Fine-Grained Caching of Verification Results

K. Rustan M. Leino[1](✉) and Valentin Wüstholz[2](✉)

[1] Microsoft Research, Redmond, WA, USA
leino@microsoft.com
[2] Department of Computer Science, ETH Zurich, Zurich, Switzerland
valentin.wuestholz@inf.ethz.ch

Abstract. Developing provably correct programs is an incremental process that often involves a series of interactions with a program verifier. To increase the responsiveness of the program verifier during such interactions, we designed a system for fine-grained caching of verification results. The caching system uses the program's call graph and control-flow graph to focus the verification effort on just the parts of the program that were affected by the user's most recent modifications. The novelty lies in how the original program is instrumented with cached information to avoid unnecessary work for the verifier. The system has been implemented in the Boogie verification engine, which allows it to be used by different verification front ends that target the intermediate verification language Boogie; we present one such application in the integrated development environment for the Dafny programming language. The paper describes the architecture and algorithms of the caching system and reports on how much it improves the performance of the verifier in practice.

1 Introduction

Making formal program verification useful in practice requires not only automated logical theories and formal programming-language semantics, but also—inescapably—a human understanding of why the program under verification might actually be correct. This understanding is often gained by trial and error, debugging verification attempts to discover and correct errors in programs and specifications and to figure out crucial inductive invariants. To support this important trial and error process, it is essential that the integrated development environment (IDE) provides rapid feedback to the user.

In this paper, we enhance the IDE for the specification-aware programming language Dafny [20] by adding fine-grained caching of results from earlier runs of the verifier. The effect of this caching is to reduce the time from user keystrokes in the editor to the reporting of verification errors that are gathered in the background. In some cases, this lag time can now be around a second for examples where it previously may have taken tens of seconds for the verifier to repeat the checking of proof obligations that were not affected by the change.

D. Kroening and C.S. Păsăreanu (Eds.): CAV 2015, Part I, LNCS 9206, pp. 380–397, 2015.
DOI: 10.1007/978-3-319-21690-4_22

These improvements rely on a basic caching technique that tracks dependencies using the program's call graph to avoid re-verification of methods that were not affected by the most recent change to the program. Our fine-grained caching takes this a step futher. It is motivated by the fact that when a proof obligation is not automatically verified, a user tends to spend human focus and editing in one small area of the program. Often, this area can be in one branch of a method, so if the tool can rapidly re-verify just what has changed, the user can make progress more quickly. Our fine-grained caching thus makes use of the program's control-flow graph.

Like other verifiers, the Dafny verifier generates proof obligations by translating Dafny to an intermediate verification language (IVL), namely Boogie [2,21]. We designed our fine-grained caching to operate at the level of the IVL, which makes it possible for other Boogie front ends to make use of the new functionality. Our novel caching approach compares the current *snapshot* of a Boogie program with a previously verified snapshot. It then instruments the current snapshot to adjust the proof obligations accordingly. Finally, it passes the instrumented Boogie program to the underlying satisfiability-modulo-theories (SMT) solver in the usual way. Our implementation is available as part of the Boogie and Dafny open source projects.

In Sect. 2, we explain a motivating example in more detail. Sect. 3 gives background on the architecture of the Dafny verifier and describes the basic, coarse-grained caching based on the program's call graph. We describe our fine-grained caching in Sect. 4 and evaluate how both techniques improve the performance of the verifier in Sect. 5.

2 Motivating Example

Let us consider some typical steps in the interactive process of developing a verifiably correct program, indicating where our caching improvements play a role. Figure 1 shows an incomplete attempt at specifying and implementing the Dutch Flag algorithm, which sorts an array of colors.

The program gives rise to several proof obligations, following the rules of Hoare logic. The loop invariants are checked when control flow first reaches the loop. The loop body with its three branches is checked to decrease a termination metric (here provided by the tool: the absolute difference between `w` and `b`) and to maintain the loop invariants. The postcondition of the method is checked to follow from the loop invariants and the negation of the guard (without further inspection of the loop body). For every call to method `Sort` in the rest of the program, the method's precondition is checked and its postcondition is assumed.

In addition, all statements and expressions, including those in specifications, are verified to be well-formed. For example, for the assignment that swaps two array elements in the loop body (line 18), the well-formedness checks ensure that the array is not `null`, that the indices are within bounds of the array, that the method is allowed to modify the heap at these locations, and that the parallel assignment does not attempt to assign different values to the same heap location.

```
datatype Color = Red | White | Blue

predicate Ordered(c: Color, d: Color) { c = Red ∨ d = Blue }

method Sort(a: array<Color>)
  requires a ≠ null
  modifies a
  ensures forall i,j • 0 ≤ i < j < a.Length ⟹ Ordered(a[i], a[j])
{
  var r, w, b := 0, 0, a.Length;
  while w ≠ b
    invariant 0 ≤ r ≤ w ≤ b ≤ a.Length
    invariant forall i • 0 ≤ i < r ⟹ a[i] = Red
    invariant forall i • r ≤ i < w ⟹ a[i] = White
    invariant forall i • b ≤ i < a.Length ⟹ a[i] = Blue
  {
    match a[w]
      case Red ⇒
        a[r], a[w] := a[w], a[r]; r := r + 1;
      case White ⇒
        w := w + 1;
      case Blue ⇒
        b := b - 1;
  }
}
```

Fig. 1. Incomplete attempt at implementing the Dutch Flag algorithm. As written, the program contains a specification omission, a specification error, and two coding errors. As the program is edited, our fine-grained caching of verification results enables a more responsive user experience by avoiding re-verification of unaffected proof obligations.

To provide *design-time* feedback to the user, the Dafny IDE automatically runs the verifier in the background as the program is being edited. This allows the verifier to assist the user in ways that more closely resemble those of a background spell checker. Given the program in Fig. 1, the Dafny verifier will report three errors.

The first error message points out that the method body may not establish the postcondition. Selecting this error in the Dafny IDE brings up the verification debugger [18], which readily points out the possibility that the array contains two `White` values. To fix the error, we add a disjunct `c = d` to the definition of predicate `Ordered`. Instead of expecting the user to re-run the verifier manually, the Dafny IDE will do so automatically. To speed up this process, the basic caching technique will already avoid some unnecessary work by using the call graph: only methods that depend on the predicate `Ordered` will be re-verified, which includes the body of `Sort` and, since the postcondition of `Sort` mentions the predicate, all callers of `Sort`. Caller dependencies get lower scheduling priority, since they are likely to be further away from the user's current focus of attention. However, we can hope for something even better: the maintenance of the loop invariant in `Sort` need not be re-verified, but only the fact that the loop invariant and the negation of the guard establish the postcondition. Our fine-grained caching technique makes this possible.

The second error message points out that the loop may fail to terminate. Selecting the error shows a trace through the `Red` branch of the `match` statement, and we realize that this branch also needs to increment `w`. As we make that change, the tool re-verifies only the loop body, whereas it would have re-verified the entire method with just the basic caching technique.

The third error message points out that the last loop invariant is not maintained by the `Blue` branch. It is fixed by swapping `a[w]` and `a[b]` after the update to `b`. After doing so, the re-verification proceeds as for the second error.

Finally, it may become necessary to strengthen `Sort`'s postcondition while verifying some caller—it omits the fact that the final array's elements are a permutation of the initial array's. If only the basic caching was used, the addition of such a postcondition would cause both `Sort` and all of its callers to be re-verified. By using the fine-grained caching, the body of `Sort` is re-verified to check only the new postcondition (which in this case will require adding the postcondition also as a loop invariant). For callers, the situation is even better: since the change of `Sort`'s specification only strengthens the postcondition, proof obligations in callers that succeeded before the change are not re-verified.

The performance improvements that we just gave a taste of have the effect of focusing the verifier's attention on those parts of the program that the user is currently, perhaps by trial and error, editing. The result is a user experience with significantly improved response times. In our simple example program, the time to re-verify the entire program is about 0.25 seconds, so caching is not crucial. However, when programs have more methods, contain more control paths, and involve more complicated predicates, verification times can easily reach tens of seconds. In such cases, our fine-grained caching can let the user gain insight from the verification tool instead of just becoming increasingly frustrated and eventually giving up all hopes of ever applying formal verification techniques.

3 Verification Architecture and Basic Caching

In this section, we describe the role of the intermediate verification language Boogie and the basic caching technique that the fine-grained caching builds on. We have presented an informal overview of the basic caching technique in a workshop paper describing different novel features of the Dafny IDE [22].

3.1 Architecture

Like many other verifiers, such as Spec# [3] and VCC [8], Dafny uses the Boogie [2] intermediate verification language to express proof obligations to be discharged by the Boogie verification engine using an SMT solver, such as Z3 [10]. The language constructs of the source language are translated into more primitive constructs of

Boogie, including variables, axioms, and procedures. For example, a Dafny *method* is translated to several Boogie constructs: (1) a *procedure (declaration)* that captures the specification of the method, (2) a *procedure implementation* that captures the method body and checks that it adheres to the method specification, and (3) a second procedure implementation that captures the well-formedness conditions for the method specification [19]. As another example, a Dafny *function* is translated to a corresponding Boogie *function* and a procedure implementation that captures the function's well-formedness conditions. Boogie functions are given meaning by *axioms*, but to simplify our presentation, we omit some details of the translation of Dafny functions.

Boogie supports a modular verification approach by verifying procedure implementations individually. More precisely, calls in procedure implementations are reasoned about only in terms of their specification (i.e., the corresponding procedure declaration). Consequently, a change to a program often does not invalidate verification results obtained for independent program entities. In particular, a change in a given procedure implementation does not invalidate verification results of other procedure implementations, and a change in a procedure's specification may invalidate verification results only of its callees and of the corresponding procedure implementation.

3.2 Basic Caching

While the Boogie pipeline accepts a single program, obtains verification results, and then reports them, the basic caching mechanism turns Boogie into more of a verification service: it accepts a stream of programs, each of which we refer to as a *snapshot*.

The basic caching approach exploits the modular structure of Boogie programs by determining which program entities have been changed *directly* in the latest program snapshot and which other program entities are *indirectly* affected by those changes. To determine direct changes, Boogie relies on the client front end (Dafny in our case) to provide an *entity checksum* for each function, procedure, and procedure implementation. For example, the Boogie program in Fig. 2 shows entity checksums provided by a front end to Boogie via the `:checksum` custom attribute. In our implementation, Dafny computes them as a hash of those parts of the Dafny abstract syntax tree that are used to generate the corresponding Boogie program entities. This makes checksums insensitive to certain textual changes, such as ones that concern comments or whitespace.

To determine indirect changes, Boogie computes *dependency checksums* for all functions, procedures, and procedure implementations based on their own entity checksum and the dependency checksums of entities they depend on directly (e.g., callees). These checksums allow the basic caching to reuse verification results for an entity if its dependency checksum is unchanged in the latest snapshot.

For example, when computing dependency checksums from entity checksums in Fig. 2, Boogie takes into account that both implementations depend on the

```
procedure {:checksum "727"} abs(a: int) returns (r: int)
  ensures 0 ≤ r;

implementation {:checksum "733"} abs(a: int) returns (r: int)
{ r := 0; }

implementation {:checksum "739"} main()
{ var x: int; call x := abs(-585); assert x = 585; }
```

Fig. 2. Boogie program that shows how a front end uses custom attributes on declarations to assign entity checksums, which can be computed in front-end specific ways.

procedure declaration of `abs` (implementation `abs` needs to adhere to its procedure declaration and `main` contains a call to `abs`). Consequently, a change that only affects the entity checksum of *procedure* `abs` (e.g., to strengthen the postcondition) will prevent Boogie from returning cached verification results for both implementations. However, a change that only affects the entity checksum of *implementation* `abs` (e.g., to return the *actual* absolute value) will allow Boogie to return cached verification results for implementation `main`.

Figure 3 gives an architectural overview of the caching system. In terms of it, the basic caching works as follows. First, Boogie computes dependency checksums for all entities in a given program snapshot. Then, for each procedure implementation P, the cache is consulted. If the cache contains the dependency checksum for P, branch (0) is taken and the cached verification results are reported immediately. Otherwise, branch (1) is taken and the procedure implementation is verified as usual by the Boogie pipeline. Our fine-grained caching may also choose branch (2), as we explain in Sect. 4.

3.3 Prioritizing Procedure Implementations Using Checksums

Besides using them for determining which procedure implementations do not need to be re-verified, we use the checksums for determining the order in which the others should be verified. Ideally, procedure implementations that are more directly related to the user's latest changes are given higher priority, since these most likely correspond to the ones the user cares about most and wants feedback on most quickly. The checksums provide a metric for achieving this by defining four priority levels for procedure implementations:

- low (unlike the entity checksum, the dependency checksum in the cache is different from the current one): Only dependencies of the implementation changed.
- medium (entity checksum in the cache is different from the current one): The implementation itself changed.
- high (no cache entry was found): The implementation was added recently.
- highest (both the entity and the dependency checksum is the same as the one in the cache): The implementation was not affected by the change and a cache lookup is sufficient for reporting verification results to the user *immediately*, instead of waiting for other implementations to be verified.

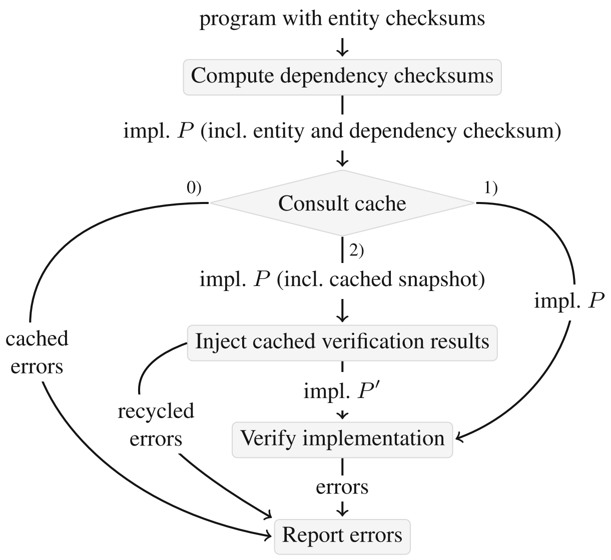
program with entity checksums
Compute dependency checksums
impl. P (incl. entity and dependency checksum)
0)
Consult cache
1)
2)
impl. P (incl. cached snapshot)
impl. P
cached errors
Inject cached verification results
impl. P'
recycled errors
Verify implementation
errors
Report errors

Fig. 3. Overview of the verification process for procedure implementations. Boxes correspond to components and arrows illustrate data flow. The caching component produces three possible outputs: 0) cached errors in case the entity and dependency checksums are unchanged, 1) the implementation P in case it is not contained in the cache, or 2) the implementation P and the cached snapshot in case either the entity or the dependency checksum have changed. Cached snapshots are used to inject verification results into the implementation and to identify errors that can be recycled.

4 Fine-Grained Caching

Basic caching can determine which procedure implementations in a new snapshot do not need to be re-verified at all, but it does not track enough information to allow us to reuse verification results for parts of an implementation. In this section, we present an extension of the basic caching that reuses verification results in fine-grained ways. In particular, our extension avoids re-verification of checks that were not affected by the most recent change and it recycles errors that are still present in the current snapshot.

Before giving our full algorithm, we sketch how it works in two common scenarios we want to address: when an isolated part of a procedure implementation (e.g., one of two branches or a loop body) has been changed, and when the specification of a procedure has been changed. We proceed by example, starting from the program in Fig. 4. Running Boogie on this program results in two errors: a failure to establish the postcondition on line 2 and an assertion violation on line 7. To fix the postcondition error in the program in Fig. 4, the user might add an explicit else branch on line 10 and insert statement `r := x`. This is an instance of the common change-in-isolated-part scenario. In particular,

```
0  procedure gcd(x, y: int) returns (r: int)
1    requires 0 < x ∧ 0 < y;
2    ensures 0 ≤ r;
3
4  implementation gcd(x, y: int) returns (r: int) {
5    if (x < y) {
6      call r := gcd(x, y - x);
7      assert 1 ≤ r;
8    } else if (y < x) {
9      call r := gcd(x - y, y);
10   }
11   assert 0 < x + y;
12 }
```

Fig. 4. Incomplete attempt at implementing a Boogie procedure for computing the greatest common denominator. Boogie reports a postcondition violation for the implementation and an assertion violation on line 7.

the change has no effect on the assertion on line 7, and thus we would hope to be able to cache and recycle the error.

4.1 Fine-Grained Dependency Tracking Using Statement Checksums

To cache and reuse verification results at this fine granularity, we need to know what each statement depends on. To determine this, we compute a *statement checksum* for every statement from a hash of its pretty-printed representation and—to keep the overhead small—the statement checksums of *all* statements that precede it in the control flow (as opposed to ones that actually affect it). If a statement contains a function call in some subexpression, then the statement depends on the callee's definition and we include the callee's dependency checksum when computing the statement checksum.

The computation of statement checksums occurs after the Boogie program has undergone some simplifying transformations. For example, loops have been transformed using loop invariants and back-edges of loops have been cut [4]; thus, the computation of statement checksums does not involve any fixpoint computation. As another example, the checks for postconditions have been made explicit as `assert` statements at the end of the implementation body and the preconditions of procedure implementations have been transformed into `assume` statements at the beginning of the implementation body; thus, these statements are taken into account for computing the statement checksums. In contrast to an `assert` statement, which instructs the verifier to check if a condition holds at the given program point, an `assume` statement instructs the verifier to blindly assume a condition to hold at the given program point.

After the simplifications from above, there are only two kinds of statements that lead to checks: assertions and calls (precondition of callee). We will refer to them as *checked statements*. We introduce a cache that associates statement checksums of such statements in a given implementation with verification results. Before verifying a new snapshot, we compute statement checksums for the new snapshot and then instrument the snapshot by consulting this cache.

Let us describe this in more detail using our example. We will refer to the program in Fig. 4 as Snapshot 0 and the program resulting from adding the else branch and assignment on line 10 as Snapshot 1. After verifying Snapshot 0, the cache will have entries for the statement checksums of the following checked statements: the failing assertion on line 7, the succeeding precondition checks for the calls on lines 6 and 9, the succeeding assertion on line 11, and the failing check of the postcondition from line 2. The statement checksums for the first three checked statements (on lines 6, 7, and 9) in Snapshot 1 are the same as in Snapshot 0. Since the cache tells us the verification results for these, we report the cached error immediately and we add `assume` statements for the checked condition before these checked statements in Snapshot 1. The statement checksums of the fourth and fifth checked statement are different in Snapshot 1, since they are affected by the modification of line 10. Since the new checksums are not found in the cache, the statements are not rewritten. As a result, Boogie needs to only verify those checks. Indeed, Boogie is now able to prove both and it updates the cache accordingly. With reference to Fig. 3, what we have just described takes place along branch (2) after the basic cache has been consulted.

4.2 Injecting Explicit Assumptions and Partially Verified Checks

To fix the failing assertion on line 7 in Fig. 4, the user might now decide to strengthen the postcondition of the procedure by changing it to `1` $\leq$ `r`. This is an instance of the common change-in-specification scenario. In particular, since the change involves a strengthened postcondition, we would hope to avoid re-verifying any previously succeeding checks downstream of call sites.

We will refer to the program resulting from the user's change as Snapshot 2. After Boogie computes the statement checksums, only the statement checksum for the assertion of the postcondition will be different from the ones in the cached snapshot. However, since the dependency checksums of the callee changed for both calls, we introduce an *explicit assumption* [7] after each call to capture the condition assumed at this point in the cached snapshot. We do so by introducing an *assumption variable* for each such call that is initialized to `true` and is only assigned to once (here, after the corresponding call) using a statement of the form `a` := `a` $\wedge$ P, where `a` is the assumption variable and P is a boolean condition. The variable allows us to later refer to an assumption that was made at a specific program point; e.g., to mark a check that was not failing in the corresponding cached snapshot as *partially verified* under a conjunction of assumption variables.

To illustrate, consider the rewrite of Snapshot 2 in Fig. 5. At this stage, the precondition is assumed explicitly on line 2 and the postcondition is asserted explicitly on line 15 as described earlier. On line 0, we introduce one assumption variable for each call to a procedure with a different dependency checksum, and these are initialized to `true` on line 1. The call on line 5 gets to assume the new postcondition of `gcd`. If that call happens to return in a state that was allowed by the previous postcondition (`0` $\leq$ `r`), then assumption variable `a0` will remain true after the update on line 6. But if the call returns in a state that does not satisfy the previously assumed postcondition, then `a0` will be set to `false`. In our example, since the postcondition of the callee is strengthened, the explicit assumption `0` $\leq$ `r` will always evaluate to true. Indeed, this works particularly well when postconditions are not weakened, but, depending on the calling context, it may also simplify the verification otherwise. For instance, it would work for a call where the state is constrained such that for this particular call site the previous postcondition holds after the call, even though the new postcondition is indeed weaker.

Next, we inject assumptions into the program about checked statements that are found to be non-failing in the cached snapshot based on their statement checksum. More precisely, for each statement with checked condition P whose statement checksum is in the cache and that was non-failing in the cached snapshot, we inject an assumption $A \implies P$, where A is the conjunction of all assumption variables. Intuitively, this tells the verifier to skip this check if all assumption variables are true. Otherwise, the verifier will perform the check since a state was reached for which it has not already been verified in the cached snapshot. We say that the check has been marked as *partially verified.* As an optimization, we include in A only those assumption variables whose update statement definition can reach this use; we refer to these as *relevant* assumption variables. Figure 5 shows the assumptions being introduced on lines 4, 9, and 13, preceding the precondition checks and the assert statement, thus marking these checks as partially verified. Note that the assertion on line 7 is not marked as partially verified, since it is a failing assertion in Snapshot 1. Since the assumption variables remain true, the partially verified checks in effect become fully verified in this example. Note that the verifier may discover that only some partially verified checks are in effect fully verified depending on the state at those checks. For instance, this may happen if the state after some call was not always allowed by the callee's previous postcondition, but some partially verified checks after that call are in a conditional branch where the branching condition constrains the state such that all states are allowed by the previous postcondition *there.*

4.3 Algorithm for Injecting Cached Verification Results

In this subsection, we present our algorithm for injecting cached verification results in procedure implementations of medium or low priority, for which no limit on the number of reported errors was hit when verifying the cached implementation. At this point, most existing Boogie transformations have been applied to the implementation as described earlier (e.g., eliminating loops using loop invariants and adding explicit assertions for procedure postconditions).

As a first step, we compute statement checksums for all statements in an implementation as defined earlier. As a second step, we insert explicit assumptions for calls if the dependency checksum of the callee has changed in the current snapshot. More precisely, for each call, we distinguish between three different cases, in order:

```
 0  var {:assumption} a0, a1: bool;
 1  a0, a1 := true, true;
 2  assume 0 < x ∧ 0 < y;  // precondition
 3  if (x < y) {
 4    assume (true) ⟹ (0 < x ∧ 0 < y - x);
 5    call r := gcd(x, y - x);
 6    a0 := a0 ∧ (0 ≤ r);
 7    assert 1 ≤ r;
 8  } else if (y < x) {
 9    assume (true) ⟹ (0 < x - y ∧ 0 < y);
10    call r := gcd(x - y, y);
11    a1 := a1 ∧ (0 ≤ r);
12  } else { r := x; }
13  assume (a0 ∧ a1) ⟹ (0 < x + y);
14  assert 0 < x + y;
15  assert 0 ≤ r;  // postcondition
```

Fig. 5. Body of the procedure implementation for Snapshot 2 after injecting cached verification results (underlined). The instrumented program contains two explicit assumptions [7] on lines 6 and 11 derived from the postcondition of the cached callee procedure. Also, all checks that did not result in errors in the cached snapshot have been marked as partially verified by introducing `assume` statements on lines 7, 9, and 13.

1. Dependency checksum of callee is the same as in the cached snapshot: We do not need to do anything since the asserted precondition and the assumed postcondition are the same as in the cached snapshot.
2. All functions that the callee transitively depended on in the cached snapshot are still defined and unchanged in the current snapshot: Before the call, we add the statement `assume` $? \implies P$, where ? is a placeholder that will be filled in during the final step of the algorithm and P is the precondition of the callee in the cached snapshot. This may allow us to reuse the fact that the precondition of a call has been verified in the cached snapshot. To simplify the presentation, we will only later determine if the precondition has indeed been verified and under which condition. Since the dependency checksum of the callee is different from the one in the cached snapshot, we additionally introduce an explicit assumption to capture the condition that was assumed after the call in the cached snapshot. This condition depends on the callee's *modifies clause* (which lists the global variables that the callee is allowed to modify) and its postcondition. To capture the former, let V be the set of global variables that were added to the callee's modifies clause since the cached snapshot. We now add `ov` := `v` for each global variable `v` in this set V before the call, where `ov` is a fresh, local variable. This allows us to express the explicit assumption by adding the statement `a` := `a` $\wedge$ (Q $\wedge$ M) after the call, where `a` is a fresh assumption variable, Q is the postcondition of the callee in the cached snapshot and M contains a conjunct `ov == v` for each global variable `v` in the set V . Note that M does not depend on global variables that

were removed from the callee's modifies clause since the cached snapshot; the statements after the call have already been verified for all possible values of such variables.

3. Otherwise: Since we cannot easily express the pre- and postcondition of the callee in the cached snapshot, we need to be conservative. We therefore do not add any assumption about the precondition and we add the statement `a` $:=$ `a` $\wedge$ `false` after the call, where `a` is a fresh assumption variable.

As a third step, we transform each checked statements with the checked condition P to express cached verification results. We distinguish four cases, in order:

1. Some relevant assumption variable is definitely false when performing constant propagation: We do not do anything, since we cannot determine under which condition the check may have been verified.
2. There was an error for this check in the cached implementation and there are no relevant assumption variables: Since it has previously resulted in an error under identical conditions, we add the statement `assume` P before and report the error immediately to avoid unnecessary work.
3. There was no error for this check in the cached implementation: Since it has been verified previously, we add the statement `assume` $A \Longrightarrow P$ before, where A is the conjunction of all relevant assumption variables. If there are any such assumption variables, we say that the check has been marked as *partially* verified; otherwise, we say that it has been marked as *fully* verified.
4. Otherwise: We do not do anything. For instance, this may happen if we cannot determine that we have seen the same check in the cached snapshot.

As a last step, we replace the placeholder `?` in each statement `assume ?` $\Longrightarrow$ P with the conjunction of all relevant assumption variables, if none of the relevant assumption variables are definitely false and there was no error for the corresponding call in the cached implementation. Otherwise, we drop the statement.

Optimization for Explicit Assumptions Within Loops. By default, loop bodies are verified modularly in Boogie. That is, on entry to a loop body, all variables that are modified within the body are "havocked" by assigning a non-deterministic value and the invariant is assumed. After the loop body, only the invariant remains to be checked. For this reason, an assumption (e.g., as a result of a procedure call) that was made in the loop body when verifying the cached snapshot was neither used for verifying statements after the loop (provided there is no `break` statement in the loop) nor for verifying statements within the loop that precede the assignment to the corresponding assumption variable. To reproduce this behavior for the current snapshot, it is safe not to havoc assumption variables that would usually be havocked in this case. By doing so, such assumption variables usually remain true at that point unless the corresponding loop has previously been unrolled a number of times.

5 Evaluation

To evaluate the effectiveness of our caching techniques *in practice*, we recorded eight verification sessions during expert use of the Dafny IDE for regular development tasks. Those sessions were not scripted and therefore cover real workloads that such a tool faces when it is being used by a user to develop provably correct software. The sessions span a wide range of activities (including extension, maintenance, and refactoring) that are encountered when developing programs of several hundred lines. Sessions consist of up to 255 individual program snapshots (see Fig. 6) since the Dafny IDE automatically verifies the program as the user is editing it. To make this a pleasant experience for the user, the responsiveness of the tool is of paramount importance.

Figure 6 clearly shows that this user experience could not be achieved without caching. The basic caching alone decreases the running times of the verifier tremendously (more than an order of magnitude for many sessions) and complementing it with fine-grained caching decreases them even more. This confirms the positive feedback that we received from users of the Dafny IDE, including members of the Ironclad project at Microsoft Research, whose codebase includes more than 30'000 lines of Dafny code [15]. Interestingly, caching turned out to have a more significant effect on the responsiveness of the tool than parallelization of verification tasks in Boogie using multiple SMT solver instances.

Figure 7 sheds more light on why the basic caching is so effective by showing the priorities of the procedure implementations that are sent to the verifier for each snapshot in session 5: most of the procedure implementations do not need to be re-verified at all and only two implementations (originating from a single Dafny method) need to be verified for most snapshots. This data looks very similar for the other sessions and demonstrates that the basic caching benefits significantly from the modular verification approach in Dafny. Besides this, we can see that there are occasional spikes with procedure implementations of low priority. For example, snapshot 2 consists of a change to a function that may affect all callers. In fact, due to the way that functions are handled, all *transitive* callers are affected, which is not the case for procedures. While in this case the basic caching needs to re-verify 11 procedure implementations from scratch, the fine-grained caching is able to mark 400 out of 971 checked statements in Boogie as fully verified. This reduces the running time from 28 s to 14 s and at the same time avoids a timeout (by default, 10 s per procedure implementation) for one of those procedure implementations.

Overall, Fig. 6 shows that the fine-grained caching performs even better than the basic caching for all sessions (42 % faster for session 3 and on average 17 % faster compared to the basic caching). For session 7, there is no significant speedup even though the fine-grained caching is able to mark a large number of checks as verified. It seems that, in this case, most of the time is spent on verifying a single check (e.g., the postcondition of the edited method) that could not be marked as verified. Such cases can come up occasionally since the times that are needed for verifying different checks are usually not distributed uniformly.

Session	Snapshots	Time (in Seconds)			Number of Timeouts		
		NC	BC	FGC	NC	BC	FGC
0	70	4’395.3	315.3	277.5	58	3	1
1	13	758.5	88.2	74.8	11	4	2
2	59	3’648.0	220.2	206.1	83	5	4
3	254	13’977.6	1’734.7	1’008.7	2	1	0
4	255	6’698.6	533.8	499.8	16	6	5
5	27	1’956.0	785.7	519.7	0	0	0
6	29	106.9	33.3	27.3	0	0	0
7	7	765.5	20.5	20.0	0	0	0

Fig. 6. Comparison of three configurations for verifying eight recorded IDE sessions: no caching (**NC**), basic caching (**BC**) and fine-grained caching (**FGC**). The second column shows the number of program snapshots per session. The next three columns show the running times for each configuration and the rightmost three columns show the number of timed-out procedure implementations for each configuration.

Besides increasing responsiveness, caching helps in reducing the number of procedure implementations that fail to verify due to timeouts (see Fig. 6). Again, the basic caching avoids the majority of timeouts and the fine-grained caching avoids even more of them (between 17 % and 100 % less), which is not obvious given our program transformations. This additional reduction over the basic caching is due to the fact that Boogie is able to focus on fewer unverified or partially verified checks.

To provide a better indication of how much the fine-grained caching is able to reduce the verification effort, Fig. 8 shows the number of checked statements for each snapshot in session 5 that were transformed when injecting cached verification results. This demonstrates that for many snapshots, more than half of the checks can be marked as fully verified or errors from the cached snapshot can be recycled (two errors each for snapshots 5 and 6 and one error each for snapshots 7 and 8). At an early development stage, fewer checks were marked as verified since statement checksums changed more often. It turned out that small changes in a Dafny program could result in significant changes to the

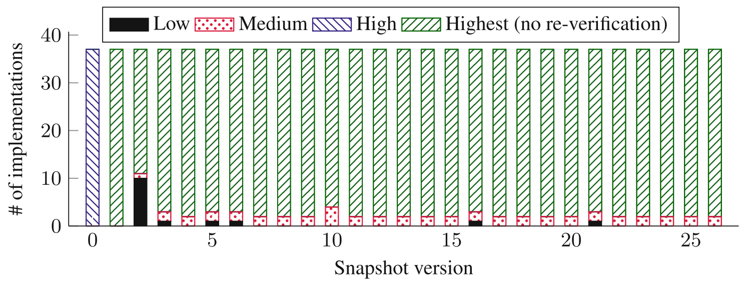

Fig. 7. Priorities of procedure implementations for session 5. The bars show the number of procedure implementations of a given priority for each snapshot version. Most implementations are assigned the highest priority and do not need to be re-verified.

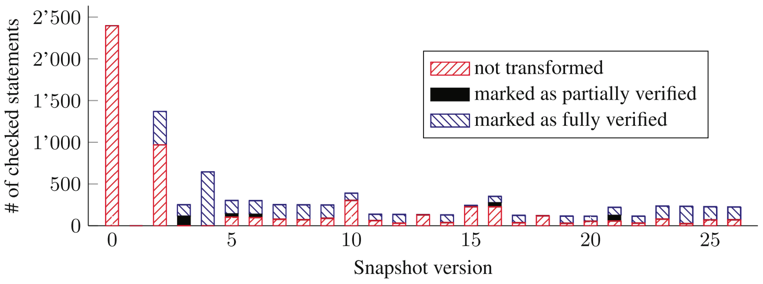

Fig. 8. Transformed checked statements in session 5. The bars show the number of checked statements for each snapshot version that are marked as fully verified, partially verified, or not transformed at all. Additionally, a number of errors are recycled: two errors each for snapshots 5 and 6 and one error each for snapshots 7 and 8.

corresponding Boogie program due to the way in which names (e.g., of auxiliary variables) were generated. After taking this into account during the translation of Dafny into Boogie, performance improved significantly.

6 Related Work

Caching is a widely used technique for reusing information that was computed in the past. More specifically, there are several existing approaches for reusing results from previous runs of static analyzers, model checkers, program verifiers, and automatic test-case generation tools. Clousot [12], a static analyzer for .NET, uses caching to retrieve the results of previous runs of its cloud-based analysis service [1]. Unlike our fine-grained caching, it only reuses such results if a method itself did not change *and* if the specifications of all its callees did not change. Clousot also supports "verification modulo versions" [23], which uses conditions inferred for a previous version of a program to only report new errors for the current version. The Why3 verification platform uses checksums to maintain program proofs in the form of *proof sessions* as the platform evolves (e.g., by generating different proof obligations). In particular, it matches goals from the existing proof with new goals using both checksums and *goal shapes*, a heuristic similarity measure. Maintenance of proofs is particularly important for interactive proof assistants since proofs are largely constructed by users and, ideally, do not need to be changed once they are completed. Such work has been done for the KIV [24] and KeY [17] tools. Grigore and Moskal [14] have worked on such techniques for proofs that were generated by SMT solvers to verify programs using ESC/Java2.

There are several approaches for reusing information that was computed when running a non-modular tool on an earlier revision of a program. In the area of model checking, such information can consist of summaries computed using Craig interpolation [25], *derivation graphs* that record analysis progress [9], or parts of the reachable, abstract state space [16]; even the precision of the analysis that was sufficient for analyzing an earlier program revision may be used later [5]. Work on incremental compositional dynamic test generation [13] presents techniques for determining if function summaries that were obtained for an earlier version of a program can be safely reused when performing symbolic execution on the current version of the program.

Regression verification [26] is another area that developed techniques for reusing information that was collected during runs of a tool on earlier versions of a program. Unlike in our approach, the goal is to check if the behavior of the latest version of a program is equivalent to the one of an earlier version, much like in regression testing.

In spirit, our caching scheme is an instance of a truth maintenance system [11]. However, the mechanisms used are quite different. For example, a truth maintenance system records justifications for each fact, whereas our caching scheme tracks snapshots of the programs that give rise to proof obligations, not the proofs of the proof obligations themselves.

7 Conclusions and Future Work

We have presented two effective techniques for using cached verification results to improve the responsiveness and performance of the Dafny IDE. Both techniques are crucial for providing *design-time feedback* at every keystroke to users of the IDE, much like background spell checkers. The key novelties of our technique are its use of checksums for determining which parts of a program are affected by a change and how a program is instrumented with cached information to focus the verification effort. In particular, we use explicit assumptions to express the conditions under which we can reuse cached verification results. We have designed our technique to work on the level of an intermediate verification language. This makes it immediately usable for other verifiers that use the Boogie verification engine (e.g., VCC [8] or AutoProof [27]) and should make possible to adopt by other intermediate verification languages, such as Why3 [6].

As future work, we would like to make the existing caching more fine-grained in cases where assumptions in the program (e.g., resulting from user-provided `assume` statements, preconditions, and user-provided or inferred loop invariants) are affected by a change. We believe that—much like for procedure calls—we can use explicit assumptions to capture assumptions that were made in the cached snapshot, and thereby mark more checks as partially verified. We would also like to look into techniques, such as slicing, for determining if certain partially verified checks could be marked as fully verified by identifying the explicit assumptions they depend on more precisely.

Acknowledgments. We are grateful to the users of the Dafny IDE—notably, Nada Amin, Maria Christakis, Arjun Narayan, and Bryan Parno—for providing feedback on its caching system. We thank Maria for comments on a draft of this paper and the reviewers for their constructive comments.

References

1. Barnett, M., Bouaziz, M., Fähndrich, M., Logozzo, F.:. A case for static analyzers in the cloud. In: Workshop on Bytecode Semantics, Verification, Analysis, and Transformation (Bytecode 2013) (2013)
2. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
3. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. Commun. ACM **54**(6), 81–91 (2011)
4. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Workshop on Program Analysis for Software Tools and Engineering (PASTE), pp. 82–87. ACM (2005)
5. Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: ESEC/FSE, pp. 389–399. ACM (2013)
6. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Why3: shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, pp. 53–64 (2011)
7. Christakis, M., Müller, P., Wüstholz, V.: Collaborative verification and testing with explicit assumptions. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 132–146. Springer, Heidelberg (2012)
8. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
9. Conway, C.L., Namjoshi, K.S., Dams, D.R., Edwards, S.A.: Incremental algorithms for inter-procedural analysis of safety properties. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 449–461. Springer, Heidelberg (2005)
10. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
11. Doyle, J.: A truth maintenance system. Artif. Intell. **12**(3), 231–272 (1979)
12. Fähndrich, M., Logozzo, F.: Static contract checking with abstract interpretation. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 10–30. Springer, Heidelberg (2011)
13. Godefroid, P., Lahiri, S.K., Rubio-González, C.: Statically validating must summaries for incremental compositional dynamic test generation. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 112–128. Springer, Heidelberg (2011)
14. Grigore, R., Moskal, M.: Edit and verify. In: Workshop on First-Order Theorem Proving (FTP) (2007)
15. Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D., Zill, B.: Ironclad apps: End-to-end security via automated full-system verification. In: OSDI, USENIX Association, pp. 165–181 (2014)
16. Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.A.: Extreme model checking. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 332–358. Springer, Heidelberg (2004)
17. Klebanov, V.: Extending the reach and power of deductive program verification. Ph.D. thesis. Department of Computer Science, Universität Koblenz-Landau (2009)

18. Le Goues, C., Leino, K.R.M., Moskal, M.: The Boogie verification debugger (tool paper). In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 407–414. Springer, Heidelberg (2011)
19. Leino, K.R.M.: Specification and verification of object-oriented software. In: Engineering Methods and Tools for Software Safety and Security, Volume 22 of NATO Science for Peace and Security Series D: Information and Communication Security, Summer School Marktoberdorf 2008 Lecture Notes, pp. 231–266. IOS Press (2009)
20. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
21. Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: design and logical encoding. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 312–327. Springer, Heidelberg (2010)
22. Leino, K.R.M., Wüstholz, V.: The Dafny integrated development environment. In: Workshop on Formal Integrated Development Environment (F-IDE), Electronic Notes in Theoretical Computer Science, vol. 149, pp. 3–15 (2014)
23. Logozzo, F., Lahiri, S.K., Fähndrich, M., Blackshear, S.: Verification modulo versions: towards usable verification. In: PLDI, pp. 294–304. ACM (2014)
24. Reif, W., Stenzel, K.: Reuse of proofs in software verification. In: Shyamasundar, R.K. (ed.) FSTTCS 1993. LNCS, vol. 761, pp. 284–293. Springer, Heidelberg (1993)
25. Sery, O., Fedyukovich, G., Sharygina, N.: Incremental upgrade checking by means of interpolation-based function summaries. In: FMCAD, pp. 114–121. IEEE (2012)
26. Strichman, O., Godlin, B.: Regression verification - a practical way to verify programs. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 496–501. Springer, Heidelberg (2008)
27. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: AutoProof: auto-active functional verification of object-oriented programs. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 566–580. Springer, Heidelberg (2015)