Using a Single-Threaded Object to Speed a Verified Graph Pathfinder

Matthew Wilding

Rockwell Collins, Inc.
Advanced Technology Center

mmwildin@collins.rockwell.com

Abstract

We have written hardware simulators in ACL2 in order to unify high-speed simulators and formal analysis models [2, 7]. The techniques used for these simulators extend to other kinds of software, which we demonstrate in this paper by implementing a much faster version of an algorithm for graph pathfinding previously verified by J Moore using ACL2 [5]. This exercise also highlights a weakness in ACL2: the occasional need to add computational complexity to functions in order to admit them to the logic.

1 Introduction

Formal verification of software requires the availability of a clear, formalizable specification. Hardware device simulators usually have such a specification as hardware developers typically understand what the device under development is supposed to do. We have been building simulators for microprocessor microarchitectures in the ACL2 logic to support both standard simulator use and formal, machine-assisted design analysis [2, 7].

An important consideration when writing software is execution efficiency. Our experience writing hardware simulators with straightforward use of the applicative ACL2 logic is that there are inefficiencies associated with maintaining multiple copies of the program state. In a benchmark of a toy microarchitecture model reported in [7], an applicative Common Lisp program executed more than 100x slower than an equivalent C-language simulator implementation. The performance difference stems from the overhead associated with "objects" introduced by the Lisp compiler to accommodate Common Lisp arbitrary-sized integer arithmetic and the the need to maintain multiple copies of program state. This overhead is unacceptable in software where performance is an issue.

We want to verify software formally, so we would prefer to use an applicative language such as that supported by the ACL2 theorem prover. However, we generally cannot tolerate low performance. One important execution optimization for applicative programs is to code in a style in which only a single copy of the program state exists at any instant and use this property to optimize program execution speed by replacing the data structure operations with fast, destructive operations. Software that accesses data structures sequentially in this way is called *single-threaded*. We have developed several methods for exploiting single-threadedness in applicative code, such as using Lisp macros to enforce a single-threaded programming style and building a tool that detects violations of this style [3].

Fortunately, such homegrown approaches are no longer necessary. At least two theorem provers have added a capability for efficient execution that exploits single-threadedness reliably. PVS automatically detects occasions when this optimization can be made and provides an execution environment that exploits it [6]. The ACL2 system has also recently been extended to support *single-threaded objects*, or *stobjs*. ACL2 enforces restrictions on the use of stobjs to ensure that stobjs are not copied, and provides a destructive implementation of stobjs that allows operations on them to execute quickly [1]. Although stobjs are relatively new to ACL2, they are basically a user-accessible version of what has always existed in ACL2 in its handling of STATE [4]. They combine a functional semantics about which we can reason with a high-speed imperative implementation.

This paper describes the application of stobjs to speed a small, previously-verified ACL2 algorithm that finds a path in a directed graph [5]. This small example illustrates how ACL2 can be used to develop verified programs that execute efficiently.

2 Moore's Pathfinder Proof

J Moore presents a proof of a linear-time pathfinding algorithm in [5], and the corresponding ACL2 input is in the standard ACL2 distribution. It is a good example of the development of a verified algorithm using ACL2. The path-finding algorithm, called `linear-find-path`, searches a graph to find a path between two vertices. The algorithm maintains a data structure that represents the vertices that have already been visited (the *marked* vertices) and does not explore candidate paths with marked vertices. The algorithm's theoretical worst-case complexity is linear in the number of edges in the graph being searched, since the number of basic operations required to run the algorithm — operations

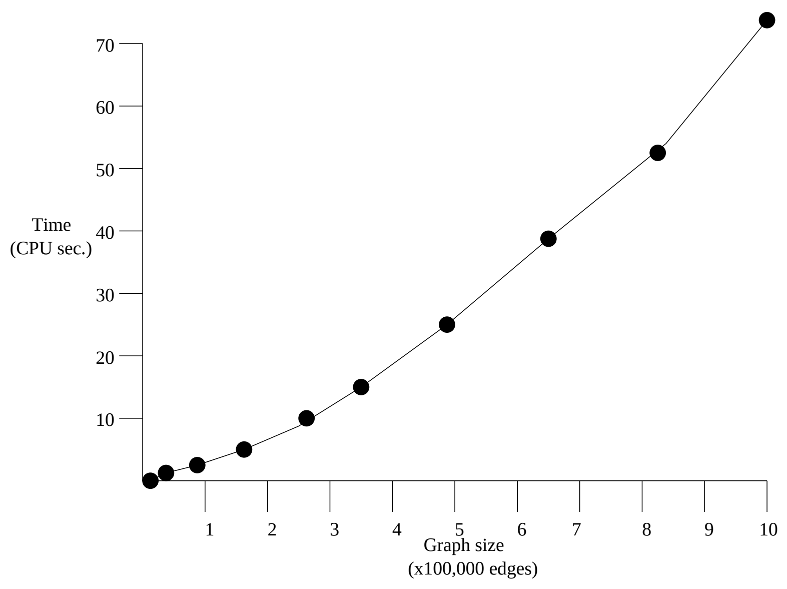


Figure 1: Execution Time of `linear-find-path`

such as marking a vertex, finding the edges emanating from a vertex, or checking whether a vertex is marked — increases in the worst case linearly as one increases the number of edges in the graph. The `linear-find-path` algorithm is proved to return a path if one exists.

ACL2 can not only verify algorithms but also be used to verify implementations. The formalization in ACL2 of `linear-find-path` in [5] is also an implementation since it is expressed in the executable ACL2 logic. Figure 1 presents the time required for `linear-find-path` to execute a benchmark. For the purposes of this benchmark, we construct a graph with vertices 0 to $N+1$ where there is no edge to vertex $N+1$ and each of the vertices 0 to N has an edge to each of the vertices 0 to N. Searching for a path from vertex 0 to vertex $N+1$ causes the algorithm to search (unsuccessfully) the entire graph of N^2 edges.

The benchmark results appearing in Figure 1 indicate that the implementation of `linear-find-path` has time complexity that is somewhat more than linear. Indeed, `linear-find-path` is inefficient because the implementations of the underlying data structure operations upon which it depends are inefficient. The program uses lists to represent the graph and other needed data structures, and the speed of the data structure operations is slower on larger graphs. If one considers the basic computational operations to be primitives such as comparing two values or setting a pointer, then the number of operations required to execute `linear-find-path` increases in the worst case faster than linearly.

3 A stobj-based implementation

We implement a version of the pathfinding program that uses data structure operations provided by ACL2's stobj mechanism so as to improve the speed of the `linear-find-path` algorithm. The nonlinear time complexity is the result of the underlying data structure operations, and by changing the underlying data structure we can provide a linear-time implementation. We define a single-threaded object `st` that contains the data we need.

```
(defstobj st
  (g  :type (array list (10000)) :initially nil)
  (marks :type (array (integer 0 1) (10000)) :initially 0)
  (stack :type (satisfies true-listp))
  (status :type (integer 0 1) :initially 0))
```

Graph vertices are numbered with a natural less than 10,000. Element `g` of stobj `st` is used to represent the graph by recording in the array element corresponding to a vertex's number a list of its *children* — those vertices to which it has an edge. Element `marks` is an array of bits that indicate which vertices have been already visited. Element `stack` contains a stack containing the current path being explored. A final element, `status`, contains a flag that indicates with a 1 when the algorithm has failed to find a path. Each of these data structure elements corresponds to a data structure in the implementation of `linear-find-path` in [5].

We implement a measure function for the path-finding algorithm. Each step of the algorithm reduces the children of the vertex currently being explored while maintaining the number of marked vertices, or reduces the number of vertices not yet marked. Again, this corresponds to what was done in [5], except now it is defined in terms of the stobj-based data structure.

```
(defun measure-st (c st)
  (declare (xargs :stobjs st
                  :guard (stp st)))
  (cons (1+ (number-unmarked st)) (len c)))
```

We formalize the notion of a "good" graph using the function `graphp-st` which checks that each vertex's list of children contains only valid vertex numbers. The function `bounded-natp` returns whether the its first argument is a natural less than its second argument, and `numberlistp` returns whether its first argument is a true-list containing naturals less than its second argument.

The stobj-based version of the find-path algorithm appears in Figure 2. It works much as the original in [5], except for its use of the stobj-based operations. Note that the ACL2 syntactic restrictions on the use of stobjs in definitions guarantee that `st` is accessed in a single-threaded way as described in [1].

```
(defun linear-find-next-step-st (c b st)
  (declare (xargs :stobjs st
                  :measure (measure-st c st)
                  :guard (and (graphp-st st)
                              (bounded-natp b (maxnode))
                              (numberlistp c (maxnode)))
                  :verify-guards nil))
  (if (endp c) st
    (let ((cur (coerce-node (car c)))
          (temp (number-unmarked st)))
      (cond
       ((equal (marksi cur st) 1)
        (linear-find-next-step-st (cdr c) b st))
       ((equal cur b)
        (let ((st (update-status 0 st)))
          (update-stack (myrev (cons (car c) (stack st))) st)))
       (t (let ((st (update-marksi cur 1 st)))
            (let ((st (update-stack (cons (car c) (stack st)) st)))
              (let ((st (linear-find-next-step-st (gi cur st) b st)))
                (if (or (<= temp (number-unmarked st))  ; always nil
                        (equal (status st) 0))
                    st
                  (let ((st (update-stack (cdr (stack st)) st)))
                    (linear-find-next-step-st (cdr c) b st))))))))))))
```

```
(defun linear-find-st (a b st)
  (declare (xargs :stobjs st
                  :guard (and (stp st)
                              (bounded-natp a (maxnode))
                              (bounded-natp b (maxnode))
                              (graphp-st st))))
    (let ((st (linear-find-next-step-st (list a) b st)))
      (if (not (equal (status st) 0))
          (mv 'failure st)
        (mv (stack st) st))))
```

Figure 2: Stobj-based pathfinder

4 Comments on the Proof

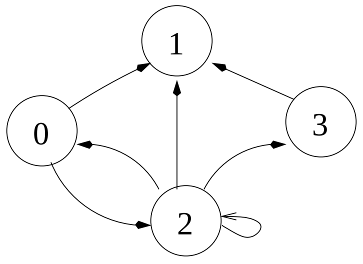
The stobj-based implementation and the proof of its equivalence with the verified algorithm of [5] accompany this paper. The proof requires the standard sort of lemmas needed in ACL2 proofs to guide the theorem prover. The fact that it is about a program that runs fast is irrelevant, which is the motivation behind the introduction of stobj in ACL2.

An interesting aspect of the proof is the complexity of the induction scheme required to show the equivalence of the pathfinder implementations. As can be seen in Figure 2, the function `linear-find-next-step-st` contains a recursive call with an argument that is the result returned by another recursive call. The induction scheme has a similar structure. In the case of this proof, the ACL2 induction-generation heuristics fail to generate a good scheme, so we provide one explicitly. (See `induct-equiv` in the accompanying ACL2 input.) The induction scheme provides an induction hypothesis that reflects a single step of the algorithm for both implementations of the algorithm, and also calculates a result that can be used in the second recursive call in a manner consistent with the operation of the algorithm. This is a rare instance where the value calculated in a definition used to provide an induction scheme matters!

The function `load-st` translates a graph represented using lists as expected by `linear-find-path` into a graph represented with a stobj as expected by `linear-find-st`. The final lemma that shows the equivalence of the stobj implementation and the original verified algorithm is

```
(implies
 (and
  (bounded-natp a (maxnode))
  (bounded-natp b (maxnode))
  (mygraphp g)
  (stp st))
(equal
 (car (linear-find-st a b (load-st g st)))
 (linear-find-path a b g)))
```

We demonstrate the application of the correctness lemma on an example. Consider the graph



We find a path from vertex 0 to vertex 3 using both implementations. As guaranteed by the lemma above, the paths are the same.

```
ACL2 !>(assign g '((0 2 3) (1) (2 0 1 2 3) (3 1)))
 ((0 2 3) (1) (2 0 1 2 3) (3 1))
ACL2 !>(mygraphp (@ g))
T
ACL2 !>(let ((st (load-st (@ g) st))) (linear-find-st 0 3 st))
((0 2 3) <st>)
ACL2 !>(linear-find-path 0 3 (@ g))
(0 2 3)
ACL2 !>
```

5 An Unsupported Performance Optimization

But there's a fly in the ointment. Function `linear-find-next-step-st` in Figure 2 contains a computationally expensive test that does not affect the result calculated by the function. The term `(<= temp (number-unmarked st))` in the body of the function supports a proof that the previously-discussed measure function `measure-st` is reduced on each recursive call of the function. This proof allows us to admit the function in the ACL2 logic. The test is irrelevant (except for justifying the function's admissibility) because it always evaluates to `nil`, since the recursive call of the function never increases the number of marked vertices. Obviously, since we are concerned with execution speed, it is desirable to eliminate this check. Unfortunately, the check is needed to prove termination of the function and is therefore necessary in order to admit the function into ACL2.

We prove that this check is irrelevant with the lemma in Figure 3. This lemma states that when the guards to `linear-find-next-step-st` are satisfied, then the function's body can be replaced by a version that does not contain the irrelevant check. Unfortunately, despite the proof of the lemma in Figure 3, there is no ACL2-supported way to use the version of the algorithm without the irrelevant check to justify termination of the function and thereby admit it to the ACL2 logic.[1]

[1] It is possible in this case to replace the check with a more-efficient one, but it is not possible to eliminate the check altogether. For example, since the number of recursive calls is bounded by total number of edges in the graph, one more-efficient check would be to test that a counter that is decremented on each recursive call is negative. If the counter is initialized with the value of the total number of edges, then this test would always return `nil`. Of course, while this check would allow admissability and would be more efficient, it would be similarly irrelevant to the value produced by the algorithm.

Note also that while it seems that the simpler version of the function without the "irrelevant" check terminates, the lemma of Figure 3 does not imply this.

```
(defthm linear-find-next-step-st-simpler
  (implies
   (and
    (graphp-st st)
    (bounded-natp b (maxnode))
    (numberlistp c (maxnode)))
   (equal
    (linear-find-next-step-st c b st)
    (if (endp c) st
      (cond
       ((equal (marksi (car c) st) 1)
        (linear-find-next-step-st (cdr c) b st))
       ((equal (car c) b)
        (let ((st (update-status 0 st)))
          (update-stack (myrev (cons b (stack st))) st)))
       (t (let ((st (update-marksi (car c) 1 st)))
            (let ((st (update-stack (cons (car c) (stack st)) st)))
              (let ((st (linear-find-next-step-st (gi (car c) st) b st)))
                (if (equal (status st) 0)
                    st
                  (let ((st (update-stack (cdr (stack st)) st)))
                    (linear-find-next-step-st (cdr c) b st))))))))))
  :rule-classes :definition)
```

Figure 3: Lemma justifying elimination of irrelevant check

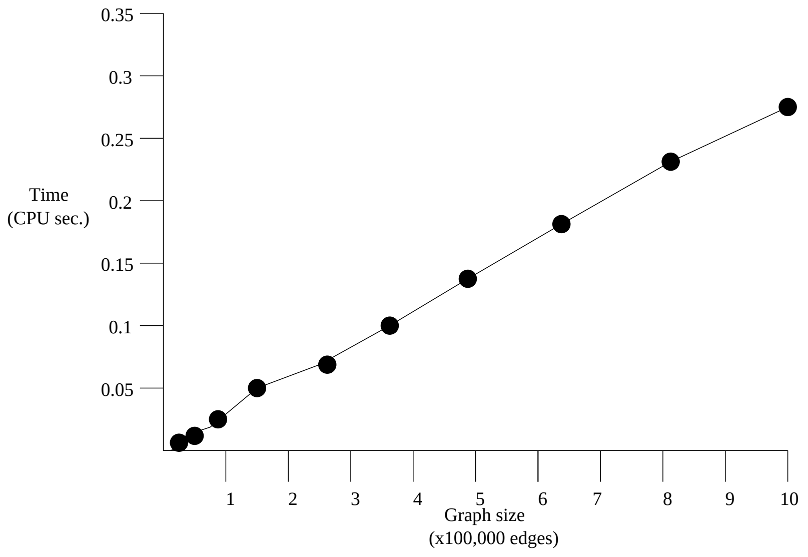


Figure 4: Execution Time of `linear-find-st`

We take matters into our own hands and use the ACL2 "skip-proofs" command to define `linear-find-st`, a version of the stobj-based pathfinder that omits the irrelevant check. Lemma `linear-find-next-step-st-simpler` implies that this version of the algorithm is consistent with the version we have verified, but it is unsettling not to be able to use ACL2 to check our work here. The ACL2 developers are considering adding a "defbody" command to ACL2 that could be used to define `linear-find-st`, requiring the user to justify it by proving a lemma similar to `linear-find-next-step-st-simpler`. This example suggests that this enhancement would be beneficial.

6 Benchmark Results

Figure 4 presents the result of benchmarking the stobj-based implementation (omitting the check we have proved irrelevant) in the same way as the original implementation's benchmark presented in Figure 1. The new implementation executes faster — 0.28 seconds versus 78 seconds for the benchmark with 1,000,000 edges. It executes in linear-time, and is guaranteed correct by virtue of the equivalence lemma presented here and the correctness lemma of [5].

7 Conclusions

Our conclusion from writing hardware simulators is that ACL2 enhanced with stobjs can express complex software that executes efficiently. The experience documented in this paper suggests that software other than hardware simulators can benefit from this technique, providing algorithm implementations that are provably correct and that execute at their theoretical maximum efficiency. This exercise also highlights a weakness in ACL2, which is the occasional need to write unnecessarily complex functions.

References

[1] Robert S. Boyer and J Strother Moore. Single-threaded objects in ACL2, 1999. `http://www.cs.utexas.edu/users/moore`.

[2] David Greve, Matthew Wilding, and David Hardin. High-speed, analyzable simulators. In *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000. `http://www.pobox.com/users/hokie/docs/-hsas.ps`.

[3] David Hardin, David Greve, Matthew Wilding, and John Cowles. Single-threaded formal processor models: Enabling proof and high-speed execution. Technical report, Rockwell Collins Advanced Technology Center, Cedar Rapids, IA, 1999. `http://www.pobox.com/users/hokie/docs/tr99.ps`.

[4] M. Kaufmann and J S. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203 – 213, April 1997.

[5] J Strother Moore. An exercise in graph theory. In *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

[6] Natarajan Shankar. Efficiently executing PVS. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 1999.

[7] Matthew Wilding, David Greve, and David Hardin. Efficient simulation of formal processor models. *Formal Methods in System Design*, to appear. Draft TR available as `http://pobox.com/users/hokie/docs/efm.ps`.