Using Specialized Procedures and Specification-Based Analysis to Reduce the Runtime Costs of Modularity

Mark T. Vandevoorde* and John V. Guttag*

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA

ABSTRACT

Managing tradeoffs between program structure and program efficiency is one of the most difficult problems facing software engineers. Decomposing programs into abstractions simplifies the construction and maintenance of software and results in fewer errors. However, the introduction of these abstractions often introduces significant inefficiencies.

This paper describes a strategy for eliminating many of these inefficiencies. It is based upon providing alternative implementations of the same abstraction, and using information contained in formal specifications to allow a compiler to choose the appropriate one. The strategy has been implemented in a prototype compiler that incorporates theorem proving technology.

Keywords: Program Modularity, Software Interfaces, Formal Specifications, Compilers, Program Optimization.

1 INTRODUCTION

Many approaches to programming emphasize the use of specifications of interfaces. The basic idea is to achieve a separation of concerns. The *client* of an interface looks at the specification and writes code that uses the interface. He need not concern himself with how the specified behavior is achieved. The implementor's job is to provide an implementation that satisfies the specification and is efficient in the contexts in which it is used. In practice what often happens is that the first implementation proves unacceptably inefficient and must be tuned. Changing the implementation may necessitate re-compiling the client, but the client source code should not have to be changed.

Unfortunately, what often occurs in practice is that achieving the desired efficiency requires changing the interface; then all bets about the client code are off. Consider the following scenario. Suppose that the specifications in Figure 1 are implemented by an unsorted list of domain/range pairs with the invariant that no domain element appears more than once. To maintain this invariant, the implementation of `Map_Define` must, at each invocation, search the list to find out if domain element being defined is already there. If clients of map have code that looks like that in Figure 2, they will run unnecessarily slowly, because time will be spent in the implementation of `Map_Define` checking if a domain element is in the list—when the calling context guarantees that it isn't.

One conventional solution to this problem is to extend the interface by adding `Map_DefineUnique`, as specified in Figure 3. Depending upon the context of the call, clients can then call either `Map_Define` or `Map_DefineUnique`, depending upon whether or not the requires clause of `Map_DefineUnique` is satisfied. There are, however, some unfortunate ramifications of solving the problem this way.

1. The unhappy client who wrote the code appearing in Figure 2 will have to change his code.
2. What about other clients? Will they all be notified and encouraged to look at their code?

```
Map_Define (Map *m, Dom d, Ran r)
      Modifies: *m
      Ensures: Defines r as the image of d in *m. If d is
      already defined in *m, changes the image of d to r.

Ran Map_Lookup (Map *m, Dom d)
      Modifies nothing.
      Ensures: If d is undefined in *m, returns NULL_RAN.
      Otherwise, returns the image of d in *m.
```

Figure 1: Interface for Maps

```
r = Map_Lookup(m,d);
if (r == NULL_RAN) {
        r = Ran_Create();
        Map_Define(m,d,r);
    }
```

Figure 2: An Opportunity to Optimize `Map_Define`

*Supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research Research under contract N00014-92-J-1795, and in part by the National Science Foundation under grant 9115797-CCR. Authors' address: MIT Lab for Computer Science, 545 Technology Square, Cambridge, MA 02139. Internet: {mtv,guttag}@lcs.mit.edu

Appeared in *Proceedings of the 1994 ACM/SIGSOFT Foundations of Software Engineering Conference.*

```
Map_DefineUnique (Map *m, Dom d, Ran r)
        Requires: d is undefined in *m.
        Modifies: *m
        Ensures: Defines r as the image of d in *m.
```

Figure 3: An Addition to Interface for Maps

3. A new opportunity for bugs has been created. Clients may call `Map_DefineUnique` when the invariant is in fact not satisfied. In general, having procedures with non-trivial requires clauses should be avoided when at all possible.

4. The solution may have to be applied at each level of abstraction. Suppose, for example, that maps are being used to represent graphs and that `Map_Define` is being called within an implementation of the procedure `Graph_AddNode`. Can the call to `Map_Define` be replace by one to `Map_DefineUnique`? Perhaps this can be done only when the node being added is new. If so, this might then lead to changing the graph abstraction by adding a procedure `AddNewNode`, etc.

5. Such a change is difficult to back out of. Suppose, for example, that the implementation of `Map` is changed to a binary tree. There will no longer be anything to be gained by calling `Map_DefineUnique` instead of `Map_Define`, but unless one is willing to revisit the client code yet again both interfaces will have to be maintained.

A better approach to the original problem is to change the implementation of `Map_Define` so that it can automatically take advantage of information available in the calling context, as illustrated in Figure 4, and leave the client code untouched. This avoids all of the problems listed above. Unfortunately, it raises other problems.

1. If any efficiency is to be gained, the implementation can't do a runtime test to decide which case it is in. Therefore, the compiler must be able to decide whether the guard is true.

2. In order for the compiler to know that it is safe to use the implementation that doesn't test for membership, it must be able to deduce at compile time that the actual parameter `d` is not in the map `m`. Deducing this from the context of the call and the implementations of `Map_Lookup`, `Map_Define`, and `Ran_Create` is well beyond the capability of contemporary compilers.

This paper presents an approach to dealing with these two problems. The basic idea is that by allowing the compiler to take advantage of information contained in (perhaps incomplete) formal specifications, it becomes possible to do a superior analysis of the program text. This analysis makes it possible to evaluate at compile time the guards of *Specialized Procedure Implementations*, SPIs, such as the one in Figure 4. That, in turn, allows one to write client programs that are both simple and efficient.

In the remainder of this paper we deal with three questions:

- Are there significant opportunities to improve program performance by using SPIs? We address this question, in Section 2, by reporting on our analysis of some legacy code.

```
Map_Define (Map *m, dom d, Ran r) {
        Pair  *p;
        p = FindPair(*m, d);
        if (p == NULL); {
           InsertPair(m,d,r);
           return;
           }
        p→ran = r;
    specialize when d is undefined in *m:
        InsertPair(m,d,r);
}
```

Figure 4: A Dual Implementation for `Map_Define`

- How does one go about building a compiler that takes advantage of these opportunities? We address this question by describing, in Section 3, how we built our prototype implementation of an SPI compiler. Section 6 discusses the extent to which this implementation realizes the potential opportunities discussed in Section 2.

- How much and what kind of work does a programmer have to do to take advantage of these opportunities? This is discussed in Section 4.

2 POTENTIAL FOR SPEED-UP

To estimate the potential benefits of SPIs, we examined two existing programs, neither of which we wrote. We tried to measure how much merely introducing SPIs might improve performance. For this purpose, we made the unrealistic assumption that a compiler would detect all opportunities to use SPIs. In Section 6, we shall see that one can come surprisingly close to satisfying this assumption.

The two programs that we studied perform very different tasks and are written in different programming languages. One program, AC-Unify, computes the unifying substitutions for terms containing associative-commutative operators [22]. It is 8,000 lines of commented CLU code. The other program, SIM, simulates an object-oriented database [7], and is 7,000 lines of commented C++ code.

These programs did have some key attributes in common. Although both were carefully designed for algorithmic efficiency, neither had been highly tuned for performance at the code-level. Thus, each had room for improvement. Also, both programs are well-structured and make good use of data abstraction. This made them relatively easy to read and understand.

For each program, we

1. profiled the program to identify hot spots,

2. introduced SPIs to speed-up these hot spots, and

3. measured the speed-up under the assumption that the compiler would detect all opportunities to use these SPIs to reduce the time spent in these hot spots.

In AC-Unify, we added SPIs for the three procedures shown in Figure 5. Each of the procedures is an operation of an abstract data type. The SPI for `set$insert` avoids the check to see whether `e` is already in `s`, which is represented as an unsorted array without duplicates. The SPI for `mapping$insert` is similar to that discussed in the

```
set$insert = proc (s: set[elem], e: elem)
     specialize when e is not in s

mapping$insert = proc (m: mapping[dom,ran],
                       d: dom,
                       r: ran)
                  signals (exists)
     specialize when d is not defined in m

assignment$create = proc (env: sequence[var])
                    returns (assignment[var,val])
                    signals (empty, duplicates)
     specialize when env contains no duplicate vars
```

Figure 5: SPIs of AC-Unify

```
substitution$store = proc (s: substitution,
                           v: variable,
                           t: term)
                     signals (exists)
     specialize when v is not defined in s
```

Figure 6: Propagated SPI in AC-Unify

introduction. The SPI for `assignment$create` avoids a check to see whether `env` contains duplicate `vars`.

Figure 6 contains the signature of a procedure that was specialized by propagating the SPI of `mapping$insert`. A `substitution` is represented as a `mapping[variable,term]`. Thus, the implementation of `substitution$store` calls `mapping$insert`. (AC-Unify makes `substitution` a separate data type to maintain invariants about `substitutions` that are not enforced by `mapping`.)

The procedures in Figure 5 and Figure 6 were called from a total of 14 call sites. Of the 14 sites, nine could be safely optimized and five could not. Optimizing the nine sites reduced running time by 14%. Writing the SPIs added a negligible amount of source code–seven lines.

In SIM, we added two SPIs for the procedure shown in Figure 7. The operator `[]` returns a reference to an element in a `DynamicArray`, a user-defined abstraction. The operator `[]` causes a runtime assertion failure if `index` is not within the current array bounds. The first SPI eliminates just the low bound check, and the second SPI eliminates both the high and low bound checks. In general, a third SPI that eliminates just the high bound check would be useful, but this SPI was unnecessary for the hot spots in SIM. These optimizations illustrate one common use of SPIs—removing unnecessary error checks. Such uses of SPIs are closely related to work done by [9] and [17] on eliminating checks mandated by the source language. The key difference is that SPIs allow the compiler to remove checks that are particular to programmer-supplied abstractions.

The hot spots of SIM invoke the operator `[]` from 12 call sites. Of these, ten require no bounds checks and two require low bound checks. When the unnecessary checks are eliminated, running time is reduced by 14%.

Note that SIM is a good example of why it's a bad idea to disable all error checks. There are conditions under which two of the checks prevent the program from clobbering memory.

```
class DynamicArray {
  public:
    Element& operator[](int index) const;
       specialize when index ≥ low bound
       specialize when
          index ≥ low bound and index < high bound
    ⋮
}
```

Figure 7: SPI of SIM

```
insert = proc (s: intset, e: int)
  requires --
  modifies s
  ensures s' = s^ ∪ {e}
```

Figure 8: A Speckle Procedure Specification

```
int = immutable type spec
  based on Integer
  ⋮

intset = mutable type spec
  based on IntegerSet
  ⋮
```

Figure 9: Speckle Data Type Specifications

3 TECHNICAL FRAMEWORK

As Section 2 indicated, there are significant opportunities for using SPIs to reduce the need to trade abstraction for efficiency. The next question to address is the feasibility of exploiting such opportunities.

The central problem is how to go about building a compiler that detects contexts where SPIs can be used. A subsidiary problem is designing a programming language that incorporates formal specifications and SPIs and related annotations. To come to grips with these problems, we built a prototype compiler for a dialect of CLU [14] that we call Speckle. The *Prototype Speckle Compiler*, PSC, incorporates primitive, automated theorem proving technology to identify opportunities to use SPIs. It does not generate code.

Speckle retains most of CLU's features, including static typing, side effects, data abstraction, a garbage-collected heap, iterators, and exceptions. The combination of side effects and pointers to heap-allocated data structures requires PSC to handle aliasing, the most challenging problem for optimizing compilers.

The specification language portion of Speckle builds on the Larch family of specification languages [10] and borrows heavily from the Larch/CLU language [21]. Procedure specifications consist of pre- and postconditions written in a stylized fashion.

Figure 8 is a specification of a procedure to insert an `int` into an `intset`. Here, the empty `requires` clause indicates that the precondition is vacuous (true). The postcondition

```
intset = mutable type
  rep = intlist

  insert = proc (s: rep, e: int)

    % General implementation
       if rep$member(s, e) then return end
       rep$insert_last(s, e)

    % Specialized implementation
       special when ¬(e ∈ s^)
         rep$insert_last(s, e)
    end insert
```

Figure 10: A Dual Implementation of `insert`

is the combination of the `modifies` clause, which states that the only visible side effects are to `s`, and the `ensures` clause, which states that upon return, the value of `s` is the union of {`e`} with the value of `s` before the call. The functions used in pre- and postconditions, *e.g.*, ∪, are defined in a separate language called the Larch Shared Language (LSL). LSL provides a syntax and semantics for specifying functions and sorts in first order multi-sorted predicate logic.

Speckle also has facilities for writing data type specifications, which consist of two primary parts. The first part declares whether objects of the type are *mutable* or *immutable*. Immutable objects have the same value throughout a program execution, whereas mutable objects can be modified to have different values at different times. Declaring a type to be immutable simplifies analysis of code that uses objects of the type, primarily because aliasing is not an issue for immutable objects. The `based on` clause specifies an LSL sort used to model the values of objects of the type. Figure 9 contains parts of the specifications of `int` and `intset`.

Once specifications are incorporated into the programming language, it is straightforward to add constructs to express SPIs. Speckle allows users to supply multiple implementations for a procedure. The first implementation is the general one—it must work correctly in any calling context. This is critical, since in general one cannot count on a compiler discharging the guard for any of the specialized implementations, and to be sound the compiler needs a safe default implementation to fall back on. Specialized implementations follow the general one and are prefixed by a `special when` clause. This clause supplies a guard condition, written in the syntax of the specification language, which the compiler must discharge before it can substitute the specialized implementation for the general one. Figure 10 shows how multiple implementations of the procedure `insert` from Figure 8 might be written.

The final component needed before building a compiler is a formal model for reasoning about programs. Our model includes a formalization of program states and a set of proof rules. We sketch our model here; a detailed description of the model used in PSC is contained in [20].

In the model, a procedure implementation is represented as a control flow graph (CFG) with six kinds of nodes: assignment, procedure call, iterator call, branch, merge, and loop. Each edge of the CFG has an associated program state that represents the state of the program when control is at the edge.

The program state contains the usual kind of information about the program store. The program state is represented using a logical theory available for reasoning about that point of the computation. This theory is built, in part, from the specifications of the data types used in the code.

To reason about the values of program states at edges in a CFG, we use a small set of proof rules (one for each the six kinds of nodes) similar to Hoare and Floyd rules [4, 12]. This is where procedure specifications come into play. The proof rule for a procedure call node uses the specification of the called procedure to define the program state after the call in terms of the program state before the call. All interprocedural analysis is based on propagating information in specifications. If a procedure, say `P`, invokes another procedure, say `P1`, the compiler uses information in the specification of `P1` to optimize `P`. It does not use information derivable from the code implementing `P1` to optimize `P`.

The organization of PSC is based directly on the proof rules. First, PSC builds a CFG. Then, using the proof rules for assignment, procedure call, iterator call, and branch nodes, PSC constructs a logical system for each edge in the CFG. Next, it uses the logical systems at each edge to try to discharge the guards of SPIs. During this step, PSC uses the proof rules for merge and loop nodes to perform automated proof-by-cases and proof-by-induction. Finally, PSC chooses the implementation associated with the first guard it is able to discharge. If no guard can be discharged, it chooses the general implementation.

Reasoning about the guards is done using equational conditional term rewriting. Early versions of PSC did this by invoking the LP [5] theorem proving system. This proved unsatisfactory. The current version of PSC internalizes the theorem proving technology. A great deal of code was borrowed from LP, and then modified to make it more suitable for this non-interactive application of theorem proving.

4 WHAT THE PROGRAMMER MUST DO

Our experience building and using PSC leaves little doubt about the technical feasibility of building SPI compilers. A more difficult question is whether the benefits of such a compiler are sufficiently large relative to the effort required to use one.

This effort falls into two classes:

- Supplying program documentation that might not normally be provided, *i.e.*,
  - Specifications of key interfaces,
  - Abstraction functions for implementations of data abstractions, and
  - Representation and module invariants.
- Looking for problems in specifications, as well as in code, when debugging and tuning performance.

In evaluating the cost/benefit tradeoff, it is important to keep in mind that supplying better quality program documentation has several advantages beyond facilitating the use of SPIs. Moreover, as discussed below, the amount of specification required to use SPIs is less than that suggested by good software engineering practice. In contrast, while many have suggested that supplying abstraction functions and invariants is useful [8], such information is, in practice, not usually provided.

Because SPI compilers rely on specifications, bugs in specifications can lead to optimizations that change the behavior of code. When such a program is run, it may exhibit behavior that cannot be explained merely by examining the code. Similarly, a program may exhibit performance that cannot be explained merely by examining the code. Since current debugging tools are not designed to work with SPI compilers, such bugs can be hard to find.

4.1 Supplying Specifications, Invariants, and Abstraction Functions

The information found in formal specifications of procedures and (in the case of Speckle) iterators is used to help discharge the guards of SPIs. Usually, a complete specification has more information than PSC actually needs.

To minimize the effort involved in using PSC, we designed Speckle to support partial specifications of procedures and iterators. A partial specification has a weaker pre- or postcondition than the "intended" specification.[1] In the extreme case, the specification is omitted entirely.

Often, the only part of a procedure specification needed to discharge the guard of an SPI is the `modifies` clause. To enable the compiler to approximate omitted `modifies` clauses, Speckle requires that each data type specify the types of mutable objects that may be contained by a value of the type. For example, a type for sets of mutable `Queue` objects would specify `contains Queue`. PSC uses the `contains` clauses to bound the types of objects that are reachable from a procedure's arguments, which in turn bounds the objects that the procedure may modify.

To test the utility of partial specifications, we deliberately wrote the weakest partial specifications needed to detect all of the optimizations in AC-Unify. Partial specification worked well: specifications were necessary for only 11 of the roughly 300 procedures in AC-Unify. In total, we wrote 69 lines of specifications in Speckle and 139 lines of LSL—a small fraction of the 8,000 lines of commented source code. Furthermore, many of the LSL specifications were for generic sorts such as `Integer`, `Mapping`, `Sequence`, and `Set` that should probably be considered to be part of a theorem proving library.

Partial specifications worked well for two reasons. The first reason was modularity. To optimize a hot spot in a procedure, P, the compiler needed only the procedure specifications of P and those procedures called by P. Because the program was decomposed into many small procedures, the fan-out of the call graph was small, so only a small number of procedure specifications were needed for each optimization.

The second reason that partial specifications worked well was the `contains` clauses. When a `modifies` clause was omitted, PSC could still deduce a useful approximation using the `contains` clauses. In AC-Unify, PSC deduced seven `modifies` clauses that were essential for four of the optimizations.

In addition to the specifications, PSC sometimes needs invariants and abstraction functions. Programs often rely on invariants to constrain the values of data that is accessed by a group of procedures. PSC, which performs no interprocedural analysis, may need such invariants to discharge the guards of SPIs.

[1] For the purpose of detecting SPIs, the distinction between partial and intended specifications is unnecessary. However, for other optimizations performed by PSC, the distinction is necessary for soundness.

Abstraction functions[2] [13] map values used to represent an abstract type into values of the abstract type. Suppose that P is a procedure in the interface of data type T. PSC uses the abstraction function of T in two ways. One is to translate the information in the precondition of P from the abstract level used in P's specification down to the representation level used in P's implementation. This makes more information available for discharging guards of SPIs in P's implementation. The other way PSC uses the abstraction function is to propagate an SPI up through P's implementation, a technique which is explained in [20]. In AC-Unify, PSC used the abstraction function of `substitution` to automatically propagate the SPI of `mapping$insert` to its caller, `substitution$store`. Thus, PSC is able to optimize the callers of `substitution$store`.

Again, the question we wanted to explore was how little information is needed by PSC to perform the desired optimizations. We therefore provided abstraction functions and invariants only where they were needed to enable optimizations in AC-Unify.

PSC needed representation invariants for the implementations of the `mapping`, `partition_tree`, and `solution` data types, and abstraction functions for `mapping` and `substitution`. In total, this required three lines to state the representation invariants and seven lines to state the abstraction functions.

4.2 Debugging and Performance Tuning

When a program has been compiled using an SPI compiler, it may exhibit behavior and performance that cannot be understood merely by examining the code. This means that programmers must consider both code and specifications when debugging and performance tuning.

When the code and the specifications are inconsistent with each other, an SPI compiler may perform unsound optimizations. One way to avoid such problems is to detect at compile time places where the code and the specifications may be inconsistent with each other. While this is an undecidable problem, there are many occasions where such inconsistencies can be found relatively easily [3]. It is relatively easy, for example, to flag places where a formal parameter that should not be modified is mutated. Such mutations are not necessarily violations of the specification, because the side-effect may later be undone or may be invisible to the caller, but they are good places to look for bugs.

Another possibility is to have the compiler generate, or the programmer supply, code to check the guards of specialized implementations. In debugging mode, this code can be executed to detect places where an inappropriate SPI has been chosen. This is similar to the assertion checking supplied by ANNA [15].

Finally, the compiler could be directed to selectively turn off optimizations. This can be used to pinpoint which optimizations triggered the changed behavior.

When tuning performance, we first use conventional techniques, *e.g.*, profiling, to localize the problem. If the identified code region includes calls to SPIs, we then examine the output of the compiler to see if the implementation chosen is the one we expected. If it is not, we then need to understand why.

Our experience suggests that the most common cause of PSC failing to choose the appropriate specialized implemen-

[2] Abstraction functions are analogous to reification in the VDM literature.

tation is the compiler's limited theorem proving capability. In particular, it sometimes fails to deduce appropriate consequences from underlying LSL specifications. One way to circumvent this problem would be to provide a mechanism that would allow programmers to insert assertions in their code. We are reluctant to do this, however, since we fear that wishful thinking will often lead to the insertion of incorrect assertions. A sounder approach, which we use, is to add *implications* (lemmas deducible from the axioms) to LSL specifications.

To identify useful implications, we used LP to examine the logical system at the point where PSC failed to choose the appropriate specialization. This was easy for us, but then we understand both PSC and LP well. Others would surely have more difficulty. Part of the solution to this problem is to reduce the need for implications by using better theorem proving technology, *e.g.*, by incorporating more of LP. Another part is to provide diagnostics to help the user identify missing implications. The challenge is to provide diagnostics that contain the necessary information and are concise and easy to understand.

5 RELATED WORK

SPIs are but one of a variety of techniques for improving program speed by introducing special-purpose code. Another approach is to inline procedure calls so that information in the calling context is available to perform traditional optimizations, such as constant folding. This, in essence, is a form of partial evaluation.

SPIs and inlining are complementary approaches. Inlining is good for performing low-level optimizations, such as eliminating common subexpressions located in different procedure bodies. SPIs, on the other hand, are good for higher-level optimizations, such as not checking whether a newly allocated element object is a member of a previously existing set object.

Speckle is not the first language that allows users to define optimizations. In [11], Hisgen presents an unimplemented design of a strategy based on transformation rules rather than specifications. To define an optimization, an implementor describes transformations to be performed by the compiler. A transformation rule may have a precondition expressed using applications of side-effect free functions, which play a role analogous to that of LSL functions. Thus, the transformation language is sufficiently powerful to express any specialized procedure. In fact, the transformation language is more expressive than Speckle.

The main problem of the transformation rule strategy is that it lacks modularity. To apply a transformation rule, the compiler must reorder the program so as to match the pattern of the rule. The problem is that to commute one procedure call with another, the compiler must in general rely on "commutative" transformation rules supplied by the user. To maximize the compiler's ability to perform transformations, the user must consider all pairs of procedures. In contrast, Speckle uses `modifies` clauses—one per procedure—to determine whether a procedure call interferes with an optimization.

A common use of specialized procedures is to eliminate runtime checks. Many have focussed on eliminating such checks for operations that are primitive to the source language, e.g., array bounds checking, nil checks in pointer dereferences, overflow, assignments from supertypes to subtypes, etc. Specialized procedures are more general because they can be used to eliminate runtime checks that are not primitive to the source language.

In [18], Sites describes a technique for proving that programs written in a language like Algol 60 terminate without runtime errors. This requires proving properties sufficient to eliminate runtime checks in array references, numeric operations, assignments from supertypes to subtypes, etc. The language does not have pointers, so the problem of aliasing is simpler than in Speckle. Sites simulates his technique manually on several examples.

In [6], German develops a tool for verifying the absence of runtime errors, such as arithmetic overflow and invalid array indices. Users write formal specifications for procedures (entry and exit assertions) and decorate their code with sufficiently strong assertions so that the verifier can discharge all of the assertions plus the absence of runtime errors. German's work focuses on defining Pascal formally and expressing assertions sufficient to preclude a runtime error. He does not describe the strategies used to discharge assertions.

In [16], McHugh examines all of the static checks of Gypsy, a derivative of Pascal. Gypsy is a programming environment for verified software, so programs typically contain entry, exit, and other assertions. McHugh's compiler generated optimization conjectures that, when discharged by the UT Interactive Prover [2], resulted in the elimination of code supporting exceptions—i.e., a broad category of runtime checks. McHugh does not describe strategies used to prove the conjectures.

In [9], Gupta reduces the overhead of array bounds checks by eliminating redundant checks that occur in code fragments such as "`a[i] := a[i]+1`" and by moving checks out of loops. The strategy used relies on the programming language semantics of arrays and does not extend to user-defined types.

Currently, Greg Nelson and David Detlefs are studying ways of eliminating array bounds checking, nil checks, and other runtime checks in Modula-3 [17].

6 STATUS AND CONCLUSIONS

We have extended the programming language CLU [14] to include SPIs and, using the framework outlined in Section 3, built a prototype compiler for that language. PSC identifies sound optimizations, but does not apply them. In addition to performing optimizations based on specialized procedures, PSC takes advantage of the information provided in specifications to perform enhanced global common subexpression elimination, code motion out of loops, and dead code elimination [1]. Those aspects of our work are reported in [19, 20].

When PSC was used on the AC-unify program in the way described in Section 2, it was able to optimize all of the nine possible calls. This resulted in a performance improvement of 14%. Compiling the procedures containing the nine call sites took 105 seconds on a DEC Alpha running at 150 Mhz. This is unacceptably slow, but PSC could easily be sped up considerably.

CLU is an unusual programming language. It has many features, *e.g.*, garbage collection, exception handling, and iterators, not found in most imperative programming languages. To help us better understand the general utility of SPIs, we recently began looking at applying our ideas in the context of C++. As discussed in Section 2, there seem to be plenty of opportunities to apply SPIs in that context.

We have not yet implemented a compiler for C++ plus SPIs. Therefore we cannot be sure how many of the opportunities for optimization identified in Section 2 are realizable by a practical compiler. To estimate this we translated the relevant sections of SIM into Speckle and fed them to PSC. PSC identified 14 of 22 possible optimizations, which improved performance by 9%. The missed optimizations would have brought the overall improvement to 14%. PSC took 109 seconds to compile the procedures containing the 22 call sites.

For many years, those of us working on what we like to think of as the foundations of software engineering have concentrated on understanding how to create programs with appropriate functionality. We have devoted relatively little attention to understanding how to achieve appropriate performance. This is unfortunate, since much of the complexity of useful programs is caused by the desire to achieve good performance. In this paper, we presented a mechanism that has the potential of greatly reducing the need to trade simplicity for efficiency.

Our experiments have convinced us that SPIs can be used to write code that is at once clear and efficient. It remains to be seen how much effort a typical programmer must expend to realize these advantages.

7 ACKNOWLEDGMENTS

Kathy Yelick and Sanjay Ghemawat provided the AC-Unify and SIM programs. Steve Garland helped us to integrate theorem proving into PSC and in general provided help throughout this work. Jim Horning provided sound advice on several occasions. Dorothy Curtis maintains the PCLU system, which was used to implement PSC.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] W. W. Bledsoe and M. Tyson. The UT Interactive Prover. ATP 17, University of Texas Mathematics Dept., May 1975.

[3] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, December 1994.

[4] R. W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics*, volume 19, pages 19–31. American Mathematical Society, 1967.

[5] S. Garland and J. Guttag. A guide to LP, The Larch Prover. TR 82, DEC Systems Research Center, Palo Alto, CA, December 1991.

[6] S. M. German. Verifying the absence of common runtime errors in computer programs. Technical Report CS-81-866, Stanford, June 1981.

[7] S. Ghemawat. Disk management for object-oriented databases. In *Third International Workshop on Object Orientation in Operating Systems*, pages 222–225. IEEE Computer Society, December 1993.

[8] D. Gries, editor. *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*. Springer-Verlang, 1978.

[9] R. Gupta. A fresh look at optimizing array bound checking. In *Proceedings of the 1990 Conference on Programming Language Design and Implementation*, pages 272–282. ACM, June 1990.

[10] J. V. Guttag, J. J. Horning, with S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[11] A. Hisgen. *Optimization of User-Defined Abstract Data Types: A Program Transformation Approach*. PhD thesis, Carnegie-Mellon University, 1985.

[12] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[13] C. A. R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1(1):271–281, 1972.

[14] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*. Lecture Notes in Computer Science No. 114. Springer-Verlag, 1981.

[15] D. C. Luckham, F. W. von Henke, B. Krieg-Bruckner, and O. Owe. *ANNA Reference Manual*, volume 260 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[16] J. McHugh. Towards the generation of efficient code from verified programs. Technical Report 40, University of Texas at Austin, March 1984.

[17] G. Nelson and D. Detlefs. Extended static checking. Private communication on work in progress at DEC Systems Research Center, 1994.

[18] R. Sites. Proving that computer programs terminate cleanly. Technical Report CS-74-418, Stanford, 1974.

[19] M. T. Vandevoorde. Specifications can make programs run faster. In *Proceedings of TAPSOFT '93*. Springer Verlag, April 1993.

[20] M. T. Vandevoorde. Exploiting specifications to improve program performance. Technical Report MIT/LCS/TR-598, M.I.T., Cambridge, Ma 02139, February 1994.

[21] J. M. Wing. A two-tiered approach to specifying programs. Technical Report MIT/LCS/TR-299, M.I.T., 1983.

[22] K. Yelick. A generalized approach to equational unification. Technical Report MIT/LCS/TR-344, M.I.T., August 1985.