initially invalidates summaries in C that cannot be justified by calls outside C.

This process can result in over-approximate reachability information. Consider a scenario where f calls g. Now suppose that f is modified. The algorithm repairs derivations in f, but does not touch g. However, derivations in f representing calls to g might have been deleted, making corresponding entry configurations for g unreachable. To avoid reporting spurious errors resulting from this over-approximation, the (nondeterministic) RETRACE procedure re-determines reachability for all error configurations.

Theorem 3. *The derivation graph after the IncrBack algorithm is an over-approximation of the graph generated by the full analysis algorithm on the modified CFGs, but has the same set of error configurations.*

5 Complexity and Optimality

The non-incremental algorithm takes time and space linear in the product of the size of the automaton and the size of the collective control-flow graphs. Algorithms with better bounds have been developed [6, 7], but these are based on knowing in advance the number of exit configurations of a function; this is impossible for an on-the-fly state exploration algorithm.

From their similarity to the non-incremental algorithm, it follows that the incremental algorithms cannot do more work than the non-incremental one, so they have the same worst-case bound. However, worst-case bounds are not particularly appropriate, since incremental algorithms try to optimize for the common case. Ramalingam and Reps [12, 13] propose to analyze performance in terms of a quantity $||\delta||$, which represents the difference in reachability after a change. They show that any "local" incremental algorithm like ours has worst-case inputs where the work cannot be bounded by a function of $||\delta||$ alone. At present, the precise complexity of incremental reachability remains an open question [14].

6 Implementation and Experiments

We have implemented the Full, IncrFwd, and IncrBack algorithms in the Orion static analyzer. In the implementation, we take a function as the unit of change. This is done for a number of reasons. It is quite difficult, without additional machinery (such as an incremental parser), to identify changes of finer granularity. It also fits well into the normal program development process. Furthermore, functions scale well as a unit of modification—as the size of a program increases, the relative size of individual functions decreases. In the case of large programs, attempting to identify changes at the CFG or parse tree level may not lead to significant gains.

We present data on five open source applications: sendmail, the model checker spin, spin:tl (spin's temporal logic utility), guievict (an X-Windows process migration tool) and rocks (a reliable sockets utility). We perform an interprocedural program analysis from a single entry function, checking whether global pointers are set before being dereferenced. For sendmail and spin, this analysis is run for a small subset of the program's global pointers in order to reduce the time necessary for the experiments. We simulate the incremental development of code as follows. For each function f in the program, we run the incremental analysis on the program with f removed (i.e., replaced with an empty stub). Then we insert f and run the incremental analysis. The time taken for this incremental analysis is compared with a full analysis of the program with f. We thus have one analysis run for each function in the program; each run represents an incremental analysis for the modification of a single function (in this case, replacing an empty stub with the actual body of the function).

This experiment exercises both the addition and deletion algorithms: the modification of a function is equivalent to deleting and reinserting a call edge at each of its call sites; if the function summary changes, derivations based on the

Table 1. Experimental results

	Lines of code	Reachable functions	No. ptrs analyzed	Full analysis time (s)	Average speedup IncrFwd	Average speedup IncrBack	Incr. data (KB)
sendmail	47,651	336	3	33.75	1.6	**8.6**	73.72
spin	16,540	348	6	24.91	1.3	**10.1**	91.61
spin:tl	2,569	93	2	1.09	1.3	**8.0**	7.01
guievict	4,545	115	1	0.60	1.4	**5.9**	3.54
rocks	4,619	134	1	0.66	1.3	**4.3**	4.36

old summary are deleted and new derivations are generated based on the newly available, now-accurate summary.

The experimental results are shown in Table 1. The overall average speedup for IncrBack is 8.2; the average for IncrFwd is 1.4. IncrFwd improves on Full essentially by "caching" state data between analysis runs. This caching behavior is able to provide modest performance increases, but the average-case performance of the algorithm is unbounded in $||\delta||$. IncrBack is able to improve on the performance of IncrFwd by skipping large portions of the derivation graph, when possible, and using the call graph structure of the program to minimize its workload. This intuition is confirmed by the experimental results.

To better illustrate the performance characteristics of IncrBack, Fig. 6 plots the speedups for each analysis run in terms of percentiles. For each percentage x, we plot the minimum speedup for the best-performing $x\%$ of analysis runs. For example, 50% of analysis runs overall showed a speedup of at least 7.5 (i.e., 7.5 is the median speedup). The legend shows the number of analysis runs (i.e., the number of (statically) reachable functions) for each benchmark. The data on the horizontal axis is plotted in uniform intervals of length $100/N$ for each benchmark. The plateaus evident in the plots for spin, spin:tl, and rocks represent clustering of data values (possibly due to rounding) rather than a sparsity of data points.

There was quite a bit of variation between benchmarks: over 50% of runs on spin showed a speedup of at least 12.4, while the maximum speedup for rocks was only 5.5. It is likely that larger programs will exhibit higher speedups in general. We observed no strong correlation between the the speedup for a function and its size, its depth in the call graph, or its number of callers and callees (see [11]).

The tests we describe here are conservative in the sense that we only analyze functions that are reachable from a single distinguished entry function. Preliminary tests on all of the entry functions in guievict show an average speedup of 11.3 (instead of 5.9)—since many functions are unconnected in the call graph to the modified function, the full algorithm does much unnecessary work.

These tests concentrate on changes that are quite small with respect to the total size of the program. We hypothesize that changes on the order of a single function are a reasonable model of how the analysis would be applied in a development scenario. However, we have also run tests in which 10-50% of the functions in the program are modified in each increment. In these tests, IncrBack

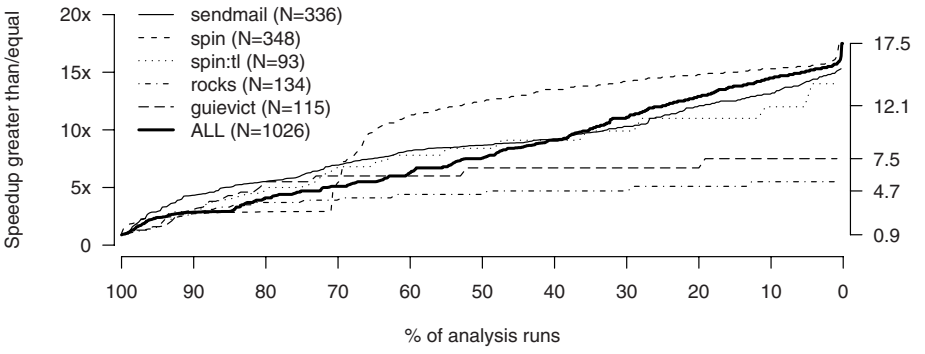

Fig. 6. Distribution of speedups for IncrBack, with quantiles for all at right

showed a more modest average speedup of 2.5, with larger speedups for smaller incremental changes.

Table 1 also shows the size of the incremental data stored after the re-run of the incremental analysis, on the complete program. This data may be written to disk, taking time proportional to the size. In an interactive setting, this time can be considered irrelevant: the user can begin inspecting errors while the tool performs I/O.

7 Related Work and Conclusions

The approach of automaton-based model checking of push-down systems [15, 6, 7] has contributed algorithms for program analysis that are conceptually simple and powerful. We have developed incremental versions of these algorithms and shown that this approach leads to incremental dataflow algorithms that are simple yet precise and general. The algorithms lend themselves to simple implementations showing excellent experimental results: a factor of 8.2 on average for IncrBack, at the cost of a manageable overhead in storage. To the best of our knowledge, the algorithms we propose are the first for inter-procedural, automaton-based static analysis.

There is a strong similarity between the behavior of our checking procedure and tracing methods for garbage collection [16] (cf. [17]). A key difference is the pushdown nature of the derivation graphs, which has no analogy in garbage collection.

Incremental data flow analysis has been studied extensively. Existing data flow algorithms are not directly applicable to model checking, because they either compute less precise answers than their from-scratch counterparts; are applicable only to restricted classes of graphs, such as reducible flow-graphs; or concern specific analyses, such as points-to analysis [18, 19, 20] (cf. the excellent survey by Ramalingam and Reps [21]). Sittampalam et al. [22] suggest an approach to incremental analysis tied to program transformation (cf. [23]). Since analyses

are specified on the abstract syntax tree, the technique only applies to idealized Pascal-like languages. The SCC decomposition has been applied to the flow graphs of individual functions to speed up analysis by Horwitz et al. [24] and Marlowe and Ryder [25].

Perhaps most closely related to our work is the research on the incremental evaluation of logic programs by Saha and Ramakrishnan [26, 27]. Their support graphs play the same role as derivation graphs in our work. The techniques used for updating these graphs are reminiscent of Doyle's truth maintenance system [28]. While inter-procedural analysis is readily encoded as a logic program (cf. [6]), we suspect that it may be hard to recover optimizations such as the SCC-based method. Previous algorithms for incremental model checking [29, 30] do not handle either program hierarchy or recursion, working instead with a flat state space. Some tools (e.g., Uno [2], MOPS [31]) pre-compute per-file information; however, the interprocedural analysis is still conducted from scratch.

The Orion tool in which the algorithms are implemented is aimed at producing error reports with a low false-positives ratio. In this context, it seems especially attractive to devote the time gained by incrementalization towards a further improvement of this ratio, especially for inter-procedural analysis.

Acknowledgements. Thanks to Nils Klarlund for many helpful comments on a draft of the paper. We also thank D. Saha and C.R. Ramakrishnan for kindly sending us a draft of their paper [27], Sape Mullender for a useful discussion, and Rupak Majumdar for his comments. This work was supported in part by NSF grant CCR-0341658. Stephen Edwards and his group are supported by an NSF CAREER award, a grant from Intel corporation, an award from the SRC, and from New York State's NYSTAR program.

References

1. Hallem, S., Chelf, B., Xie, Y., Engler, D.: A system and language for building system-specific, static analyses. In: PLDI, Berlin, Germany (2002) 69–82
2. Holzmann, G.: Static source code checking for user-defined properties. In: Integrated Design and Process Technology (IDPT), Pasadena, CA (2002)
3. Ball, T., Rajamani, S.K.: The SLAM toolkit. In: CAV, Paris, France (2001) 260–264
4. Reps, T., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: POPL, San Francisco, CA (1995) 49–61
5. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: CAV, Paris, France (2001) 324–336
6. Alur, R., Etessami, K., Yannakakis, M.: Analysis of recursive state machines. In: CAV, Paris, France (2001) 207–220
7. Benedikt, M., Godefroid, P., Reps, T.: Model checking of unrestricted hierarchical state machines. In: ICALP, Crete, Greece (2001) 652–666
8. Dams, D., Namjoshi, K.S.: Orion: High-precision static error analysis for C and C++ programs. Technical report, Bell Labs (2003)

9. Saff, D., Ernst, M.D.: An experimental evaluation of continuous testing during development. In: ISSTA, Boston, MA (2004) 76–85
10. Dijkstra, E.: Guarded commands, nondeterminacy, and formal derivation of programs. Communications of the ACM **18** (1975)
11. Conway, C.L., Namjoshi, K.S., Dams, D., Edwards, S.A.: Incremental algorithms for inter-procedural analysis of safety properties. Technical Report CUCS-018-05, Columbia University, New York, NY (2005)
12. Reps, T.: Optimal-time incremental semantic analysis for syntax-directed editors. In: POPL, Albuquerque, NM (1982) 169–176
13. Ramalingam, G., Reps, T.: On the computational complexity of dynamic graph problems. Theoretical Computer Science **158** (1996) 233–277
14. Hesse, W.: The dynamic complexity of transitive closure is in DynTC^0. Theoretical Computer Science **3** (2003) 473–485
15. Schmidt, D., Steffen, B.: Program analysis *as* model checking of abstract interpretations. In: SAS, Pisa, Italy (1998) 351–380
16. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine. Communications of the ACM **3** (1960) 184–195
17. Wilson, P.: Uniprocessor garbage collection techniques. In: International Workshop on Memory Management (IWMM), Saint-Malo, France (1992) 1–42
18. Yur, J.S., Ryder, B., Landi, W., Stocks, P.: Incremental analysis of side effects for C software systems. In: ICSE, Los Angeles, CA (1997) 422–432
19. Yur, J.S., Ryder, B., Landi, W.: An incremental flow- and context-sensitive pointer aliasing analysis. In: ICSE, Boston, MA (1999) 442–451
20. Vivien, F., Rinard, M.: Incrementalized pointer and escape analysis. In: PLDI, Snowbird, Utah (2001) 69–82
21. Ramalingam, G., Reps, T.: A categorized bibliography on incremental computation. In: POPL, Charleston, SC (1993) 502–510
22. Sittampalam, G., de Moor, O., Larsen, K.: Incremental execution of transformation specifications. In: POPL, Venice, Italy (2004) 26–38
23. Liu, Y.A., Stoller, S.D., Teitelbaum, T.: Static caching for incremental computation. ACM Trans. on Programming Languages and Systems **20** (1998) 546–585
24. Horwitz, S., Demers, A., Teitelbaum, T.: An efficient general iterative algorithm for dataflow analysis. Acta Informatica **24** (1987) 6790–694
25. Ryder, B., Marlowe, T.: An efficient hybrid algorithm for incremental data flow analysis. In: POPL, San Francisco, CA (1990) 184–196
26. Saha, D., Ramakrishnan, C.: Incremental evaluation of tabled logic programs. In: ICLP, Mumbai, India (2003) 392–406
27. Saha, D., Ramakrishnan, C.: Incremental and demand driven points to analysis using logic programming. Provided by authors (2004)
28. Doyle, J.: A truth maintenance system. Artificial Intelligence **12** (1979) 231–272
29. Sokolsky, O., Smolka, S.: Incremental model checking in the modal mu-calculus. In: CAV, Stanford, CA (1994) 351–363
30. Henzinger, T., Jhala, R., Majumdar, R., Sanvido, M.: Extreme model checking. In: Verification: Theory and Practice, Sicily, Italy (2003) 332–358
31. Chen, H., Wagner, D.: MOPS: an infrastructure for examining security properties of software. In: CCS, Washington, DC (2002) 235–244