Detecting Runtime Check Patterns and Applying Optimization in GVC0

Sam Estep
Carnegie Mellon University
Pittsburgh, PA, USA
estep@cmu.edu

Paulo Canelas
Carnegie Mellon University
Pittsburgh, PA, USA
pacsantos@cmu.edu

ABSTRACT

Defining a specification to verify a program is a challenging and complex task statically. Gradual verification uses ideas from gradual typing to allow developers to incrementally and progressively verify their system by evaluating programs with varying levels of specification precision. The only existing tool for gradual verification, GVC0, allows developers to introduce partial specifications that contain static verifications and runtime checks inserted. Nevertheless, introducing runtime checks increases the execution time of the system. Ideally scenario, runtime verification should have a negligible impact on the system's performance, but in practice, GVC0 introduces a significant performance hit. In this work, we analyze the runtime checks introduced by GVC0 and apply an optimization that improves the performance of programs via a combination of pointer analysis, loop-invariant code motion, and redundancy elimination. We evaluated our approach in the 100 slowest programs and our results improved the performance of these gradually verified programs by 7%.

1 INTRODUCTION

Defining a specification to statically verify a program is a challenging and complex task [6]. For instance, a strict static verification of a large or evolving codebase may be impractical or overly restrictive. Recent work in gradual typing [4, 7, 8], allowed the introduction of techniques that help developers progressively specify their system.

Gradual verification [2] uses ideas from gradual typing technique to allow developers to incrementally and progressively verify their system by evaluating programs with varying levels of specification precision. Specifications in gradual verification are either statically and dynamically verified. Statically, the verifier assumes any strenghtening of imprecise specifications, and to guarantee soundness, it generates dynamic checks when the partial specifications are optimistically strengthened.

The only existiong tool for gradual verification is GVC0 [3], which allows developers to introduce partial specifications. These partial specifications contain static verifications and runtime checks inserted by GVC0. C0 [1] a variant of C used for education, and GVC0 works by augmenting C0 with specifications for gradual verification. These specifications may appear as pre- and post-conditions in methods, as well as loop invariants of cycles and other properties defined in the code. Listing 1 shows a simple example of the header for a GVC0 program that inserts a value into a sorted list. The `sorted` predicate is defined earlier in the source

Listing 1: Signature and specification for the `list_insert` function.

```
1 struct Node *list_insert(
2   struct Node *list, int val)
3   //@ requires sorted(list);
4   //@ ensures sorted(\result);
```

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

file, and GVC0 uses the source code of `list_insert` to ensure that the `list` remains sorted after the function returns. If anything in the function body is imprecisely specified, GVC0 inserts code to check necessary facts at runtime.

Introducing runtime checks increases the execution time of the system. Ideally, runtime verification should have a negligible impact on the performance of the system. This means, that any inserted checks should be optimized as much as possible. However, in practice, these runtime checks currently make the augmented program drastically slower than the original; see fig. 1. We talked to the authors of GVC0 and learned that there are likely many opportunities for optimizations to make these runtime checks much more efficient.

In this work, we **analyze the runtime checks introduced by GVC0 and apply optimizations that improve the performance of programs**. Similarly to previous work in literature [5], the objective is to minimize the overhead of the injected runtime checks. To do this, we perform a three step approach. First, we downloaded a benchmark of program permutations that range from fully static verified to fully dynamically verified. Second, we sorted the programs according to their performance and manually inspected a subset of these programs, trying to identify possible optimizations. Finally, we designed and implemented an optimization to improve common patterns we identified, and evaluated the performance of the optimized programs.

This work focuses tries to answer the following two research questions.

RQ1. What runtime check patterns does GVC0 introduce? To answer RQ1, we executed GVC0 on the datasets presented in prior work [3, 9], manually inspected the generated code with the injected execution checks, studied the patterns in which these are introduced, and investigated how they can be optimized.

RQ2. To what extent do optimizations to runtime checks impact program performance? Regarding RQ2, we used the detected pattern to improve the generated code. We performed the optimization by developing a LLVM pass that deduplicates usages of C0's builtin **`c0_deref`** function, applying loop invariant code motion and redundancy elimination. Finally, we evaluateed the impact of the optimizations by comparing the runtime performance of the original code with and without optimizations.

The contributions of this work are described below.

(1) a conveniently-packaged version of a dataset obtained from the compilation of the GVC0 benchmark with different levels of the specification lattice;
(2) an LLVM pass that uses Loop Invariant Code Motion and Redundancy Elimination to optimize **`c0_deref`** calls, and its respective evaluation; and
(3) a `Dockerfile` and Docker image that allow the execution of GVC0 and the replication of our results.

The remainder of this report is structured as follows. Section 2 presents our methodology in detecting possible optimizations and Section 3 provides implementation details regarding the optimization performed. Section 4 describes the experimental setup used to replicate our results. Section 5 highlights the research questions this work studies and discusses the results of the proposed approach. Section 7, present our conclusions and evidence future work that can be applied to optimize gradual verification programs written in c0.

2 MANUAL ANALYSIS OF RUNTIME CHECKS

Figure 2 shows the process we used to run GVC0 on a collection of over ten thousand permutations (partially specified variants) of four base programs. All the permutations were stored in a database on one student's desktop computer, so we wrote and ran a script to download all those permutations, then packaged them as a `*.zip` file.

Then we built a JAR file for GVC0 itself, and wrote a Python script to run it in parallel across all the permutations to insert runtime

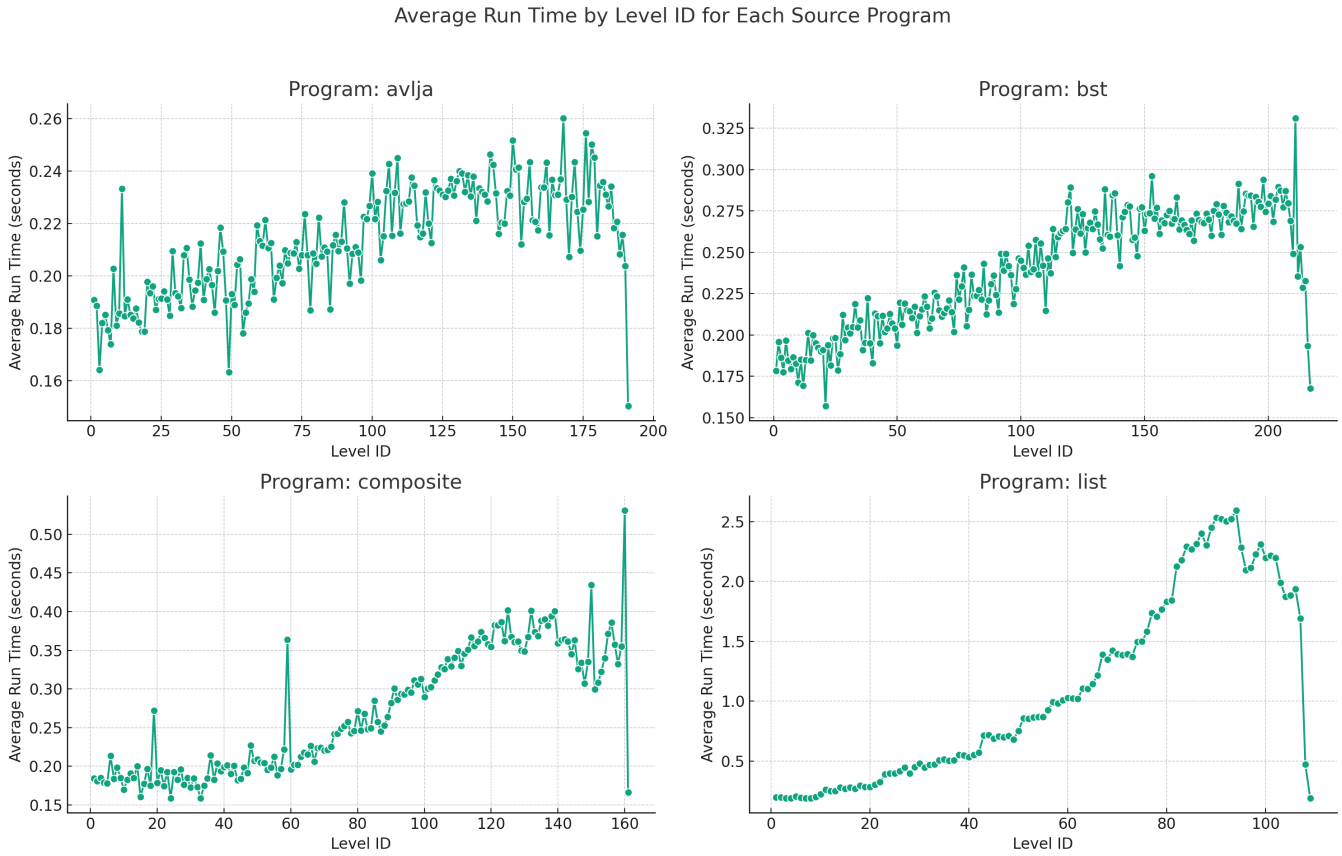

Figure 1: Performance of four GVC0 programs with varying levels of precision in their specification. The x-axis ranges from fully imprecise (level ID 0, no verification) to fully precise (maximum level ID, complete static verification).

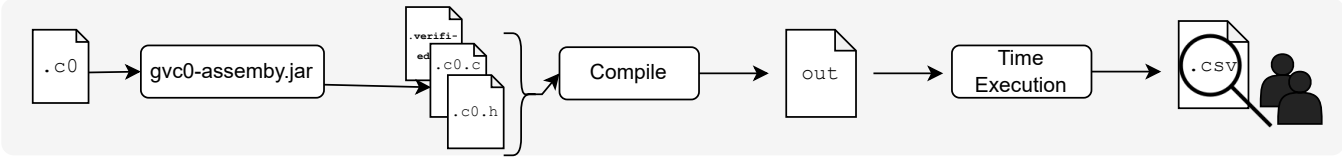

Figure 2: Outline of our initial data collection process. We initially compile a c0 program using GVC0, obtaining a verified.c0 program, a c program and its respective headers. Then, we compiled the c generated file with any possible introduced runtime checks and compile it, obtaining an executable. Given this executable we time it using a `--stress 128`, we then sort the execution times and started manually inspecting from the slowest programs.

checks for each. After that, we wrote another Python script to compile all those resulting C0 programs to C, and yet another to use LLVM to compile those with `-O3`, then another Python script to run those compiled binaries and gather the timing data shown in Figure 1.

Upon looking at the runtime checks inserted into the slowest programs, we noticed many instances of redundant pointer dereferences, such as in listing 2. We looked at the generated LLVM IR with `clang -O3`, and learned that LLVM does not deduplicate these deferences. This is because, while it looks like a dereference in the original C0 code, the C0 compiler `cc0` actually transpiles it to a function in the C code it generates.

Listing 2: The definition of the find function used by GVC0 to track ownership of heap locations.

```
1  FieldArray* find(OwnedFields* fields, int _id){
2    if(_id >= 0){
3      int index = hash(_id, fields->capacity);
4      while(fields->contents[index] != NULL){
5          if(!fields->contents[index]->deleted && fields->
               contents[index]->_id == _id){
6              return fields->contents[index];
7          }else{
8              index = (index + 1) % fields->capacity;
9          }
10       }
11     }
12   return NULL;
13 }
```

Listing 3 shows the C source code produced by the cc0 compiler from the find function in Listing 2. Each pointer dereference is replaced by a usage of the cc0_deref macro defined in cc0lib.h, which expands to a call to the c0_deref function defined in c0rt.c. LLVM doesn't know the semantics of c0_deref, so it does not optimize this code. But we do know its semantics, so we used this to write a custom LLVM pass for optimizing C0 code that uses pointers.

3 OPTIMIZATION OF C0_DEREF CALLS

In this section, we provide details w.r.t. to the optimization of the **c0_deref** calls previously identified. Listing 4 presents the pseudocode of the loop invariant code motion and redundancy elimination used in the LLVM pass. We use LLVM's LoopPass class so that our pass gets run on every loop in the program, working from inner loops first to outer loops last. The pseudocode presents six overall steps described as follows.

Step 1. We check if there a loop preheader, but note that by running LLVM's -loop-simplify pass before ours, we can guarantee it to exist. Instructions deemed relevant for code motion are later moved to this preheader.

Step 2. We append all the instructions to a set, InLoop. This variable is relevant to later check (line 23), if an SSA variable is defined inside the loop, then it cannot be moved to the preheader.

Listing 3: The definition of the find function used by GVC0 to track ownership of heap locations.

```
1  _c0_FieldArray* _c0_find(_c0_OwnedFields* _c0v_fields,
       int _c0v__id) {
2  if ((_c0v__id >= 0)) {
3  int _c0t__tmp_36 = (cc0_deref(struct _c0_OwnedFields,
       _c0v_fields))._c0_capacity;
4  int _c0t__tmp_37 = _c0_hash(_c0v__id, _c0t__tmp_36);
5  int _c0v_index = _c0t__tmp_37;
6  while (true) {
7    cc0_array(struct _c0_FieldArray*) _c0t__tmp_38 = (
         cc0_deref(struct _c0_OwnedFields, _c0v_fields)).
         _c0_contents;
8    struct _c0_FieldArray* _c0t__tmp_39 = cc0_array_sub(
         struct _c0_FieldArray*,_c0t__tmp_38,_c0v_index);
9    if ((_c0t__tmp_39 != NULL)) {
10   } else {
11     break;
12   }
13   bool _c0t__tmp_46;
14   cc0_array(struct _c0_FieldArray*) _c0t__tmp_40 = (
         cc0_deref(struct _c0_OwnedFields, _c0v_fields)).
         _c0_contents;
15   struct _c0_FieldArray* _c0t__tmp_41 = cc0_array_sub(
         struct _c0_FieldArray*,_c0t__tmp_40,_c0v_index);
16   bool _c0t__tmp_42 = (cc0_deref(struct _c0_FieldArray,
         _c0t__tmp_41))._c0_deleted;
17   if (!(_c0t__tmp_42)) {
18     cc0_array(struct _c0_FieldArray*) _c0t__tmp_43 = (
           cc0_deref(struct _c0_OwnedFields, _c0v_fields)).
           _c0_contents;
19     struct _c0_FieldArray* _c0t__tmp_44 = cc0_array_sub(
           struct _c0_FieldArray*,_c0t__tmp_43,_c0v_index);
20     int _c0t__tmp_45 = (cc0_deref(struct _c0_FieldArray,
           _c0t__tmp_44))._c0__id;
21     _c0t__tmp_46 = (_c0t__tmp_45 == _c0v__id);
22   } else {
23     _c0t__tmp_46 = false;
24   }
25   if (_c0t__tmp_46) {
26     cc0_array(struct _c0_FieldArray*) _c0t__tmp_47 = (
           cc0_deref(struct _c0_OwnedFields, _c0v_fields)).
           _c0_contents;
27     struct _c0_FieldArray* _c0t__tmp_48 = cc0_array_sub(
           struct _c0_FieldArray*,_c0t__tmp_47,_c0v_index);
28     return _c0t__tmp_48;
29   } else {
30     int _c0t__tmp_49 = (cc0_deref(struct _c0_OwnedFields,
           _c0v_fields))._c0_capacity;
31     int _c0t__tmp_50 = c0_imod((_c0v_index + 1),
           _c0t__tmp_49);
32     _c0v_index = _c0t__tmp_50;
33   }
34 }
35 }
36 return NULL;
37 }
```

Step 3. We create a map for the dereferences and the duplications encountered. Derefs is a dictionary that maps pointers to instructions for dereferencing those pointers. If another dereference is encountered, we replace all usages of the call instruction with the already existing dereference. Dups contains the list of call instructions that are duplicated and later removed.

Step 4. We perform the loop invariant code motion and deduplication. We look at each instruction to find call instructions for the c0_deref function. When encountering this instruction, we obtain the first argument, check if there it was assigned somewhere inside the loop, and if not, we check if that dereference was already performed. If the instruction was already performed, we replace the call instruction with the existing ones, otherwise we add the call instruction and the arg to the Derefs mapping.

Step 5. We remove all the duplications of the dereferences encountered.

Step 6. We move all remaining dereferences to the preheader of the loop.

Unlike the programs we encountered during assignments for the class, the C code produced by GVC0 depends on the C0 standard library, so we must perform extra steps for linking before we can apply our LLVM optimization pass. First we compile the C source code using clang, then run llvm-dis to produce LLVM IR. Then we sue opt to run our optimization on that LLVM IR, generating the optimized bytecode which we then compile to a binary for evaluation.

Listing 4: Loop Invariant Code Motion and Deduplication pseudocode implemented in the LLVM pass.

```
// Step 1. Retrieve loop's preheader
Preheader: BasicBlock := Get Preheader of loop L

if Preheader does not exist:
    return false

InLoop: set[Value];

// Step 2. Populate with instructions from loop blocks
For each BasicBlock B in blocks of loop L:
    For each Instruction I in BasicBlock B:
        Insert I into InLoop

// Step 3. Create a map for dereference operations and a vector for duplicates
Derefs: map[Value, CallInst];
Dups: vector[CallInst];

// Step 4. Process call instructions, handle duplicates and update Derefs
For each BasicBlock B in blocks of loop L:
    For each Instruction I in BasicBlock B:
        If I is a Call Instruction (CI) and calls function "c0_deref":
            Get the first argument of CI as Arg
            If Arg is not in InLoop:
                If Arg is already in Derefs:
                    Replace CI with existing Call Instruction
                    Add CI to Dups
                Else:
                    Add CI to Derefs with key Arg

// Step 5. Remove duplicate call inst. from parent blocks
For each Call Instruction CI in Dups:
    Remove CI from its parent

// Step 6. Relocate dereference calls to preheaders end
For each entry in Derefs:
    Move Call Instruction to Preheaders terminator

return true
```

4 EXPERIMENTAL SETUP

To allow the reproduction of our results, we created a Dockerfile that installs all the required dependencies by GVC0. Furthermore, we also provide the compiled recreated .c, .c0 and .c0.h corresponding to the permutations static and dynamic checks of programs.

To setup the environment, the developer is required to have Docker[1] installed, and execute the commands described as follows.

```
docker build -t gvc0
docker run --name gvc0_container gvc0
```

The image build may take several minutes to complete. We used Visual Studio Code Remote Server to access the container, develop the pass and the perform the evaluation. To reproduce our results, executing the following command with produce a .csv file with the performance for each recreated file from GVC0.

```
python evaluation.py
```

We executed our evaluation on the Docker container on Ubuntu 22.04.3 LTS running on AMD Ryzen Threadripper 2990WX 32-Core Processor with 64 GB of RAM.

5 EVALUATION

To evaluate the performance, we executed the LLVM pass over the 100 slowest sampled files from dataset, and collected the execution time before and after the optimization. Each file was executed three times to guarantee and non-determinism in the execution. Figure 3

[1] https://www.docker.com/

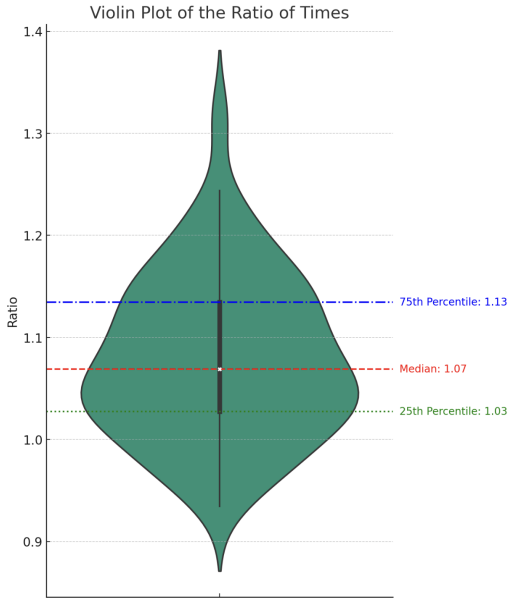

Figure 3: Violin plot of the performance improvement on the 100 slowest sampled files of the dataset. The median performance increase is of 7%.

presents the results from our evaluation. Overall, we achieved a mean improvement of performance of 8%, and a median of 7% improvement.

6 SURPRISES AND LESSONS LEARNED

It took a long time (multiple weeks) for us just to gather the example C0 programs, run GVC0 on them, and run them so we could see which ones were slowest and thus worth looking at. Once we did that, it only took about a day to identify an optimization that we could write. It then took a few days to successfully set up an environment in which we could run both LLVM and GVC0, since we had trouble running the former on Apple Silicon. After that, it only took about a day to actually write the optimization. It was interesting to see that just building and running things was the hardest part of the project.

7 CONCLUSION & FUTURE WORK

We presented a quantitative analysis of the performance impact GVC0 has when inserting runtime checks, a qualitative analysis of some patterns in these runtime checks that can be exploited to perform compiler optimizations, an implementation of a hybrid optimization using pointer analysis with loop-invariant code motion and redundancy elimination to improve these patterns, and an evaluation of that optimization on the 100 slowest programs we encountered showing that we achieve a 7% performance increase on these gradually verified programs.

Future work would dig deeper into the runtime checks inserted by GVC0 to identify more patterns to optimize. In this work we only really scratched the surface, implementing a rather generic optimization that works on any C0 program. We are guessing that a more targeted optimization for gradual verification could have much greater impacts than what we show here.

8 DISTRIBUTION OF TOTAL CREDIT

Both students contributed 50% to the final project.

REFERENCES

[1] Rob Arnold. 2010. *C0, an imperative programming language for novice computer scientists.* Ph. D. Dissertation. Master's thesis, Department of Computer Science, Carnegie Mellon University.

[2] Johannes Bader, Jonathan Aldrich, and Éric Tanter. 2018. Gradual program verification. In *Verification, Model Checking, and Abstract Interpretation: 19th International Conference*. Springer, 25–46.

[3] Jenna DiVincenzo, Ian McCormack, Hemant Gouni, Jacob Gorenburg, Mona Zhang, Conrad Zimmerman, Joshua Sunshine, Éric Tanter, and Jonathan Aldrich. 2022. Gradual C0: Symbolic Execution for Efficient Gradual Verification. arXiv:2210.02428 [cs.LO]

[4] Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting gradual typing. In *Principles of Programming Languages*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 429–442. https://doi.org/10.1145/2837614.2837670

[5] Olatunji Ruwase, Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. 2010. Decoupled lifeguards: enabling path optimizations for dynamic correctness checking tools. In *Programming Language Design and Implementation*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 25–35. https://doi.org/10.1145/1806596.1806600

[6] Todd W. Schiller and Michael D. Ernst. 2012. Reducing the barriers to writing verified specifications. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 95–112. https://doi.org/10.1145/2384616.2384624

[7] Jeremy Siek and Walid Taha. 2006. Gradual typing for functional languages. *Scheme and Functional Programming*, 11.

[8] Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *European Conference Object-Oriented Programming (Lecture Notes in Computer Science, Vol. 4609)*, Erik Ernst (Ed.). Springer, 2–27. https://doi.org/10.1007/978-3-540-73589-2_2

[9] Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. 2020. Gradual verification of recursive heap data structures. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28.