Accelerating Object-Oriented Simulation via Automatic Program Specialization

Daniel Weise and Scott Seligman

Technical Report: CSL-TR-92-519
(also FUSE Memo 92-10)

April, 1992

Computer Systems Laboratory
Departments of Electrical Engineering & Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

Object-oriented simulations in an object-oriented environment are easier to construct and maintain than conventionally programmed simulations. Unfortunately, they are also slower because of message passing and other runtime overhead. We have developed an automatic program transformer that solves the efficiency problem for a large class of simulation programs. It automatically constructs an efficient program from the inefficient simulation program and the objects it will receive as input. Depending on the object-oriented language used, and the application, the new program can be more than an order of magnitude faster than the original program. In this paper we describe the benefits of object-oriented simulation, our transformer, and how such dramatic speedups are possible.

This research has been supported in part by NSF Contract No. MIP-8902764, and in part by Advanced Research Projects Agency, Department of Defense, Contract No. N0039-91-K-0138.

Key Words and Phrases: Object-Oriented Simulation, Partial Evaluation, Program Transformation, Program Specialization

Copyright © 1992, Daniel Weise and Scott Seligman

Introduction

Object-oriented programming, which was invented to ease the construction of simulators [16], offers many benefits over conventional methods for constructing simulators. Object-oriented simulators directly model the system being simulated. Because each object is responsible for its own behavior, implementations of simulators become more modular, and the customization of simulators becomes routine. When an object-oriented programming environment is employed, the object metaphor is available to the user, not just to the programmer. Unfortunately, object-oriented simulation can be slow because of message passing overhead. The more object-oriented and powerful the programming system, the larger the overhead.

We have constructed an automatic *program specializer* (also called a *partial evaluator*) to help solve the performance problem of object-oriented simulations. A program specializer is a program transformer: given a program and a description of the class of inputs the program will be run on, it constructs a new program that is tuned (specialized) for that class of inputs. Program specializers have created compilers and compiler generators [11], rediscovered important algorithms [9], accelerated computations by two orders of magnitude [5], parallelized scientific code [3], and optimized programs [4]. We have extended program specialization technology to apply to object-oriented programs and object-oriented simulations, and have achieved greater than order-of-magnitude speedups for them.

A simple description of program specialization assumes a two input program $P(a, b)$: a specializer is a program that, given P and the first argument A of P, produces a new program R (Figure 1). The specialized program R has one input that corresponds to the second input of P. The output of R when run on some value B is identical to running P on A and B. For example, consider P to be an interpreter for some programming language whose first input is a program to be interpreted and whose second input is the data the program will be run on. Specializing P with respect to the program will "compile" the program. Experiments have shown that compiling a program in this way yields order of magnitude speedups over simple interpretation [15, 11, 6]. As another example, let P be a numerical integrator, such as a fourth order Runge-Kutta integrator, and let P's first and second arguments be a program that computes a differential and the initial conditions, respectively. Specializing P with respect to the differential produces a specialized integrator that can be order of magnitude faster than the general integrator [5].

$$Result = P(A, B)$$

$$P' = Specialize(P, A)$$

$$Result = P'(B)$$

Figure 1: Idealized Equations for Program Specialization. A program specializer *Specializer* accepts a two argument program P, and the first argument to the program. It then produces a new program P' that accepts the second argument, and that runs more rapidly than P.

In these examples the first argument fixes most of the structure of the computation, *i.e.*, which operations will be invoked and when they will be invoked. The original program must constantly rediscover and reconstruct this fixed computational structure. For example, consider an interpreter as it interprets the body of a loop. It reexamines the same expressions over and over again, each time determining the type of the expression, the operator, and the location of the operands. This information is fixed because the program doesn't change as the program runs. If the interpreter knew ahead of time the type of the expression, and its operator and location of the operands, it would simply perform the computation directly, saving considerable time and

overhead.

A specializer produces a new program that performs only the computations required by the varying (*i.e.*, second) input of the computation. A specializer achieves this goal by symbolically executing the program on its first input. Any computations that depend only on the first input occur during specialization, and will not occur when the specialized program runs. Considering the interpreter again, the specializer determines the type of each expression, plus the operator and location of the operands. This determination is not repeated at runtime. Computations that cannot be performed until the second argument is provided appear in the specialized program.

This idea is best presented with an example. Consider an interpreter for directed acyclic dataflow graphs where each interior node in the graph has two incoming arcs for its inputs, and one outgoing for its output. Each node specifies an arithmetic operation. Inputs nodes have no incoming arcs. The graph interpreter takes a graph as its first argument, and an assignment of numbers to input nodes as its second argument. It then "evaluates" the graph, assigning to each node the value that it computes based on the node's inputs and operation. The main loop of the interpreter might be

```
process_node(node) =
  input_a := node.lwire.source.value;
  input_b := node.rwire.source.value;
  new_value :=
    case node.operator in
      '+':  input_a + input_b,
      '-':  input_a - input_b,
      ...;
  current_node.value := new_value;
  output_arcs := node.output_arcs;
  while output_arcs != nil do {
    process-node(*output_arcs);
    output_arcs :=
      output_arcs->next-arc};
```

where the initial nodes to process are the successors of the input nodes. Symbolically executing the main loop of the simulator on the graph of Figure 2 would produce the following assembly-level code

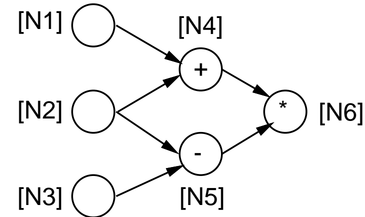


Figure 2: A graph to be simulated. The names in brackets are the symbolic starting address of each node object in memory. An object's value slot is offset 2 words from the start of the object. of the object is stored.

```
M[N4+2] <-  M[N1+2] + M[N2+2]
M[N6+2] <-  M[N4+2] * M[N5+2]
M[N5+2] <-  M[N2+2] - M[N3+2]
M[N6+2] <-  M[N4+2] * M[N5+2]
```

where `M[<address>]` refers to a memory location. The specialized program will be much faster than the unspecialized program. Speedups can be larger than that indicated by merely counting saved operations and saved pointer chasing because specialization enables further opportunities for optimization. For example, dead code elimination will prevent the computation of intermediate, but useless computations (such as the second line of the above program), and code scheduling and cache planning can further boost performance, especially for superscalar architectures.

Object-oriented simulators don't follow the simple two input model that we've been describing. First, there isn't a "program" to hand the specializer because it is spread out among the objects and classes. Second, object-oriented simulators don't have an obvious first argument that represents a system to be simulated and an obvious second argument that provides the inputs

of the system. Instead, the objects in the environment, and their interconnections, serve as the system description. The inputs for the simulation are either an argument to the initial simulate message or are part of the system before simulation. Our specializer is designed to handle object-oriented simulations. It automatically produces a highly efficient specialized program from an initial object-oriented program (defined by the methods in the objects in the environment) and a partial description of the data the program will run on (defined by the relationships between the objects in the environment and a partial state description of those objects).

Our specializer is designed for compuations where the message passing pattern is determined solely by the structure of the system being simulated, and not by the varying inputs. Such simulation problems include, for example, many kinds of circuit simulations, n-body simulations, network simulations, and simulation based on differential equations. Experiments with our specializer have shown that for this class of simulations, specialization can provide dramatic performance improvements: we have produced specialized programs that are over two orders of magnitude faster than the program they were derived from. Some of this tremendous speedup is an artifact of our particular (rather inefficient) object language; nonetheless, order of magnitude speedups should be the norm for more efficient object languages.

The major contribution of this work is extending program specialization to apply to programs where the data operated on by the program have lifetimes longer than that of the running program, which is standard for object-oriented programs in an object-oriented environment (or in a persistant environment). This contribution allows program specialization to be applied to object-oriented simulations that operate in an object-oriented environment. Another contribution is that, when the specializer knows precisely which objects the specialized program will access and mutate at runtime, by hardcoding access to the these objects directly into the program itself the specializer will save variable references and pointer chasing that would have occured with a conventional specializer.

This paper has six sections. The first section describes a simple object-oriented language that will be used throughout the rest of the paper. A discussion of object-oriented simulation and the benefits of of specialization appears in Section 2. The next section shows how specialized programs are produced. The limitations of specialization are discussed in Section 4. The following section highlights related research. The closing section concludes and discusses future work.

1 A Simple Object Oriented Language

All the examples in this paper use a very simple, yet powerful, object-oriented language. The actual language of our system is much richer than that shown here,[1] but for expository reasons a simplified language is presented. We assume familarity with the basics of object-oriented languages and object-oriented programming.

Our object language includes class definitions and method definitions (Figure 3). Method names are conventionally preceded by a colon (`:`). Message sending is part of the syntax, there is no special "send" keyword. For example, to send the `:speed` message to a spaceship `ss`, one would write the statement `ss :speed`. To create an object of a given class, one sends a `:create` message to the class, which allocates storage and then runs the `:create` method.

The instance variables of a class can be de-

[1] Our experiments were performed in an object-oriented version of Scheme, a dialect of Lisp. The specializer itself operates on Scheme programs that support object-oriented programming. There is only one important respect in which the pseuducode of the examples is not a faithful redendition of the experimental code: the pseudocode is class based, whereas the experimental code is prototype based. This distinction isn't important to the major point of this paper, nor does it affect our results.

```
Class Definition:
  DEFINE-CLASS class-name
    SUPERCLASSES: class-name*.
    INSTANCE-VARIABLES:
      {identifier : expression}*
    METHODS:
      :method-name formal-parameter* =
         stmt*.
      ...
    END-CLASS

Standalone Method Definition:
  DEFINE-METHOD class-name
    :method-name formal-parameter* =
       stmt*
    END-METHOD

Standalone Variable Definition
  DEFINE-INSTANCE-VARIABLE
    class-name identifier: expression.

Sending a Message:
  object :message-name a1 a2 ... an

Defining a Block:
  [formal-parameter* | stmt*]

Object Creation:
  class-name :create initial-values*

Defining a Global Variable
  DEFINE identifier exp
```

Figure 3: Highlights of the Object Language. When a field of a class definition has no elements, the keyword associated with the field can be elided. When a block has no formal parameters, the `|` character can be elided. The characters `*`, {, }, and `...` are part of the meta-notation. All other characters are part of the language. Identifiers may have embedded dashes (`-`) in them, as well as exclamation points (`!`) and question marks (`?`).

clared outside of the class definition itself, just as methods can be. This feature of the language promotes even greater program modularity, as applications that require extra slots (variables) in a class can add them without the designer of the class, or other users of the class, being aware of the slots (except for possible name clashes). Adding slots in the traditional way, *i.e.*, by subclassing and inheritance, is not always possible because subclassing always defines a new (sub)type.

Our language contains *block*s, a feature familiar to Smalltalk and Self programmers, but less familiar to C++ programmers. A block evaluates to a function that is treated as any other object: it can be an element of a larger object, and passed to (and returned from) methods. The syntax for blocks is `[formal-parameter* | stmt*]`. For example, the block that adds the squares of its arguments is `[x y | (x * x) + (y * y)]`. Blocks are applied to actual parameters by sending them the `:value` message with the parameters as arguments. For example, assuming that the above block is bound to the variable `sos`, it is invoked by `sos :value 8 9` to yield 145. Blocks can be extended to handle other kinds of messages. For example, iteration can be performed using the `:while` or `:repeat` messages to blocks, as in Smalltalk.

The object system is extensible at runtime. That is, as the programmer interacts with and tests her program, she is free to add methods anywhere in the class hierarchy, add instance variables to existing classes, and to add to the class hierarchy.

Unfortunately, this flexibility, which simplifies system building, confounds conventional mechanisms for efficiently implementing method and variable lookup because the compiler does not know the layout of a class. A large runtime cost results from the need for a full dispatch to be performed on each message send. Rigid and inflexible programming environments, such as that provided by C++, are able to make method lookup be very rapid because the compiler knows

the full structure and layout of each class and of the class hierarchy.

Our specializer produces programs at a very low level, lower than the level of message sends. This low level exploits the representation of objects to be as efficient as possible. For the purpose of this paper, assume that an object of n data elements is represented by $n+1$ words of consecutive memory. The first word points to the class descriptor for the object, and each subsequent word points to an object member. Small objects, such as integers and floats, which fit into a word, are not pointed to, but are instead immediate quantities.

2 Example: Digital Circuit Simulation

We will use an object-oriented design and simulation system for digital logic design as the basis of our concrete examples. This system contains many components and tools; we will concentrate on the digital simulator (Figures 4 through 7). It operates by sending messages to the gates to retrieve and update their state. The result of simulation is a change in state in the gates in the database. When simulation is over, other tools display and analyze the result by sending messages to the gates.

The components of a design are all objects. The basic objects are wires and gates (Figure 4). Specific types of gates are specified as subtypes of generic gates (Figures 5 and 6). In our style of circuit specification, the basic classes, and their subtypes, only describe circuit structure, not circuit behavior (*i.e.*, how circuits are simulated). Different simulators can be written without changing the basic structural definitions.

Larger designs (circuits) are created by composing smaller designs (circuits). Circuits are interactively created and interrogated by sending messages. For example, consider the following session, where user input is preceded by ">", and comments are preceded by ";".

```
DEFINE-CLASS wire =
 INSTANCE-VARIABLES:
   consumers: nil.
 METHODS:
  :add-consumer gate =
    consumers :=
      pair :create gate consumers.
  END-CLASS.

DEFINE-CLASS unary-gate =
 INSTANCE-VARIABLES: in-wire, out-wire.
 METHODS:
  :create in out =
    in-wire := in,
    out-wire := out,
    in :add-consumer self.
 END-CLASS.

DEFINE-CLASS binary-gate =
 INSTANCE-VARIABLES:
   in1-wire, in2-wire, out-wire.
 METHODS:
   :create in1 in2 out =
     in1-wire := in1,
     in2-wire := in2,
     out-wire := out,
     in1 :add-consumer self,
     in2 :add-consumer self.
 END-CLASS.
```

Figure 4: Basic Classes for Gates and Wires. These classes provide the basic structure of circuits. The inputs and outputs of gates are wires, and each wire maintains the gates that it provides values to.

```
DEFINE-CLASS inverter =
 SUPER-CLASSES: unary-gate.
 END-CLASS.

DEFINE-CLASS or-gate =
 SUPERCLASSES: binary-gate.
 END-CLASS.

DEFINE-CLASS and-gate =
 SUPERCLASSES: binary-gate.
 END-CLASS.

DEFINE-CLASS half-adder =
 INSTANCE-VARIABLES:
   d: wire :create, e: wire :create,
   orgate1, and-gate1, inverter1,
   and-gate2.
 METHODS:
  :create a b s c =
    or-gate1  := or-gate :create a b d,
    and-gate1 := and-gate :create a b c,
    inverter1 := inverter :create c e,
    and-gate2 := and-gate :create d e s.
 END-CLASS.

DEFINE-CLASS adder =
 INSTANCE-VARIABLES:
   s1: wire :create, c1: wire :create,
   c2: wire :create, or, half-adder1,
   half-adder2.
 METHODS:
  :create a b cin s cout =
    half-adder1 :=
      half-adder :create a b s1 c1,
    half-adder2 :=
      half-adder :create cin s1 s c2,
    or  := or-gate :create c1 c2 cout.
 END-CLASS.
```

Figure 5: Gates Built from the Basic Classes and from Other Gates.

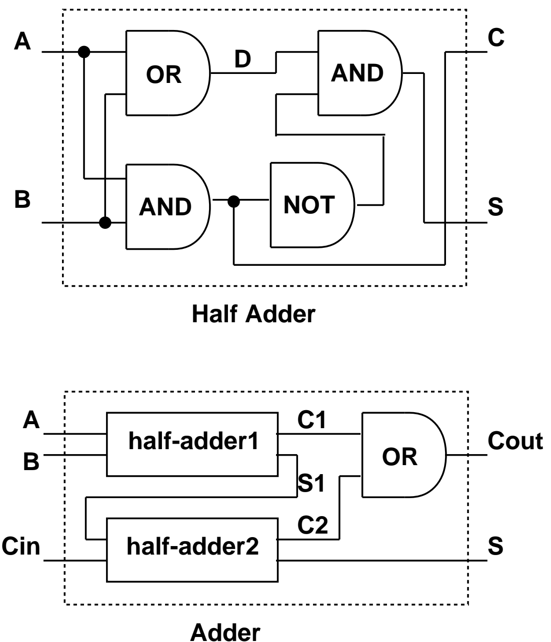


Figure 6: Schematics for Circuits Defined in Figure 5

```
> DEFINE a wire :create.
#<wire 1>
; default print method for objects
; prints them as #<class-name number>.
> DEFINE b wire :create.
#<wire 2>
> DEFINE c wire :create.
#<wire 3>
> DEFINE sum wire :create.
#<wire 4>
> DEFINE carry wire :create.
#<wire 5>
> DEFINE the-adder
    adder :create a b c sum carry.
#<adder 1>
; Now let's interrogate the adder.
DEFINE ha1 the-adder :half-adder1.
<half-adder 1>
> ha1 :orgate1.
#<or-gate 1>
> (ha1 :orgate1) :in1-wire
#<wire 1>
```

```
> a :consumers.
(#<or-gate1>, #<and-gate1>)
```

We specify a simulation model separately from structural models or circuit models (Figure 7). A simulation run consists of setting a circuit's inputs to the desired values, and then sending a `:simulate` message to the worklist, which keeps track of the wires and gates that change before and during a simulation run. The simulation uses a *unit delay* model, where one time unit is required for a gate to compute its output value from its input values. During simulation, gates whose inputs were computed during time unit n are processed during time unit $n+1$ to yield a new set of gates to be processed during time unit $n+2$. Simulation terminates when no gates need processing.

Circuits are simulated interactively. Setting an input node's value adds that node to the worklist. After the inputs are set, simulation is initiated by sending the worklist a `:simulate` message. For example, the following is a continuation of the session transcript shown earlier:

```
> a :update false.
Ready
> b :update true.
Ready
> c :update false.
Ready

> the-simulator :value
"done"
> sum :value.
true
> carry :value.
false
```

Due to message passing, an object-oriented simulation contains considerable overhead. For example, every time a gate G is sent a `:simulate` message, the desired method is retrieved from G's class' parent's method table. This method then sends another message to G to find the desired boolean function. The traversal of the circuit itself requires message passing. On a simulation run, the only useful computation is evaluating a boolean function at each gate and changing the states of wires. All else is overhead.

A specialized program (block) is created by sending a `:specialize` message to a block along with any required argument values, just as for a `:value` message. The `:specialize` method instructs the system to perform a symbolic execution of the block to produce a new, specialized, block with the same behavior as the original block but with much better performance. The arguments to the `:specialize` method may be *symbolic objects* that represent sets of objects. Also, object slots may contain symbolic objects before specialization begins. (Later sections describe symbolic objects in more detail.) In the following transcript the user constructs a specialized simulation program by performing a simulation on *symbolic objects*; the expression `(boolean :screate)` creates a symbolic object that simultaneously represents either *true* or *false*.

```
> a :update (boolean :screate).
Ready
> b :update (boolean :screate).
Ready
> c :update (boolean :screate).
Ready
> DEFINE the-fast-simulator
  the-simulator :specialize.
#<specialized-program 1>

; We now run a simulation as before, but
; instead of sending a message to the work-
; list, we activate the specialized program.
>a :update false.
Ready
>b :update true.
Ready
>c :update false.
Ready

> the-fast-simulator :value.
"done"
> sum :value.
true
> carry :value.
false
```

```
DEFINE-INSTANCE-VARIABLE wire value: false.
DEFINE-METHOD wire :update! new-value =
  value := new-value,
  the-worklist :queue-wire self.
  END-METHOD.
DEFINE-METHOD wire :queue-gates =
  consumers :for-each [oc | the-worklist :queue-wire oc].
  END-METHOD.

DEFINE-METHOD binary-gate :simulate! =
  out-wire :update! (self :b-function (in1-wire :value) (in2-wire :value)).
  END-METHOD.
DEFINE-METHOD unary-gate :simulate! =
  out-wire :update! (self :b-function (in-wire :value)).
  END-METHOD.

DEFINE-METHOD inverter :b-function a   = not(a) END-METHOD.
DEFINE-METHOD or-gate  :b-function a b = or(a, b) END-METHOD.
DEFINE-METHOD and-gate :b-function a b = and(a, b) END-METHOD.

DEFINE-CLASS worklist =
  INSTANCE-VARIABLES: gate-set: set :create, wire-set: set :create.
  METHODS:
   :queue-wire wire = wire-set :add wire.
   :queue-gate gate = gate-set :add gate.
   :simulate =
     if wire-set :empty? then "done"
      else wire-set :for-each [w | w :queue-gates],
           wire-set :empty!,
           gate-set :for-each [g | g :simulate!],
           gate-set :empty!,
           self :simulate.
  END-CLASS.
DEFINE the-worklist (worklist :create).
DEFINE the-simulator [the-worklist :simulate].
```

Figure 7: Digital Simulation Code. This simulator defines a simulation where one time unit is required for a gate to compute its output value from its input values. Unclocked feedback loops are not handled by this simulator. This code is written in object-oriented style. For example, this code introduces the `value` slot into the `wire` class. Also, the generic gates supply the simulation methods, but the specific gates supply the boolean functions. When a gate G is sent a `:simulate` message, the desired method is retrieved from G's class' parent's method table. This method sends another message to G to find the correct boolean function. This programming style is convenient for writing simulators, but results in slow programs. Program specialization removes this overhead to yield fast programs.

For this rather simple example, the specialized program is dramatically faster than the unspecialized program. (For such a small example, concrete speedup figures are meaningless. Figure 8 presents speedup figures for much larger examples.) Because the basis of specialization is symbolic execution, even more highly specialized programs can be constructed by fixing some inputs as true or false. The specializer uses identities such as $x \wedge 0 = 0$ and $x \wedge 1 = x$ to further optimize the programs it produces. The following (abbreviated) transcript shows the production of a specialized program for the `b` input being *false*:

```
> a :update (boolean :screate).
> b :update false.
> c :update (boolean :screate).

> DEFINE the-superfast-simulator
  the-simulator :specialize.
```

Because of the identities listed above, `the-superfast-simulator` will perform fewer than half the boolean operations of `the-fast-simulator`. Similar optimization can be performed for circuits containing state elements, such as registers. In the interest of brevity, we do not show examples of such circuits.

Five of the circuits from ISCA85 [7], a suite of circuits frequently used to benchmark logic simulators, were used to test our system. All the circuits are combinational. We generated 50 input vectors at random, then averaged the results. All timings are in milliseconds on a 32MB HP9000/835. We achieved some impressive speedups (Figure 8).

These speedup figures should be interpreted carefully, as they are particular to our object language and the experiment that we ran. By varying different factors, we could inflate or deflate the speedup figures. We have identified three factors that contribute to speedup: large amounts of overhead in our naive object language implementation, folding of the simulator and data together, and artifacts of specializing digital simulation. Of these factors, evidence indicates that folding the simulator and data together can reliably produce order of magnitude of speedups [1, 12, 18].

Using a less naive implementation of our object-oriented language would have reduced the overhead and lowered the speedup figures by a factor of two or three. There were several cases where we could have eschewed object-oriented programming style for more conventional programming style, but felt that optimization is the perview of the compiler/transformation system. (If one isn't going to use object-oriented methods because they are too costly, then one isn't getting the full benefit of object-oriented programming.)

Speedup is dependent upon the message passing overhead. In a system where much computation is performed between message sends, the relative overhead of sending messages is decreased, and the speedup will be lowered. Our experiments with digital circuit simulation are probably extreme in the small amount of work that is actually performed between message sends. (A pure object-oriented language doesn't offer the opportunity to compute without message passing, and will be tremendously accelerated by our methods.)

Our speedup figures are also boosted by the particular simulation problem we are investigating. The specialized program directly produces the final state of the circuit. During a conventional simulation, many unused intermediate states are produced. For example, consider the waveforms at the outputs of a combinational circuit. They may rise and fall many times before settling, but only their final values are important. The specialized program only computes the final value at each gate; it doesn't compute intermediate values that don't contribute to the final state. A similar phenomenon occurs at the inputs of D flip-flops (elements that capture and retain the state of their input wire at a given instant). Their input may vary wildly during conventional simulation, but the only value that

Circuit	Gates	Wires	Depth	Unspecialized	Specialized	Speed-Up
C432	164	196	17	15,592	130	120
C499	202	243	18	13,095	170	77
C880	383	443	24	59,640	468	127
C1355	546	587	24	98,421	855	115
C1908	886	913	40	158,102	1550	102

Figure 8: Speedups

matters is the one at the moment when flip-flop is instructed to latch a value. The specialized program will produce just that value, and none of the others (unless they affect some other flip-flop or output of the circuit). Our measurements ([1]) indicate that the elimination of intermediate gate outputs is worth a factor of five in speedup. It is unclear how much of this source of speedup can be found in other kinds of simulations.

Our speedup figures are lower than they could be because our system produces Scheme code, which prevents the production of extremely low-level programs.

Based on our experiments, and other applications of program specialization [9, 5, 14], we believe that order of magnitude speedups are routinely achievable via program specialization. The source of this fundamental speedup is described in the next section, which explicitly shows the transformations that a specializer produces.

3 Producing Specialized Programs

Program specialization produces specialized programs whose operations depend solely on the varying inputs to a computation. All computations based on fixed information, e.g., known objects, the types of objects, and the relationships between objects, are performed once during specialization, and are not performed by the specialized program. For a wide range of object-oriented simulations, large amounts of overhead can be saved by executing the overhead at specialization time, rather than at runtime. Besides saving on message passing overhead, specialization also saves the overhead of intermediate object construction. Often, an intermediate object may be constructed whose sole purpose is to transmit several objects from one region of the program to another region. Once the information flow is found by a specializer, it hardcodes the information flow without building an intermediate data structure, *c.f.* Section 3.2. (As shown in [5], eliminating intermediate objects can produce large amounts of instruction level paralellism.)

The heart of the specializer is a symbolic interpreter that runs a program in the presence of partially known data, called *symbolic objects*. Wherever the program can be executed, it will be (as in the lookup and application of methods). Where it cannot be executed, instructions are issued to be executed at runtime. The result of symbolic execution is (a graphical representation of) the specialized program. A separate program transforms the graphical program into an executable program that is returned as the result of specialization (Figure 9).

3.1 Symbolic Objects

We represent partially known information by *symbolic objects*. A symbolic object represents a set of objects, such as the set of integers, the set of reduced rational numbers whose denominator is 3, or the set of star ships captained by James T. Kirk. Even when all the slots of a symbolic object S are known objects, such as the num-

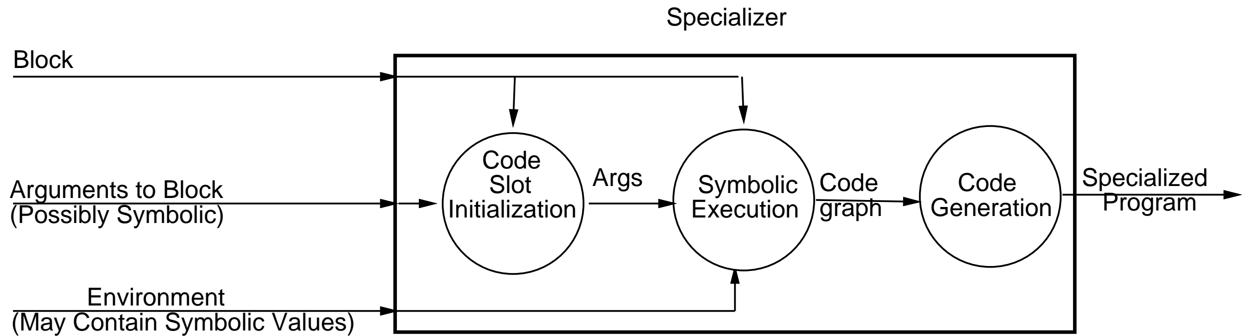


Figure 9: Overview of Specialization. The user symbolically describes the information that is to be known or unknown during specialization. This is accomplished by creating and using *symbolic objects*, which are described in Section 3.1. The specializer is then invoked on a block by sending the block a `:specialize` message along with the arguments of the call. The specializer operates in three phases. The first phase (described in Section 3.4) initializes symbolic objects with information that is used by the code generator. The second phase (described in Section 3.3) symbolically executes the program being specialized. It produces a graphical representation of the specialized program. The third phase is the code generator (described in [19]), which produces executable code from the graphical program representation. The output of the third phase is the output of the entire specialization process.

bers 3 and 4, S still represents a set of objects, namely, all the different objects whose slots will contain those objects at runtime. (Such distinctions are important for deciding object identity during specialization.) Symbolic objects are first class entities during specialization: they can be members of objects, passed to blocks, and returned by blocks. The user of the specializer employs symbolic objects to tell the specializer the information it is allowed and not allowed to use when symbolically executing a program. Before specializing a program, the user replaces slots in objects with symbolic objects in order to "blank out" from her simulation model that information which will vary at runtime. Any information not blanked out may be used during specialization.

Symbolic objects are created by the user just like other objects, except that the `:screate` message creates them, rather than the `:create` message. For example, a symbolic floating point number is created via `floating-point :screate` and a symbolic boolean is created via `boolean :screate`. For simplicity, the functions `a-floating` and `a-boolean`, and their obvious cousins, exist to save typing extra characters. There is a special class `any-class` that is at the top of the type hierarchy. To create a symbolic object that could represent any object at all, one uses the function `a-object`. Symbolic versions of non-scalar objects are created in the same way. For example, `rectangular-complex-number :screate 3.0 a-object()` creates a symbolic rectangular complex number whose imaginary part could be anything. Any non-scalar object created during specialization is a symbolic object, because it represents all the objects that could be created by the specialized program at that creation site.

The important distinction between symbolic objects and *real objects* (that is, non-symbolic objects) is that during specialization, the identity a real objects can be exploited, because the specializer knows that the runtime program will see precisely that object. The identity of a symbolic object cannot be exploited during specialization, because it is unclear pre-

cisely which object it represents. For example, the real object created by the incantation `rectangular-complex-number :create 3.0 a-object()` is distinct from the symbolic object `rectangular-complex-number :screate 3.0 a-object()`. All that is promised is that whatever object the latter is, it will be a `rectangular complex number` whose `real` slot contains 3.0.

A symbolic object has a code attribute that contains the instructions necessary to create at runtime the object that it represents. During specialization, a primitive operation (such as `+` or an object slot access) that encounters real objects (or symbolic objects with enough information) proceeds normally, returning a known result. An operation that encounters symbolic objects containing insufficient information postpones its execution until runtime by returning as output a new symbolic object whose code attribute contains the operation along with its arguments. Because these arguments may themselves be symbolic objects that contain unexecuted instructions and their arguments, we can view the code attribute of a symbolic object as a tree. The code attribute of symbolic object bound to a formal parameter is just the name of the formal parameter; such symbolic objects (and constants) form the leaves of the code tree (Figure 10).

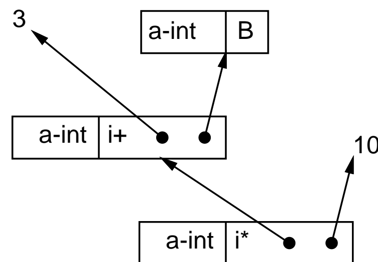
Block to be specialized:

```
DEFINE block =
  [a b c |(a :+ b) :* (a :+ c)]
```

Invocation to produce specialization:

```
DEFINE sblock =
  block :specialize
    3 (integer :screate) 7
```

Symbolic execution yields:



Code generation yields:

```
[a b c | (3 i+ b) i* 10]
```

Figure 10: A Simple Specialization Example. Symbolic objects are represented by rectangles whose left halves are the type represented by the symbolic object, and whose right halves are the code to produce a object at runtime. The functions `i+` and `i*` stand for integer addition and integer multiplication, respectively.

3.2 Specialization in Object-Oriented Environments

The novel aspect of this research is specializing a program with respect to objects that exist before the program runs (called *extant objects*), rather than just with respect to objects that the program creates (called *constructed objects*). The major issue is object mutation: mutation of extant objects must be undone after specialization and must be performed by the specialized program, whereas mutation of constructed objects doesn't occur at runtime because the effects of mutation are captured by simply constructing an object containing the correct values.

A constructed object is one created during specialization. All operations on constructed objects occur as they would at runtime. The code for runtime construction of a given constructed object may or may not appear in the specialized program. If the constructed object lived out its usefulness during specialization, there will be no need to construct it at runtime. For example, consider a program that adds up a list of complex numbers, where complex number are represented by objects with two slots. Assume that the program operates by serially adding the elements of the list together. Adding an n element list will result in $n-1$ additions and $n-1$ com-

plex numbers being produced. The only complex number of interest is the last one. The intermediate complex numbers only serve to shuttle their components from addition to addition. Consider specializing this program for a list 4 long that is known to contain complex numbers (but not the actual numbers themselves). Specializing the program on this input will produce a program that doesn't construct any of the intermediate complex number objects, it only needs to construct the final one (an example appears below).

Extant objects are all the objects (both real and symbolic) that are constructed prior to specialization. They exist before specialization and represent objects that will exist before the program runs. Extant real objects represent themselves, whereas extant symbolic objects represent sets of objects. The gates describing our digital circuit are extant (and real) objects. The purpose of the specialized program is to change the state of extant objects. During specialization, as symbolic execution proceeds, all observations of an extant object occur as they would at runtime. Mutations of extant objects also occur during specialization as they would at runtime, but those locations that are modified, as well as their pre-specialization contents, are remembered. When specialization ends, the state of each extant object is restored to its pre-specialization state, because the creation of a specialized program (which hasn't run yet) can't affect the state of the system. Instructions are added to the specialized program to update the locations to the objects computed for them during specialization. (For extant real objects, this update code directly refers to memory. For extant symbolic objects, the code employs a pointer operation.) Due to the structure of the specializer, the order of mutations (side-effects) to extant objects need not be preserved in the specialized program. Its only contract is to make sure the final state is correctly constructed.

As an example of the distinction between extant and constructed objects, we present the result of specializing some complex number rou-

```
DEFINE-CLASS complex-number =
  METHODS:
   :+ c2 =
      rect :create
        (self :real :+ (c2 :real))
        (self :imag :+ (c2 :imag)),
   :* c2 =  polar :create
              (self :magnitude :+
                    (c2 :magnitude))
   ....
  END-CLASS.

DEFINE-CLASS rect =
  SUPERCLASSES: complex-number.
  INSTANCE-VARIABLES: real, imag.
  METHODS:
    :create r i = ...
    :magnitude = ...
    :theta = ...
  END-CLASS.

DEFINE-CLASS polar =
  SUPERCLASSES: complex-number.
  INSTANCE-VARIABLES: magnitude, theta.
  METHODS:
    :create m t = ...
    :real = ...
    :imag = ...
  END-CLASS.
```

Figure 11: A complex number class

tines from a complex number class (Figure 11). In this example, we specialize a block that adds three complex numbers. We provide the specializer with three extant objects: one fully known complex number, one partially known real complex number, and one partially known symbolic complex number. During specialization, two constructed complex numbers are produced, only one will be constructed by the specialized program.

We create the specialized block as follows:

```
DEFINE real1 =
  rect :create a-float() a-object().
DEFINE symbolic1 =
  rect :screate a-float() 7.2.
DEFINE rect1 =
  rect :create 3.0 -4.0.
DEFINE block =
  [c1 c2 c3 | c1 :+ c2 :+ c3 ].
DEFINE specblock =
  block :specialize
    symbolic1 rect1 real1.
```

The specialized block `specblock` is:

```
[c1 c2 c3 |
  %alloc(%rect-class-descriptor,
         (<c1+1> f+ 3.0) f+ <3AE4>,
         %fp-addition(11.2,<3AE5>))]
```

The function `%alloc` allocates storage and initializes its contents. In this case it allocates a block three words long that points to a class descriptor and contains two floating point numbers. The identifier `%fp-addition` stands for a low level function that performs floating point addition where the type of its first argument is known to be a floating point number. (In reality, this function would be inlined for further speed, but doing so here would unnecessarily complicate the example.) The notation `<exp>` means evaluate `exp` to yield a memory address, then get the contents of this location. The identifier `f+` stands for floating point addition.

This specialized function can be read as follows: Add 1 to C1 (which points to an object) to get the address of its `real` slot. Use the hardware's floating point addition instruction to add the number at that location to 3.0. Then add that number (again using the hardware floating point instruction) to the contents of location 3AE4 to produce the value V_1. The address 3AE4 is hardwired because the specializer knew precisely which object would be accessed at runtime. Then add 11.2 and the contents of 3AE5 (which may be a pointer to another object) to produce value V_2. Then create a rectangular complex number object containing V_1 and V_2, and return this object.

Note the different treatment of the different kinds of objects. The known `rect1` was fully consumed during specialization, and need not be provided at runtime. The partially known real object `real1` was not fully consumed during specialization, but the location of the item that needs to be supplied at runtime is hardcoded in the specialized program. (When the object-oriented system uses a copying garbage collector, this address must be updated after each garbage collection.) The partially known symbolic object `symbolic1` also wasn't fully consumed, so its slots are accessed at runtime. But, because the specializer didn't know precisely which object would be provided at runtime, pointer arithmetic using the supplied object is performed to find the needed information at runtime. Although two complex numbers were produced during specialization, only one is part of the output, and only it is produced by the specialized program.

It's hard to see where the specialized routine could be further optimized. The specializer exploited considerable information to build this routine. Not only does the specializer perform method lookup and application ahead of time, enabling object-orient methods to compete with conventional methods on performance, but it goes further and builds highly specialized routines that no programmer would care to write because they are so specific and highly tuned. This code is far better than a conventional compiler technology can produce, because conven-

tional compilers aren't provided with (and can't exploit) the amount of information that specializers are provided.

3.3 Symbolic Execution

We now show in detail how the specializer produced the specialized block, specblock, shown earlier. The specializer first initializes the code attributes of all symbolic objects. It then begins its symbolic execution of the program.

First the :+ message is sent to c1, the desired method is found, and is then applied. Because these operations occur during specialization, they will not occur at runtime. The addition method then sends messages to retrieve the real portions of c1 and c2. These are then added, which requires another message send to find the correct addition method. This method (which is not shown) decides to use the floating-point addition operator. This operation cannot be performed during specialization, so a new symbolic object S_1 is produced whose code attribute is the floating-point addition f+ operator applied to <c2+1> and 3.0. This is the first time that code is produced. All the overhead to get to this point does not appear in the specialized program, a substantial savings. Next, the imaginary parts are retrieved and added, resulting in the floating point number 11.2. The symbolic object S_1 and 11.2 are combined into a new complex number C. At this point, the code generator kicks in, and produces the code for specblock from the code tree of Figure 12. The intermediate complex number C is not constructed by the specialized program because it is unneeded.

Figure 12: The Result of Symbolic Execution. Symbolic execution produces a rectangular complex number. Its code attribute, and the code attributes of the symbolic objects it (transitively) points to, are used to construct the specialized program. The symbolic objects outlined in heavy lines were provided by the user to instruct the specializer as to which information was known and unknown. The other symbolic objects were constructed during specialization. The construction of complex number C is not part of the specialized program: it lived out its usefulness during specialization. The function a+ represents address addition.

The complex number C is sent the :+ message with c3 (ie, real1) as its argument. As before, the real portions of C and c3 are retrieved and added. The addition routine can determine that the floating-point addition operater can be used. Applying this operator yields a new symbolic object S_2. When adding the imaginary portions, symbolic object S_3 is produced. However, the floating point addition operator cannot be used because the runtime type of the imaginary part of c3 is unknown. Therefore, the f+ operator is not issued, and a routine must be run to perform the addition. (For simplicity, we show this routine as a function call. In actuality, the function is inlined in the specialized program.) Then, S_2 and S_3 are combined into a rectangular complex number which is returned as the result specialization. Figure 12 shows the result of symbolic execution.

3.4 Initializing Code Attributes

The simplicity of the specializer depends upon properly initialized code attributes. Once these are set up, the specializer can uniformly handle all objects. In our earlier simple example, the initial code attributes of the symbolic objects were just the formal parameters to which the symbolic objects were bound. The invariant to be maintained is that code is only generated (*i.e.*, a new symbolic is produced) when an operation cannot be performed for lack of information. In particular, we want to freely access the elements of objects even when the elements (or their subelements) are symbolic objects. For example, when specializing the complex number example, the `:real` message to each complex number was handled identically, by getting the contents of the real slot. That different code was produced for accessing the different kinds of objects was a result of the actions of the initializer, not of the symbolic executor.

Before symbolic execution begins, the specializer initializes the code attributes of the symbolic objects. A symbolic object's code attribute must always contain the code necessary to produce the object at runtime. Therefore, for symbolic objects that are not embedded in larger objects, their code slot is simply the formal parameter that they are bound to. The code attribute of embedded symbolic objects is the code necessary to extract the object from its containing object. For symbolic objects embedded in real objects, the access code is the location of the slot in memory.[2] For symbolic objects within other symbolic objects, the access code is an the offset from the pointer to the object passed in at runtime.

For example, consider the extant object `real1` that is to be bound to `c3` and the symbolic object `symbolic1` that is to be bound to `c1`. The code attribute initializer will set the code attribute of the symbolic object in the `real` slot of `real1` to `3AE4`, the address of the slot in memory. Similarly, `3AE5` is the code attribute of the symbolic object in the `imag` slot. When the specialized program needs the objects represented by the symbolic objects at runtime, it will simply retrieve them directly from memory, which is considerably more efficient than a full message send. The initial code slots for the symbolic objects are similar, except that the starting address of the object will not be known until runtime. Therefore, addresses must be computed at runtime. The initial code attributes of symbolic objects contained within symbolic objects is of the form `<address-of-containing-object + offset>`. Examples of initial code attributes appear in Figure 13.

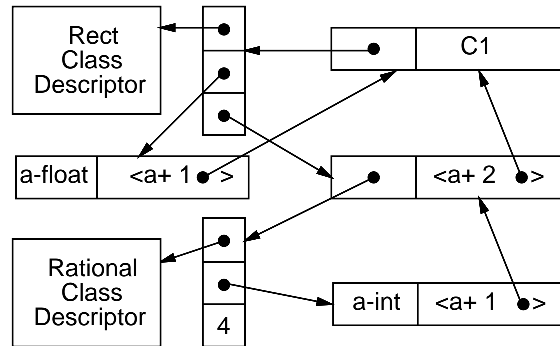


Figure 13: Initial Code Attributes. This figure shows the initial code attributes for the extant object `rect :screate a-float() (rational :screate a-int() 4)` . This figure assumes that the object is bound to the formal parameter $C1$.

4 Limitations

Our implementation of program specialization has some important limitations on the types of programs it can usefully specialize. Our prototype specializer works best on simulations sim-

[2] Our specializer produces Scheme programs that do not have such fine access to memory. Therefore the initial code attribute that our specializer produces is an expression to compute an address. We wouldn't suffer this inefficiency if our specializer produced a lower level program.

ilar to our scenario where the structure of the computation is specified by the structure of the problem and does not vary based on data dependencies. In such cases, the specialized program is largely straightline code with few conditionals and no loops. Although this is a severe limitation, there are still large classes of simulations that benefit from our methods. We are working to remove this limitation, and as we do so, we expect to see similar speedups as we specialize different kinds of programs. Our next goal is to specialize data-dependent event driven simulation where an element is queued for simulation only if its inputs actually changed.

Another limitation is that we don't yet do alias analysis (a topic of current research). This limitation restricts the class of programs for which we produce a useful result. There is no problem when the structure of the computation doesn't depend upon the varying inputs. But a problem can arise when the structure of the computation does depend upon the varying inputs. Correct alias analysis is a prerequisite to producing many specialized programs that contain loops.

Regardless of implementation technique, program specialization works best for simulations where the structure of the system stays constant and only the state changes. Simulations of circuits, networks, and solar systems fall into this class. It does not work well where the structure changes or the computations are extremely data dependent. For example, specialization produces poor speedups for sorting arrays, or for inserting elements into balanced trees (given cache effects, specialization can even produce slowdowns). Similarly, it is difficult to use these techniques for simulations based on linear programming, because the choice of pivot is data dependent.

A program must be respecialized whenever the structure of the problem changes. This is not a problem for simulations that run for a long time, or in applications where multiple sets of input data and initial conditions need to be run before the system structure is changed. The number of times specialized code must be executed to fully amortize the cost of specialization depends on the speedup and the cost of specialization. Based on other experiments not reported here, for our untuned and rather slow specializer, specialized code should be executed at least one hundred times to guarantee overall speedup.

The size of the compiled program can become a problem because loops are expanded at partial evaluation time. Very large datasets and non-linear algorithms result in very large specialized programs. When code size becomes a problem, selected objects and loops should be left intact. We have no automatic method for doing this, and leave such processes to the user.

5 Related Work

Research in partial evaluation, compiled simulation, and the compilation of object-oriented programming systems is relevant to this research.

Program specialization has been investigated as an artificial intelligence tool [2, 10], and is a proven technique for creating compilers and compiler generators [11], rediscovering important algorithms [9], speeding up computations by two orders of magnitude [5], parallelizing scientific code [3], and optimizing programs [4]. The only research of which we are aware that uses partial evaluation to accelerate simulations is presented in [1], which performed compiled simulation for digital circuits. In that work, the simulator was written in conventional style, without objects or object-oriented programming.

Previous researchers have used specific programs, rather than general purpose program transformers to fold together a simulator and a model to produce a specialized program. For example, Lewis [12], and Maurer and Wang [13, 18] have done this for digital circuit simulation. For an overview of the value of folding the simulator and circuit together, see [1].

Perlin's work is most similar to ours because his symbolic executor also produced graphs [17]. His goals were different from ours: he was in-

terested in using his graphs for the incremental recomputation of production systems. It is unclear how his system handled structured values. Most importantly, he never extended his system to handle extant objects.

Program specialization techniques are actively used by Ungar and Chambers [8] to efficiently compile *SELF*, a prototype based object-oriented language. They specialize methods on the fly as methods are first invoked. They have shown rather substantial speedups can be achieved by online specialization. Their research places a heavy emphasis on quickly producing the specialized procedures.

Our research can be viewed as an extension of some of their techniques. The only information that Ungar *et. al.* exploit is the class of an object. This limits the amount of specialization that their system can perform. They get close to making Self programs run on par with optimized C programs. Our specializer makes use of all constant information during specialization, and produces programs much faster than C code, because aggressive specialization builds highly specialized programs that no human would care to write. Our specializer takes longer to run and cannot yet be used on the fly.

Using program specialization to accelerate object-oriented programming was first proposed by the Redfun researchers [2]. (Reading this relatively historical paper is fascinating: in hindsight it is clear that they were trying to describe object-oriented programming, and how to accelerate it. They were years ahead of their time in programming technology and ambition.)

6 Summary and Future Research

We have created a system that greatly accelerates an important class of object-oriented simulations. The use of such a tool removes the tradeoff of performance versus flexibility and ease of design and maintainance. The fundamental method behind the acceleration process is rather simple: introduce the notion of a symbolic object, and then symbolically execute code keeping track of operations that can only be performed at runtime. The embellishment that we make on this fundamental idea is to accurately model the different kinds of objects with respect to their lifetime and identity properties.

Our system achieves spectacular speedups. We have traced these large speedups to three factors: large amounts of overhead in our naive object language implementation, folding of the program and data together, and artifacts of specializing digital simulation. Our belief is that discounting the speedups from factors other than folding the program and data together will still result in order of magnitude speedups for most object-oriented simulations.

The immediate next task is incorporating side effect analysis and alias analysis into the program specializer so that it can specialize more complex object-oriented simulations. For example, specializing data-dependent event-based simulations, an important class of simulations, will require developing such analyses. This is an active topic of research.

References

[1] W.-Y. Au and D. Weise. Automatic generation of compiled simulations through program specialization. In *Proceedings of the 28th Design Automation Conference*, pages 205–210. IEEE, June 1991.

[2] L. Beckman et al. A partial evaluator and its use as a programming tool. *Artificial Intelligence*, 7(4):291–357, 1976.

[3] A. Berlin. A compilation strategy for numerical programs based on partial evaluation. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, July 1989. Published as Artificial Intelligence Laboratory Technical Report TR-1144.

[4] A. Berlin. Partial evaluation applied to numerical computation. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, 1990.

[5] A. Berlin and D. Weise. Compiling scientific programs using partial evaluation. *IEEE Computer Magazine*, 23(12):25–37, December 1990.

[6] A. Bondorf and O. Danvy. Automatic autoprojection for recursive equations with global variables and abstract data types. DIKU Report 90/04, University of Copenhagen, Copenhagen, Denmark, 1990.

[7] F. Brglez, P. Pownall, and R. Hum. Accelerated atpg and fault grading via testability analysis. In *Proceedings of the International Symposium on Circuit and Systems*, 1985.

[8] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programing language. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 146–160, 1989.

[9] C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, 1989.

[10] A. Haraldsson. *A Program Manipulation System Based on Partial Evaluation.* PhD thesis, Linköping University, 1977. Published as Linköping Studies in Science and Technology Dissertation No. 14.

[11] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 1(3/4):9–50, 1988.

[12] D. M. Lewis. Hierarchical compiled event-driven logic simulation. In *Proceedings of ICCAD-89*, pages 498–500, 1989.

[13] P. M. Maurer and Z. Wang. Techniques for unit-delay compiled simulation. In *Proceedings of the 27th Design Automation Conference*, pages 480–484, 1990.

[14] T. Mogensen. The application of partial evaluation to ray-tracing. Master's thesis, DIKU, University of Copenhagen, Copenhagen, Denmark, 1986.

[15] T. Mogensen. Partially static structures. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.

[16] K. Nygaard and O. J. Dahl. The development of the simula languages. In *History of Programming Languages*, pages 439–493. Academic Press, 1981.

[17] M. Perlin. Call-graph caching: Transforming programs into networks. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 122–128, 1989.

[18] Z. Wang and P. M. Maurer. Lecsim: A levelized event driven compiled logic simulator. In *Proceedings of the 27th Design Automation Conference*, pages 491–496, 1990.

[19] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online program specialization. In *Fifth International Conference on Functional Programming Languages and Computer Architecture*, pages 165–191. Springer-Verlag Lecture Notes in Computer Science, August 1991.