DyC: An Expressive Annotation-Directed Dynamic Compiler for C

Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers

Department of Computer Science and Engineering
University of Washington

Technical Report UW-CSE-97-03-03
Last Update: May 12, 1999

http://www.cs.washington.edu/research/dyncomp/
{grant,mock,matthai,chambers,eggers}@cs.washington.edu

Abstract

We present the design of DyC, a dynamic-compilation system for C based on run-time specialization. Directed by a few declarative user annotations that specify the variables and code on which dynamic compilation should take place, a binding-time analysis computes the set of run-time constants at each program point in the annotated procedure's control-flow graph; the analysis supports program-point-specific polyvariant division and specialization. The results of the analysis guide the construction of a run-time specializer for each dynamically compiled region; the specializer supports various caching strategies for managing dynamically generated code and mixes of speculative and demand-driven specialization of dynamic branch successors. Most of the key cost/benefit trade-offs in the binding-time analysis and the run-time specializer are open to user control through declarative policy annotations.

DyC has been implemented in the context of an optimizing compiler, and initial results have been promising. The speedups we have obtained are good, and the dynamic-compilation overhead is among the lowest of any dynamic-compilation system, typically 20-200 cycles per instruction generated on a Digital Alpha 21164. The majority of DyC's functionality has been used to dynamically compile an instruction-set simulator. Only three annotations were required, but a few other changes to the program had to be made due to DyC's lack of support for static global variables. This deficiency and DyC's rudimentary support for partially static data structures are the primary obstacles to making DyC easy to use.

Keywords

Dynamic compilation, specialization, partial evaluation, constant folding, run-time code generation, program optimization, dataflow analysis, C language.

1 Introduction

Dynamic compilation offers the potential for increased program performance by delaying some parts of program compilation until run time, and then exploiting run-time state to generate code that is specialized to actual run-time behavior. The principal challenge in dynamic compilation is achieving high-quality dynamically generated code at low run-time cost, since the time to perform run-time compilation and optimization must be recovered before any benefit from dynamic compilation can be obtained. Consequently, a key design issue in developing an effective dynamic compilation system is the method for determining where, when, and on what run-time state to apply dynamic compilation. Ideally, the compiler would make these decisions automatically, as in other compiler optimizations; however, this ideal is beyond the current state-of-the-art for general-purpose programs.

Instead, current dynamic compilation systems rely on some form of programmer direction to indicate where dynamic compilation should be applied. \`C [Engler et al. 96, Poletto et al. 97] and its predecessor `dcg` [Engler & Proebsting 94] take a procedural approach to user direction, requiring the user to write programs that explicitly manipulate, compose, and compile program fragments at run time. These systems offer great flexibility and control to the programmer, but at the cost of significant programmer effort and debugging difficulty.

Alternatively, Fabius [Leone & Lee 96], Tempo [Consel & Noël 96], and our previous system [Auslander et al. 96] take a declarative approach, employing user annotations to guide dynamic compilation. Fabius uses function currying, in a purely functional subset of ML; Tempo uses function-level annotations, annotations on global variables and structure types, and alias analysis on programs written in C; and our previous system uses intraprocedural annotations, also in C. Each of these declarative approaches adapts ideas from partial evaluation, expressing dynamic compilation as run-time offline specialization (i.e., compile-time binding-time analysis and run-time specialization), where static values correspond to run-time state for which programs are specialized. Declarative approaches offer the advantages of an easier interface to dynamic compilation for the programmer (since dynamic optimizations are derived from the annotations automatically, rather than being programmed by hand) and easier program understanding and debugging (since declarative annotations can be designed to avoid affecting the meaning of the underlying programs). However, declarative systems usually offer less expressiveness and control over the dynamic compilation process than imperative systems.

We have developed a new declarative annotation language and underlying run-time specialization primitives that are more expressive, flexible, and controllable than previous annotation-based systems, but are still easy to use. Our system, called *DyC*, supports the following features:

- support for both polyvariant specialization and polyvariant division*, with the degree of specialization for different variables under programmer control,
- intra- (program-point-specific) and interprocedural (function-level) specialization, with the caller and callee separately compilable,
- arbitrarily nested and overlapping regions of dynamically generated code,
- automatic caching, reuse, and reclamation of dynamically generated code, with cache policies under programmer control,

* Polyvariant division allows the same program point to be analyzed for different combinations of variables being treated as static, and polyvariant specialization allows multiple compiled versions of a division to be produced, each specialized for different values of the static variables.

- automatic interleaving of specialization and dynamic execution to avoid unbounded static specialization for terminating programs, with the exact trade-off between speculative specialization and demand-driven specialization under programmer control,
- automatic interleaving of specialization and dynamic execution to delay specialization of some code until the appropriate run-time values have been computed, and
- run-time optimizations, including constant propagation and folding, conditional-branch folding and dead-code elimination, merge splitting, complete loop unrolling, procedure-call specialization, and strength reduction.

The next section illustrates many of these capabilities using an annotated bytecode interpreter as an example. Section 3 provides an overview of the design of the DyC dynamic-compilation system, which is then detailed in sections 4 through 7. Section 5 presents DyC's annotation language. Section 8 describes our experiences with the system, and section 9 compares DyC to related work. We conclude with our plans for future work.

2 Example

Figure 1 presents a simple interpreter like those for the Smalltalk and Java virtual machines [Goldberg & Robson 83, Lindholm & Yellin 97] or the `mipsi` simulator [Sirer 93]. We will use this example to explain DyC's capabilities, to illustrate the conciseness of the annotations, and to demonstrate the steps in DyC's dynamic-compilation process. In boldface are the annotations we added to turn the interpreter into a run-time compiler, i.e. a program that produces at run time an interpreter that is specialized for the particular array of bytecodes.

Note that while the interpreter appears simple, its successful dynamic compilation requires most of DyC's features, many of which are unique to DyC. The example is representative of the structure of a large class of interpreters and simulators that loop over run-time-constant arrays of operations, dispatching on the type of operation.

2.1 Basic Functionality

The main control annotation is `make_static`, whose argument list of variables the system treats as *run-time constants* when run-time execution reaches that point. By default, DyC will apply interprocedural polyvariant division and specialization as needed on all control-flow paths downstream of the `make_static` annotation, until the variables go out of scope*, in order to preserve the run-time constant bindings of each annotated variable. For example, the variable `pc` is annotated as static. DyC specializes code so that, at each program point in the specialized code, `pc` will have a known run-time constant value. The increments of `pc` in the `switch` body do not cause problems, since the value of a run-time constant after an increment is also a run-time constant. The loop head at the top of the `for` loop requires additional work: DyC will automatically produce a separate specialized version of the loop body for each distinct value of `pc` at the loop head, in effect, unrolling the loop completely. (In Figure 1, we have written all run-time constant operations in italics.)

The `@` symbol annotates the contents of the `bytecodes` array as static, implying that the contents of a referenced, run-time-constant memory location is a run-time-constant.† This enables DyC to

```
void interp_program(int bytecodes[], int arg) {
 printf("%d\n", interp_fn(bytecodes, 0, arg));
}

int interp_fn(int bytecodes[], int pc, int arg) {
 unsigned int inst, rs, rt, rd, offset, reg[32];
 make_static(bytecodes,pc:
             p_cache_one_unchecked,eager);
   // bytecodes,pc promoted
 reg[1] = arg;
  for (;;){ // specializable loop-head merge
   inst = bytecodes@[pc++];
   rs = R1(inst); rt = R2(inst); rd = R3(inst);
   offset = IMMEDIATE(inst);
   switch(OPCODE(inst)) {
    case LI:          // load immediate value
     reg[rt] = offset; continue;
    case MUL:
     reg[rd] = (int) reg[rs] * (int) reg[rt];
     continue;
    case SUBI:
     reg[rt] = (int) reg[rs] - offset;
     continue;
    case IF_GOTO:
     if (reg[rs] == reg[rt])
      pc += offset;
     continue;          // specializable merge
    case GOTO:
     pc = offset; continue;
    case COMPUTED_GOTO:
     pc = reg[rs]; continue;   // pc promoted
    case RET:
     return reg[31];
   }
  }
}
```

Figure 1: Simple Bytecode Interpreter

```
int count[N];
#define threshold ...
specialize interp_fn(bytecodes, pc, arg)
 on (bytecodes, pc);
int interp_fn(int bytecodes[], int pc, int arg) {
 unsigned int inst, rs, rt,rd,offset,reg[32],callee;
 if (++count[pc] >= threshold) {
   make_static(bytecodes, pc);
 } else {
   make_dynamic(bytecodes, pc);
 }
 reg[1] = arg;
 for (;;){ // specializable loop-head merge
    ... //same as above
    switch (OPCODE(inst)) {
    ... //same as above
    case GOSUB:
     callee = offset + pc++;
     reg[rd] =
         interp_fn(bytecodes, callee, reg[rs]);
     break;
   }
 }
}
```

Figure 2: Interprocedural and Conditional Specialization

```
    LI r31, #1            # r1 = 1
    LI r2, #0             # r2 = 0
L0: IF_GOTO r1,r2, L1     # if r1 == r2 goto L1
    MUL r31, r1, r31      # r31 = r31 * r1
    SUBI r1,r1,#1         # r1 = r1 - 1
    GOTO L0               # goto L0
L1: RET                   # return result in r31
```

Figure 3: Factorial Interpreter Program

* DyC currently does not continue specialization upwards past return statements, so specialization stops at the end of each function.

† DyC currently does no automatic alias or side-effect analysis, unlike some other systems, so these annotations are necessary to achieve the desired effect.

constant-fold the `switch` branch within each iteration (since `bytecodes`, `pc` and the loaded bytecode are all run-time constants), selecting just one `case` arm and eliminating the others as dead code. The code that manipulates `bytecodes` and `pc` is also eliminated as dead, once the variables' interpretation overhead is constant-folded away.

The `IF_GOTO` bytecode conditionally rebinds the value of `pc`, based on the run-time variable outcome of a previous test. At the merge after the `if`, `pc` may hold one of two possible run-time constant values, depending on which `if` arm was selected. We call merges such as this one, which have (potentially) different incoming values of run-time constants *specializable merge points*. By default, because `pc` is annotated by `make_static`, DyC will apply polyvariant specialization to the merge and all downstream code, potentially making two copies of the merge and its successors, one for each run-time constant value of `pc`. The loop head is another such specializable merge point, which enables the loop to be unrolled as described above. Thus, for an input program that contains a tree of `IF_GOTO` bytecodes, this specialization will produce a tree of unrolled interpreter loop iterations, reflecting the expected structure of a compiled version of the input program. We call the ability to perform more than simple linear unrollings of loops *multi-way loop unrolling*. DyC allows the programmer to specify less aggressive specialization policies for static variables, to provide finer control over the trade-offs between cost and benefit of run-time specialization.

At each of these specializable merge points, by default DyC maintains a cache of all previously specialized versions, indexed by the values of the static variables at the merge point. When a specializable merge point is encountered during run-time specialization, DyC examines the cache to see whether a version of the code has already been produced, and, if so, reuses it. In the interpreter example, the cache checks at the loop head merge have the effect of connecting backward-branching bytecodes directly to previously generated iterations, forming loops in the specialized code. Similarly, the cache checks allow iterations to be shared, if the input interpreted program contains other control-flow merge points. DyC allows the programmer to specify alternative caching policies or even that no caching be used, to provide finer control to the programmer over this potentially expensive primitive.

The `COMPUTED_GOTO` bytecode, which represents a computed jump, assigns a dynamic expression to `pc`. By default, DyC suspends program specialization when the bytecode is encountered, and then resumes specialization when execution of the specialized code reaches this point and assigns `pc` its actual value. As with specializable merge points, each such *dynamic-to-static promotion* point has an associated cache of specialized versions, indexed by the values of the promoted variables. The specializer consults this cache to see whether a previous version can be reused or a new version must be produced.* Again, programmer-supplied policies support finer control over the aggressiveness of dynamic-to-static promotion and the caching scheme to be used at promotion points.

Because DyC performs specialization at run time rather than at compile time, we have the option of choosing when to specialize control-flow paths ahead of actually reaching them during normal program execution. Aggressive *speculative* specialization has the lowest cost, assuming that all specialized paths will eventually be taken at run time. However, it incurs the cost of specializing any path not executed, and can lead to non-termination in the presence of loops or recursion. Alternatively, *demand-driven* specialization only specializes code that definitely will be executed at run time.

```
     ldq r24, 440(sp)      # reg[1] = arg
     ldl r18, 416(sp)
     stl r18, 4(r24)
     fnop

     ldq r24, 440(sp)      # LI r31, 1
     lda r27, 124(zero)
     lda r25, 1(zero)
     addq r24, r27, r27
     stl r25, 0(r27)

     lda r27, 8(zero)      # LI r2, 0
     lda r25, 0(zero)
     addq r24, r27, r27
     stl r25, 0(r27)

L0:  ldl r27, 8(r24)       # IF_GOTO r1, r2, L1
     ldl r25, 4(r24)
     cmpeq r27, r25, r25
     bne r25, L1

     ldl r27, 4(r24)       # MUL r31, r1, r31
     ldl r25, 124(r24)
     mull r27, r25, r25
     stl r25, 124(r24)

     ldl r27, 4(r24)       # SUBI r1, r1, 1
     lda r27, -1(r27)
     stl r27, 4(r24)

     br L0                 # GOTO L0

L1:  ldl r0, 124(r24)      # RET
     ldq ra, 128(sp)
     fnop
     lda sp, 544(sp)
     ret zero, (ra), 1
```

Figure 4: Dynamically Generated Code for Factorial

```
     ldl r1, 416(sp)       # reg[1] = arg
     lda r2, 1(zero)       # LI r31, 1
     lda r3, 0(zero)       # LI r2, 0
L0:  cmpeq r1, r3, r25     # IF_GOTO r1, r2, L1
     bne r25, L1
     mull r1, r2, r2       # MUL r31, r1, r31
     lda r1, -1(r1)        # SUBI r1, r1, 1
     br L0                 # GOTO L0
L1:  or r2, zero, r0       # RET
     ldq ra, 128(sp)
     fnop
     lda sp, 544(sp)
     ret zero, (ra), 1
```

Figure 5: Generated Code After Register Actions

This is typically done by suspending specialization at each successor of a dynamic (non-run-time-constant) branch in the program being specialized, and resuming only when that successor is actually taken. This strategy avoids non-termination problems and unneeded specialization, but incurs the cost of suspension and resumption of specialization. DyC allows the programmer to specify policies to control speculative specialization; the (safe) default introduces suspension points at each specializable loop head.

2.2 Interprocedural and Conditional Specialization

Figure 2 extends the simple single-procedure interpreter to support interpreting programs made up of multiple procedures. It also illustrates several other DyC capabilities, in particular, how it exploits polyvariant division to support conditional specialization, and annotations that support interprocedural specialization.

[*] Each `make_static` annotation is also a dynamic-to-static promotion point, with an associated cache of versions specialized for different run-time values of the newly static variables.

In the modified `interp_fn` routine, a `count` array associates with each `pc` that corresponds to a function entry point the number of times that function has been invoked. In order to apply dynamic compilation only to heavily used functions, the programmer has made the original `make_static` annotation from Figure 1 conditional – specialization occurs only when the invocation count of some interpreted procedure reaches a threshold. At the merge after the `if`, `bytecodes` and `pc` are static along one predecessor, but dynamic along the other. By default, DyC applies polyvariant division to produce two separate versions of the remainder of the body of `interp_fn`. In one, the two variables are static and lead to run-time specialization, as in Figure 1. In the other, they are dynamic, and no run-time specialization takes place; the input is interpreted normally, at no extra run-time cost.

The `specialize` annotation directs the compiler to produce an alternate entry point to the `interp_fn` procedure that is used when its first two parameters are run-time constants. At `interp_fn` call sites, where the corresponding actual arguments are static, a specialized version of `interp_fn` is produced (and cached for later reuse) for the run-time constant values of the actual arguments. The body of the specialized `interp_fn` is compiled as if its formal parameters were annotated as `make_static` at entry. (The callee procedure and each of its call sites can be compiled separately, given a `specialize` annotation in the shared header file.) This specialization has the effect of streamlining the calling sequence for specialized `GOSUB` bytecodes to specialized callees: neither `bytecodes` nor `callee` will be passed in the specialized call, and the specialized interpreter for the target function (i.e., the compiled code for the target function) will be invoked directly. If the callee function is not yet heavily executed, then after entry the `make_dynamic` annotation will turn off specialization for that input procedure; all bodies of infrequently executed procedures will branch to the same precompiled (and unspecialized) version of the interpreter.

2.3 A Compiling Interpreter

Figure 3 presents a program input for the bytecode interpreter. The program computes the factorial of its input, which is assumed to be in register `r1`. Figure 4 illustrates the code produced when the dynamically compiling interpreter executes the factorial bytecode program on a Digital Alpha 21164. Although the actual code produced at run time is executable machine code, we have presented it in assembly language for readability.*

The structure of the run-time-generated code reflects the structure of the bytecode program used as input to the interpreter. The code contains a conditional branch as a result of multi-way unrolling the interpreter loop beyond the `IF_GOTO` bytecode. Following the specialization of the `GOTO` bytecode, a backward branch is generated to the cached specialized loop iteration corresponding to the label `L0`, creating a loop in the run-time-generated code.

Since Figure 4 is obtained by straightforward specialization of the interpreter, each reference to a virtual register in the interpreter results in a load to or a store from the array that implements the registers. Better code could be generated by adding *register actions* to DyC [Auslander et al. 96]. Register actions permit memory locations to be assigned registers through pre-planned local transformations. In this case, elements of the register array, `reg`, can be allocated to registers, because all offsets into the array are run-time-constant, and all loads and stores can be rewritten as direct references to the corresponding registers. Figure 5 shows the result of applying register actions to the dynamically compiled factorial program.

* Result operands are shown in boldface. `ld[l/q]` = load 32/64 bits. `st*` = store. `mul*` = multiply. `lda` = add with 16-bit signed immediate.

3 System Overview

DyC expresses dynamic compilation as run-time specialization. Directed by a few declarative user annotations that specify the variables for which portions of a program should be specialized, DyC's static compiler produces an executable that includes both statically compiled code and a run-time specializer for code that is to be dynamically compiled. Section 4 describes our run-time specializer and its capabilities.

To achieve the fastest possible dynamic compilation, DyC does much of the analysis and planning for run-time specialization during static compile time. An offline binding-time analysis (BTA) determines which operations can be performed at dynamic compile time, and the run-time specializer is implemented by constructing generating extensions (GEs), that is, custom specializers, one for each piece of code to be dynamically compiled. These GEs perform

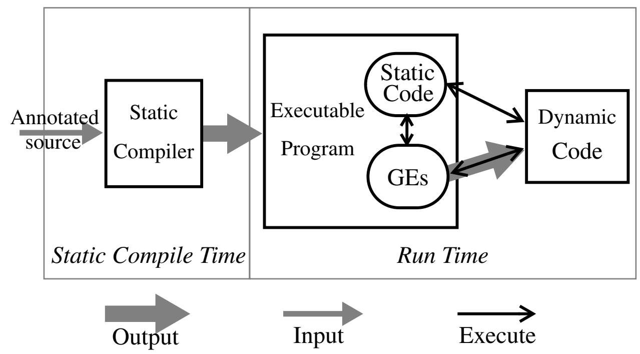


Figure 6: DyC's Static and Dynamic Components

the dynamic compilation when provided the values of the annotated variables. To enable arbitrary interleaving of execution and specialization and arbitrarily overlapping regions of dynamically compiled code (*dynamic code*), DyC is capable of invoking GEs from dynamic code as well as from statically compiled code (*static code*). Figure 6 illustrates the interactions among DyC's compile-time and run-time components.

Figure 7 depicts DyC's organization. We have implemented the binding-time analysis (BTA) and most of the generating-extension construction in the optimizing Multiflow compiler [Lowney et al. 93]. We did so to enable static global optimization of dynamic code with a minimum of restrictions. We believe that performing regular compiler optimizations over both statically compiled and dynamically compiled code is crucial for generating high-quality code.

Our analyses and transformations follow traditional dataflow optimizations, such as common-subexpression elimination, and loop unrolling, because our transformations would otherwise interfere with these optimizations. Unfortunately, these optimizations also interfere with our analyses, mainly by obscuring the intended meaning of the annotations, so some modifications to them were required to preserve information. This issue is discussed further in section 8.1.

Following DyC's core analyses and transformations, Multiflow's combined register allocator and instruction scheduler optimizes the ordinary static code, the static code to be executed by the run-time specializer, and the dynamic code. Modifications to this phase were required to handle run-time constants in the dynamic code, to introduce certain scheduling constraints, and to propagate information to the assembly-code output. Integrate, a post-pass that follows assembly-code generation, integrates the dynamic code

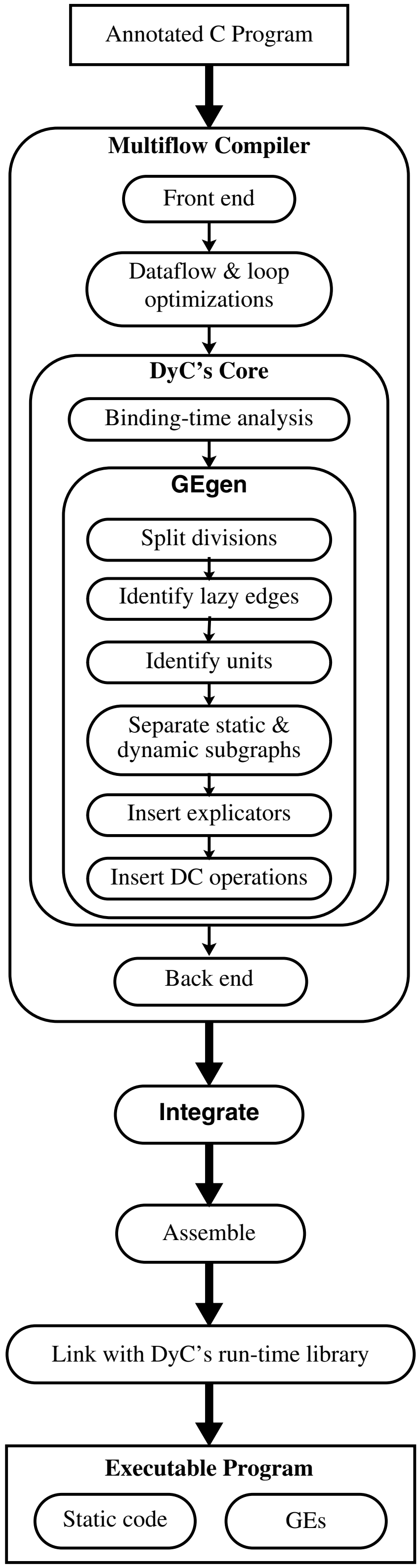


Figure 7: DyC's Compile-Time Phases

into the static specializer code so that the dynamic code is emitted at run time when the corresponding static code is executed by a generating extension. Finally, the resulting code is assembled and linked with DyC's run-time library. The resulting stand-alone executable contains both ordinary, static code and the generating extensions.

```
Specialize(unit:Unit,
           context:Context,
           backpatch_addr:Addr):Addr {
  /* see if we've already specialized this unit for
     this particular context */
  (found:bool, start_addr:Addr) :=
     CacheLookup(unit, context);
  if not found then
    /* need to produce & cache the specialization */
    (start_addr,
     edge_contexts:List<Context>,
     edge_addrs:List<Addr>) :=
       unit.ReduceAndResidualize(context);
    CacheStore(unit, context, start_addr);
    /* see how to handle each successor of the
       specialized unit */
    foreach edge:UnitEdge,
            edge_context:Context,
            edge_addr:Addr
       in unit.edges, edge_contexts, edge_addrs do
      if edge.eager_specialize then
        /* eagerly specialize the successor now */
        Specialize(edge.target_unit,
                   edge_context,
                   edge_addr);
      else
        /* lazily specialize the successor by
           emitting code to compute the values of
           promoted variables and then call the
           specializer with the revised context */
        addr:Addr :=
          edge.ResolvePromotions(edge_context);
        Backpatch(edge_addr, addr);
        patch_addr:Addr :=
          if edge.one_time_lazy
          then edge_addr else NULL;
        Emit("pc := Specialize(`edge.target_unit`,
                               promoted_context,
                               `patch_addr`)");
        Emit("jump pc");
      endif
    endfor
  endif
  /* make the predecessor unit branch to this code */
  Backpatch(backpatch_addr, start_addr);
  return start_addr;
}
```

Figure 8: Run-Time Specializer, Part I

The following sections describe DyC in more detail. We discuss the run-time specializer first, in section 4, in order to specify the functionality of the generating extensions produced by DyC's compile-time phases. Section 5 then presents the annotation language in more detail than in the motivating example in section 2, section 6 describes our BTA, and section 7 details our approach to producing generating extensions from the information the BTA derives, including descriptions of the subphases shown for GEgen. Section 7 also includes a discussion of Integrate.

4 Run-Time Specializer

Our run-time specializer (Figures 8, 9, and 10) is an adaptation of the strategy for polyvariant program-point specialization of a flow chart language described by Jones, Gomard, and Sestoft [Jones et al. 93]. The main process produces specialized code for a *unit* (a generalization of a basic block that has a single entry but possibly multiple exits), given its *context* (the run-time values of the static variables on entry to the unit). The static compiler is responsible for breaking up dynamically compiled regions of the input program into units of specialization, producing the static data structures and code that describe units and their connectivity, and generating the

```
CacheLookup(unit:Unit, context:Context)
          :(found:bool, start_addr:Addr) {
  if CacheAllUnchecked ∈ unit.cache_policies then
    /* always produce a new specialization */
    return (false, NULL);
  else
    /* first index on CacheAll values */
    let cache_alls :=
      elements of context with CacheAll policy;
    (found, sub_cache) :=
      cache.lookup(unit.id, cache_alls);
    if not found then return (false, NULL);
    /* then index on CacheOne values
       in nested cache */
    let cache_ones :=
      elements of context with CacheOne policy;
    (found, start_addr) :=
      sub_cache.lookup(cache_ones);
    /* no need to index on CacheOneUnchecked */
    return (found, start_addr);
  endif
}
CacheStore(unit:Unit, context:Context,
           start_addr:Addr):void {
  if CacheAllUnchecked ∈ unit.cache_policies then
    /* don't store it, since we won't reuse it */
  else
    /* first index on CacheAll values */
    let cache_alls :=
      elements of context with CacheAll policy;
    (found, sub_cache) :=
      cache.lookup(unit.id, cache_alls);
    if not found then
      sub_cache := new SubCache;
      cache.add(unit.id, cache_alls, sub_cache);
    endif
    /* then index on CacheOne values
       in nested cache */
    let cache_ones :=
      elements of context with CacheOne policy;
    /* store the new specialization in the cache,
       replacing any there previously */
    sub_cache.replace(cache_ones, start_addr);
  endif
}
Backpatch(source:Addr, target:Addr):void {
  /* if source != NULL, then backpatch the branch
     instruction at source to jump to target */
}
Emit(instruction:Code) {
  /* append a single instruction to the current
     code-generation point */
}
```

Figure 9: Run-Time Specializer, Part II: Helper Functions

initial calls to the `Specialize` function at the entries to dynamically compiled code.

The `Specialize` function first consults a cache to see if code for the unit and entry context has already been produced (using the unit's caching policy to customize the cache lookup process), and, if so, reuses the existing specialization. If not, the unit's `ReduceAndResidualize` function is invoked to produce code for the unit that is specialized to the input context. The updated values of the contexts at program points that correspond to unit exits are returned. The specialized code is added to the cache (again customized by the unit's caching policy).

Finally, the specializer determines how to process each of the exits of a specialized unit. Each exit edge can either be *eager*, in which case the successor unit is specialized right away, or *lazy*, indicating that specialization should be suspended until run-time execution reaches that edge; lazy edges are implemented by generating stub code that will call back into the specializer when the edge is executed. Points of dynamic-to-static promotion always correspond

```
type Context = Tuple<Value>;
class Unit {
  id:int,
  cache_policies:Tuple<CachePolicy>;
  edges:List<UnitEdge>;
  ReduceAndResidualize(context:Context)
                    :(start_addr:Addr,
                      out_contexts:List<Context>,
                      edge_addrs:List<Addr>);
    /* Take the the values of the static vars and
       produce specialized code for the unit.
       Return the address of the start of the unit's
       specialized code and, for each successor unit,
       the new values of the static variables at that
       edge and the address of the exit point in the
       specialized code for the unit */
}
class UnitEdge {
  target_unit:Unit;
  eager_specialize:bool;
  one_time_lazy:bool;
  ResolvePromotions(context:Context):Addr;
    /* Generate code to extract the current run-time
       values of any static variables being promoted
       at this edge, updating the input
       context and leaving the result in the
       "promoted_context" run-time variable.
       Return the address of the start of the
       generated code. */
}
enum CachePolicy {
  CacheAll, CacheAllUnchecked,
  CacheOne, CacheOneUnchecked
}
```

Figure 10: Run-Time Specializer, Part III: Data Structures

to lazy edges between units; here code is generated that will inject the promoted run-time values into the context before invoking the specializer.

To implement demand-driven specialization, DyC makes lazy the branch successor edges that determine execution of the code that is to be specialized on demand (identification of these edges is described in section 7.1). DyC dynamically overwrites calls to the `Specialize` function placed on these edges with direct jumps to the dynamically generated code for the target units, which achieves a one-time suspension and resumption of specialization at each such point.*

The caching structure for units is one of the chief points of flexibility in DyC. Each of the variables in the context has an associated policy (`CacheAllUnchecked`, `CacheAll`, `CacheOne`, and `CacheOneUnchecked`, listed in decreasing order of specialization aggressiveness), that is derived from user annotations and static analysis. `CacheAllUnchecked` variables are considered to be rapidly changing and their values unlikely to recur, so that there is no benefit in checking and maintaining a cache of specializations to enable code sharing or reuse; each time the unit is specialized, a new version of code is produced, used, and either connected directly to the preceding code or, in the case of dynamic-to-static promotions, thrown away. For `CacheAll` variables, the system caches one version for each combination of their values for potential future reuse, assuming that previous combinations are likely to recur. For `CacheOne` variables, only one specialized version is maintained, for the current values of those variables. If the values of any of the variables change, the previously specialized code is dropped from the cache, assuming that that combination of values is not likely to recur. The values of `CacheOneUnchecked`

* This requires the edge bear no change in cache context and no dynamic-to-static promotions.

variables are invariants or are pure functions of other non-CacheOneUnchecked variables, so the redundant cache checks for those variables are suppressed.

Our run-time caching system supports mixes of these cache policies. If any variable in the context is CacheAllUnchecked, the system skips cache lookups and stores. Otherwise, it performs a lookup in an unbounded-sized cache based on the CacheAll variables (if any); if this is successful, it is followed by a lookup in the returned single-entry cache based on the CacheOne variables, which, if successful, returns the address for the appropriate specialized code. CacheOneUnchecked variables are ignored during cache lookup. If all variables have the CacheOneUnchecked policy, then a single version of the code is cached with no cache key.

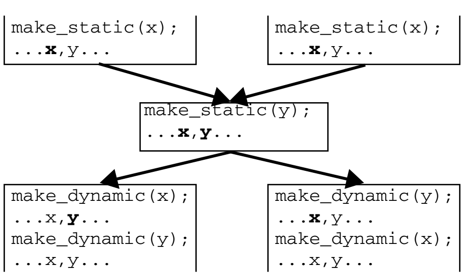
Since invoking the specializer is a source of overhead for run-time specialization, DyC performs a number of optimizations of this general structure, principally by producing a generating extension, which is essentially a specialized version of the Specialize function, for each unit. Section 7 describes these optimizations in more detail.

5 Annotations

Given the target run-time specializer described in the previous section, we now present the programmer-visible annotation language (in this section) and then the analyses to construct the run-time specializer based on the annotations (in sections 6 and 7). Appendix A specifies the syntax of our annotations, expressed as extensions to the standard C grammar rules [Kernighan & Ritchie 88].

5.1 make_static and make_dynamic

The basic annotations that drive run-time specialization are make_static and make_dynamic. make_static takes a list of variables, each of which is treated as a run-time constant at all subsequent program points until DyC reaches either a make_dynamic annotation that lists the variable or the end of the variable's scope (which acts as an implicit make_dynamic). We call the region of code between a make_static for a variable and the corresponding (explicit or implicit) make_dynamic a *dynamic specialization region*, or *dynamic region* for short. Because the placement of make_static and make_dynamic annotations is arbitrary, the dynamic region for a variable can have multiple entry points (if separate make_static annotations for a variable merge downstream) and multiple exit points. A dynamic region can be nested inside or overlap with dynamic regions for other variables, as in the following graph fragment (static variables shown in boldface):



This flexibility for dynamic regions is one major difference between DyC and other dynamic-compilation systems.

A convenient syntactic sugar for a nested dynamic region is make_static followed by a compound statement enclosed in braces, for instance

```
make_static(x, y) {
  ...
}
```

This shorthand places make_dynamic annotations for the listed variables at each of the exits of the compound statement.

5.2 Policies

Each variable listed in a make_static annotation can have an associated list of policies. These policies control the aggressiveness of specialization, division, and dynamic-to-static promotion, the caching policies, and the laziness policies. The semantics of these policies is described in Table 1, with the default policy in each

Policy	Description
poly_divide	perform polyvariant division
mono_divide	perform monovariant division
poly_specialize	perform polyvariant specialization at merges within dynamic regions (specialization is always polyvariant at promotion points)
mono_specialize	perform monovariant specialization at merges
auto_promote	automatically insert a dynamic-to-static promotion when the annotated static variable is possibly assigned a dynamic value
manual_promote	introduce promotions only at explicit make_static annotations
lazy	suspend specialization at all dynamic branches, avoiding all speculative code generation
specialize_lazy	suspend specialization at all dynamic branch successors dominating specializable merge points and specializable call sites, avoiding speculative specialization of multiple versions of code after merges
loop_specialize_lazy	suspend specialization at all dynamic branch successors dominating specializable loop-head merge points and specializable call sites, allowing speculative specialization except where it might be unbounded
eager	eagerly specialize successors of branches, assuming that no unbounded specialization will result, allowing full speculative specialization
m_cache_all_unchecked	specialize at merges, assuming that the context is different than any previous or subsequent specialization
m_cache_all	cache each specialized version at merges
m_cache_one	cache only the latest version at merges, throwing away the previous version if context changes
m_cache_one_unchecked	cache one version, and assume the context is the same for all future executions of this merge
p_cache_none_unchecked	specialize at promotion points, assuming that the promoted value is different than any previous or subsequent specialization
p_cache_all	cache all specialized versions at promotion points
p_cache_one	cache only the latest version at promotion points
p_cache_one_unchecked	cache one version, and assume the promoted value is the same for all future executions of this promotion

Table 1: Policies

category in bold. Annotations in italics are unsafe; their use can lead to changes in observable program behavior or non-termination of specialization, if their stated assumptions about program behavior are violated. All of our default policies are safe, so the novice programmer need not worry about simple uses of run-time specialization. Unsafe policies are included for sophisticated users who wish to have finer control over dynamic compilation for better performance.

The polyvariant vs. monovariant division policy controls whether merge points should be specialized for a variable that may not be static along all merge predecessors. Similarly, the polyvariant vs. monovariant specialization policy controls whether merge points should be specialized for different values of a variable that flow in along different merge predecessors. Promotion points, such as `make_static`, always perform polyvariant specialization of the promoted value, beginning at the promotion point.

The eagerness vs. laziness policies indicate which code should be specialized speculatively or on demand. DyC uses these policies to determine which branch successor edges to make lazy, as described in section 7.1. DyC's default policy is to unroll loops on demand but to specialize other code speculatively, which minimizes the cost incurred by suspension and resumption of specialization, while avoiding unbounded specialization.

The cache policies specified by the annotations determine the cache policies, described in section 4, that govern how the run-time specializer caches and re-uses dynamically generated code. Each policy controls how many specialized versions of code are cached (One vs. All), and whether the values of the static variables are used to determine which cached version to use (checked vs. Unchecked). Our policies currently support either caches of size one or caches of unbounded size. It is reasonable to wish for caching policies that take an argument that indicates the desired cache size. However, bounded multiple-entry caches necessitate a non-trivial cache replacement policy, over which we would want to offer programmer control. More generally, we might wish to provide programmers with direct access to the various caches that the run-time specializer maintains. We leave the design of such interfaces to future work.

The annotations support two sets of cache policies because we frequently desired different policies to be used at the two kinds of program points where new specialized versions were spawned, dynamic-to-static promotion points and specializable merge points. For example, the CacheOneUnchecked policy is useful at dynamic-to-static promotion points when the promoted variable is invariant, but is seldom useful at specializable merge points. Conversely, the CacheAllUnchecked policy is of use primarily at specializable merge points.* Those policies prefixed by `m_` apply at specializable merge points, and those prefixed by `p_` apply at dynamic-to-static promotion points. Section 6.3.6 explains how caching policies are derived at other program points.

5.3 Partially Static Data Structures

Frequently, the result of a memory reference operation (reading a variable, dereferencing a pointer, or indexing an array) is intended to be a run-time constant. This occurs, for example, when manipulating a (perhaps partially) static data structure. By default, the result of a load operation is not a run-time constant, even if its address is a run-time constant. To inform our system that the loaded result should be treated as a run-time constant, the following code can be written:

```
make_static(t);
t = *p;
... /* later uses of t are specialized for t's value */ ...
```

This will introduce an automatic promotion and associated cache check at each execution of the load. If the programmer knows that the result of the dereference will always be the same for a particular run-time constant address, the programmer can use the `p_cache_one_unchecked` annotation:

```
make_static(t:p_cache_one_unchecked);
t = *p;
... /* later uses of t are specialized for t's first value */ ...
```

However, the semantics of this annotation still delays specialization until program execution reaches the dereference point the first time. To avoid any run-time overhead in the specialized code for this dereference, the programmer must state that the load instruction itself is a static computation, returning a run-time constant result if its argument address is a run-time constant. In our annotation language, a memory-reference operation can be prefixed with the `@` symbol, indicating that the associated memory load should be done at specialization time, assuming the pointer or array is static at that point. The programmer can use a static dereference in this example, as follows:

```
make_static(p);
...
t = @* p;
... /* later uses of t are specialized for t's value
           at specialization time */ ...
```

The `@` prefix is a potentially unsafe programmer assertion. Alternatively, we could attempt to perform alias and side-effect analysis to determine automatically which parts of data structures are run-time constants. Unfortunately, it is extremely challenging to produce a safe yet effective alias and side-effect analysis for this task, because the analysis would have to reason about aliasing relationships over the whole program (not just within dynamic regions) and also about the temporal order of execution of different parts of the program (e.g., side-effects that occur when constructing the run-time data structures before the dynamic region is first entered should be ignored). Sound, effective interprocedural alias analysis for lower-level languages like C is an open problem and the subject of ongoing research [Wilson & Lam 95, Steensgaard 96], and so we do not attempt to solve the full problem as part of our dynamic compilation system; our current system includes only simple, local information, such as that local variables that have not had their addresses taken are not aliases of any other expression. When effective alias analyses are developed, we can include them as a component of our system; even so, there may still be a need for explicit programmer annotations to provide information that the automatic analysis is unable to deduce. Other dynamic compilation systems either include an analysis that operates only within a module and rely on programmer annotations to describe the effects of rest of the program (Tempo), disallow side-effects entirely (Fabius), or rely on the programmer to perform only legal optimizations (`C).

Instead of, or in addition to, providing annotations at individual dereference operations, we could provide higher-level annotations of static vs. dynamic components along with variable or type declarations. For example, the variable `p` could be declared with a type such as `constant*` rather than `*`, to indicate that all dereferences would result in run-time constant values; the `bytecodes` array in the initial example in Figure 1 could be declared as `constant int bytecodes[]` to indicate that its contents were run-time constants, thereby eliminating the need for the `@` prefix annotation on the `bytecodes` array index expression. Tempo follows this sort of approach, at least for fields of `struct` types. This syntactic sugar may be a worthwhile addition to DyC.

* The `p_cache_none_unchecked` annotation policy maps to CacheAllUnchecked at promotion points, and implies the dynamically compiled code should be produced, used once, and thrown away.

Currently, the @ annotation does not enable stores at specialization time, and significant extensions to DyC would be required to do so. Some of these extensions are sketched in section 8.3.

5.4 Interprocedural Annotations

Run-time specialization normally applies within the body of a single procedure: calls to a procedure `P` from within a dynamic region or specialized function all branch to the same unspecialized version of `P`. `P` itself may have another specialized region in its body, but this break in the specialization will cause all the different specialized calls of `P` to merge together at the entry to `P`, only to be split back apart again by the cache checks at the `make_static` annotation in `P`'s body. To avoid this overhead, calls can themselves be specialized, branching to correspondingly specialized versions of the callee procedure, thereby extending dynamic regions across procedure boundaries.

The `specialize` annotation names a procedure with a given number of arguments and provides a list of divisions for the procedure. Each division lists a non-empty subset of the formal parameters of the procedure that will be treated as run-time constants; a division can specify the same policies for listed variables as a `make_static` annotation. As described in section 7, for each division, DyC's static compiler produces a code-generation procedure (i.e., a generating extension) for that division that takes the static formals as arguments and, when invoked on their run-time values, produces a specialized residual procedure that takes the remaining arguments of the original procedure (if any), in classical partial-evaluation style.

At each call site in a specialized region to a procedure `P` with an associated `specialize` annotation, DyC will search for the division specified for `P` that most closely matches* the division of actual arguments at the call site (favoring divisions listed earlier in `P`'s `specialize` annotation in case of ties). If one is found, the static compiler produces code that, when specializing the call site at run time, (1) invokes the generating extension for the selected division of `P`, passing the necessary run-time constant arguments, and (2) generates code that will invoke the resulting specialized version for `P`, passing any remaining arguments. Thus, when the specialized call is eventually executed, the call will branch directly to the specialized callee and pass only the run-time variable arguments. If no division specified for `P` matches the call, then the general unspecialized version of `P` is called. Calls to `P` outside any dynamic region continue to invoke the unspecialized version of `P`.

The callee procedure and any call sites can be compiled separately. All that they need to agree on is the `specialize` annotation, which typically is put next to the procedure's `extern` declaration in a header file. Since call boundaries across which specialization should take place are explicitly identified by the programmer, we avoid the interprocedural analysis that would be required to identify (and propagate run-time-constants through) specializable callees.

The `constant` prefix to the `specialize` annotation is an (unsafe) assertion by the programmer that the annotated procedure acts like a pure function; in other words, it returns the same result given the same arguments without looping forever, making externally observable side-effects, or generating any exceptions or faults. DyC exploits this information by calling a constant function from call sites whose arguments are static at specialization time and treating its result as a run-time constant, i.e., reducing the call rather than specializing or residualizing the call. This behavior is different than simply providing a division where all formals are static, since that would leave a zero-argument call whose result was a dynamic value in the specialized code.

We also allow the programmer to prefix individual function calls with the @ annotation to specify that the result of a function call should be treated as a run-time constant if its arguments are run-time constants. For instance, to indicate that a call to the cosine function is a pure function, a programmer could write:

```
make_static(x);
y = cos@(x);
... /* later uses of y are specialized for y's value
        at specialization time */ ...
```

This is a per-call-site version of the `constant` annotation. We included this annotation because the programmer may know, for example, that particular uses of a function will not generate side effects, although the function may produce side effects in general.

5.5 Global Variables

DyC is not currently capable of specializing for the values of global variables. Extensions to the function-annotation syntax to support specialization for global variables would be relatively minor (simply specifying globals in addition to parameters). However, the necessary changes to the rest of the system would be comparable to the support (described in section 8.3) required for permitting static writes to memory.

6 Analysis of the Annotations

Given the programmer annotations described in the previous section, DyC performs dataflow analysis akin to binding-time analysis over each procedure's control-flow graph representation to compute where and how run-time specialization should be performed. The output of this analysis is information associated with each program point (formally, each edge between instructions in the control-flow graph); the domain of the information, `BTA`, along with some constraints on its form, is specified in Figure 11.† This output is used to produce the generating extension which invokes the run-time specializer, as described in section 7.

The analysis essentially reasons only about scalar local variables and compiler temporaries, and annotated data structures are treated as static pointers. The binding times of memory locations are not computed.

The analysis computes a set of divisions for each program point. Each division maps variables annotated as static by `make_static` or `specialize` to their associated policies at that program point. Two divisions are distinct iff there is some variable in one division that is annotated with the polyvariant division policy and is either not found (i.e., it is dynamic) or annotated differently in the other division; divisions that do not differ in the policies of any variables annotated with the polyvariant division policy will be merged together by the analysis.

For each division the analysis computes the following pieces of information:

- The analysis computes the set of static variables (run-time constants) at that program point, including both user-annotated static variables (called *root* variables) and any derived static

* The most closely matching division is the one with the greatest number of formal parameters annotated as static that correspond to static actual arguments and no static formals that correspond to dynamic actuals.

† In our notation, $\rightarrow$ constructs the domain of partial finite maps (sets of ordered pairs) from one domain to another, dom and range project the first and second elements, respectively, of the ordered pairs in the map, and applying a map f to an element in dom(f) returns the corresponding range element. We use $\times$ to construct cross-product domains. We write D(p) to project from the product p the element that corresponds to component domain D, and we write p[$D \rightarrow$v] to compute a new product p' that is like p but whose D element has value v. Pow denotes the powerset domain constructor. Note that $A \rightarrow B \subseteq \mathrm{Pow}(A \times B)$.

Domains:

BTA $\equiv$ Division $\rightarrow$ DivisionInfo

DivisionInfo $\equiv$ StaticVarInfo $\times$ Promotions $\times$ DiscordantVars $\times$ Demotions

Division $\equiv$ Var $\rightarrow$ Policies

Var $\equiv$ *finite set of all variables in scope of procedure being compiled*

Policies $\equiv$ DivisionPolicy $\times$ SpecializationPolicy $\times$ PromotionPolicy $\times$ MergeCachingPolicy $\times$ PromotionCachingPolicy $\times$ LazinessPolicy

DivisionPolicy $\equiv$ {PolyDivision, MonoDivision}

SpecializtionPolicy $\equiv$ {PolySpecialization, MonoSpecialization}

PromotionPolicy $\equiv$ {AutoPromote, ManualPromote}

MergeCachingPolicy $\equiv$ {CacheAllUnchecked, CacheAll, CacheOne, CacheOneUnchecked}

PromotionCachingPolicy $\equiv$ {CacheAllUnchecked, CacheAll, CacheOne, CacheOneUnchecked}

LazinessPolicy $\equiv$ {Lazy, SpecializeLazy, LoopSpecializeLazy, Eager}

StaticVarInfo $\equiv$ Var $\rightarrow$ CachingPolicy $\times$ SourceRoots

CachingPolicy $\equiv$ {CacheAllUnchecked, CacheAll, CacheOne, CacheOneUnchecked}

SourceRoots $\equiv$ Pow(Var)

Promotions $\equiv$ Pow(Var)

Demotions $\equiv$ Pow(Var)

DiscordantVars $\equiv$ Pow(Var)

LiveVars $\equiv$ Pow(Var)

UsedVars $\equiv$ Pow(Var)

MayDefVars $\equiv$ Pow(Var)

Specializations $\equiv$ Proc $\rightarrow$ SpecializationInfo

Proc $\equiv$ *finite set of all procedures in scope of function being compiled*

SpecializationInfo $\equiv$ IsConstant $\times$ Divisions

IsConstant $\equiv$ {Constant, NotConstant}

Divisions $\equiv$ Pow(Division)

Constraints:

BTALegal(bta:BTA) $\equiv$
  LegalDivisions(dom(bta)) $\wedge$
  $\forall$(d,i)$\in$bta.
    StaticVars(i)$\supseteq$dom(d) $\wedge$
    $\forall$v$\in$StaticVars(i).
      (SourceRoots(v, i)$\subseteq$dom(d) $\wedge$
       v$\notin$dom(d) $\Rightarrow$
          CachingPolicy(StaticVarInfo(i)(v)) =
            CacheOneUnchecked) $\wedge$
    Promotions(i)$\subseteq$dom(d) $\wedge$
    DiscordantVars(i)$\subseteq$PolySpecializationVars(d)

LegalDivisions(ds:Pow(Division)) $\equiv$
  $\forall d_1,d_2\in$ds. $d_1=d_2 \vee$ SeparateDivisions(d_1,d_2)

SeparateDivisions(d_1:Division, d_2:Division) $\equiv$
  PolyDivisionVars(d_1)$\neq$PolyDivisionVars(d_2) $\vee$
  $\exists$v$\in$PolyDivisionVars(d_1). $d_1(v)\neq d_2(v)$

PolyDivisionVars(d:Division) $\equiv$
  { v$\in$dom(d) | DivisionPolicy(d(v)) = PolyDivision }

PolySpecializationVars(d:Division) $\equiv$
  { v$\in$dom(d) | SpecializationPolicy(d(v)) = PolySpecialization }

StaticVars(i:DivisionInfo) $\equiv$ dom(StaticVarInfo(i))

SourceRoots(v:Var, i:DivisionInfo) $\equiv$
  if v$\in$StaticVars(i) then SourceRoots(StaticVarInfo(i)(v)) else $\emptyset$

Figure 11: Domains

variables computed (directly or indirectly) from them. The computed set of static variables will be used to determine which computations and operands are static, versus which are dynamic. In addition, it is used to index into the run-time specializer's caches; consequently, the analysis also computes the appropriate caching policy for each static variable. For internal purposes, the analysis tracks the set of annotated run-time constants from which each static variable was computed, directly or indirectly, as described in subsection 6.3.6.

- The analysis computes those points that require dynamic-to-static promotions of variables. Non-empty promotion sets correspond to promotion points for the listed variables. Promotions get inserted after `make_static` annotations for non-constant variables and after (potential) assignments of dynamic values to variables that are annotated with the auto-promotion policy.
- The analysis computes those points that require the *demotion* of variables. The set of demoted variables indicates which previously static variables have become dynamic and need to be initialized with their last static value by residual assignments (called *explicators* [Meyer 91]).
- The analysis identifies which merge points require polyvariant specialization, called *specializable merges points*, because at least one variable that is annotated with the polyvariant specialization policy has potentially different definitions on different merge predecessors. The set of such *discordant variables* is computed at these merge points, and is empty at all other points.

In the remainder of this section we describe the procedure representation we assume and the set of dataflow analyses used to construct this output.

6.1 Procedure Representation

We assume that the procedures being analyzed are represented in a standard control-flow graph, where nodes in the graph can be of one of the following forms:

- an operator node such as a move, add, or call, with one predecessor and successor,
- a merge node with multiple predecessors and one successor,
- a conditional branch node with one predecessor and multiple successors, with a single operand that selects the appropriate successor edge,
- an entry node with no predecessors and a single successor, which acts to bind the procedure's formals upon entry, or
- a return node with one predecessor and no successors, with a single operand that is the procedure's result.

To enable our analyses to detect when potentially different definitions of a variable merge, we assume that merge nodes are annotated with a list of variables that have different reaching definitions along different predecessors, yielding one variable in the list for each ϕ-function that would be inserted if we converted the procedure to static single assignment (SSA) form [Cytron et al. 89].*

* `make_static` annotations must conservatively be treated as definitions of the variables they annotate.

Flow graph nodes are generated from the following grammar:

```
Node ::=        OpNode | MergeNode | BranchNode |
                EntryNode | ReturnNode

OpNode ::=      MakeStaticNode | MakeDynamicNode |
                ConstNode | MoveNode | UnaryNode | BinaryNode |
                LoadNode | StaticLoadNode | StoreNode | CallNode

MakeStaticNode    ::= make_static(Var:Policies)
MakeDynamicNode   ::= make_dynamic(Var)

ConstNode         ::= Var := Const
MoveNode          ::= Var := Var
UnaryNode         ::= Var := UnaryOp Var
BinaryNode        ::= Var := Var BinaryOp Var
LoadNode          ::= Var := * Var
StaticLoadNode    ::= Var := @* Var
StoreNode         ::= * Var := Var
CallNode          ::= Var := Proc(Var, ..., Var)

MergeNode         ::= merge(Var, ..., Var)

BranchNode        ::= test Var

EntryNode         ::= enter Proc

ReturnNode        ::= return Var
```

where Var, Const, UnaryOp, BinaryOp, and Proc are terminals and Policies is as defined in Figure 11.

6.2 Prepasses

Our analyses will need to identify those program points where a variable may be assigned. Direct assignments as part of an OpNode are clear, but assignments through pointers and as side-effects of calls are more difficult to track. We abstract this may-side-effect analysis problem into a prepass whose output is MayDefVars. MayDefVars is a set of variables at each program point that may be modified during execution of the previous node (other than the left-hand-side variable of the node).

Our analyses will work better if they can identify when annotated and derived run-time constant variables are dead. We abstract the result of a live variables analysis into a prepass that computes LiveVars, the set of live variables at each program point. We also compute and abstract a similar analysis, UsedVars, which is the set of variables at each program point that have an earlier definition and a later use (but may temporarily be dead at this point). LiveVars is used to determine when variables can be removed from StaticVarInfo. Because Division contains the policies attributed to annotated variables, a variable cannot be removed from Division when it simply goes dead: when the variable is used again downstream, its policy information will be needed. Hence, UsedVars is used to determine when an annotated variable can be removed from Division.

Finally, we process the interprocedural specialization directives and record them in the Specializations domain. Specializations maps each annotated procedure to a set of divisions given in the `specialize` annotation and indicates whether the procedure was annotated as `constant`. This information is assumed to be replicated at all program points, for convenience in writing the analysis functions.

6.3 The Main Analysis

Figures 12, 13, 14, and 15 define the annotation analysis. The *BTA* family of dataflow equations defines the information on the program point(s) after a node in terms of the information computed for the point(s) before the node (bta), the helper information described in subsection 6.2 for the program point(s) after the node (lvs, uvs, and mds), and the ever-present specialized function information (sp). A solution to the (recursive) dataflow equations is the greatest fixed-point of the set of equations for each node in the procedure, which we solve by simple iterative dataflow analysis; the top element of the lattice, used to initialize back-edges during

BTA_{OpNode}: OpNode→LiveVars→UsedVars→MayDefVars
→Specializations→BTA→BTA

BTA_{OpNode} ⟦`make_static(x:p)`⟧ lvs uvs mds sp bta ≡
Merge(lvs, { (d_{out},i_{out}) |
(d,i) ∈ bta ∧
d_{out} = ForgetDeadVars(uvs, d – { (x',p')∈d | x' = `x` } ∪ {(`x`,`p`)},
StaticVarInfo(i)) ∧
let i' = MakeStatic(`x`,d_{out},i[DiscordantVars→∅]) in
i_{out} = ComputeDemoted(lvs,d_{out}, i, i')})

BTA_{OpNode} ⟦`make_dynamic(x)`⟧ lvs uvs mds sp bta ≡
Merge(lvs, { (d_{out},i_{out}) |
(d,i) ∈ bta ∧
d_{out} = ForgetDeadVars(uvs, d – { (x',p')∈d | x' = `x` },
StaticVarInfo(i)) ∧
let i' = i[DiscordantVars→∅] in
i_{out} = ComputeDemoted(lvs, d_{out}, i, i')})

BTA_{OpNode} ⟦`x := k`⟧ lvs uvs mds sp bta ≡
Merge(lvs, { (d_{out},i_{out}) |
(d,i) ∈ bta ∧
(d_{out},i_{out}) = ProcessAssignment(lvs, `x`, true, ∅, uvs,
mds, d, i)})

BTA_{OpNode} ⟦`x := y`⟧ lvs uvs mds sp bta ≡
Merge(lvs, { (d_{out},i_{out}) |
(d,i) ∈ bta ∧
(d_{out},i_{out}) = ProcessAssignment(lvs,
`x`, `y`∈StaticVars(i), SourceRoots(`y`,i), uvs, mds, d, i)})

BTA_{OpNode} ⟦`x := op y`⟧ lvs uvs mds sp bta ≡
Merge(lvs, { (d_{out},i_{out}) |
(d,i) ∈ bta ∧
(d_{out},i_{out}) = ProcessAssignment(lvs,
`x`, `y`∈StaticVars(i) ∧ Pure(`op`), SourceRoots(`y`,i),
uvs, mds, d, i)})

BTA_{OpNode} ⟦`x := y op z`⟧ lvs uvs mds sp bta ≡
Merge(lvs, { (d_{out},i_{out}) |
(d,i) ∈ bta ∧
(d_{out},i_{out}) = ProcessAssignment(lvs,
`x`, {`y`,`z`}⊆StaticVars(i) ∧ Pure(`op`),
SourceRoots(`y`,i) ∪ SourceRoots(`z`,i), uvs, mds, d, i)})

BTA_{OpNode} ⟦`x := *p`⟧ lvs uvs mds sp bta ≡
Merge(lvs, { (d_{out},i_{out}) |
(d,i) ∈ bta ∧
(d_{out},i_{out}) = ProcessAssignment(lvs, `x`, false, ∅, uvs,
mds, d, i)})

BTA_{OpNode} ⟦`x := @* p`⟧ lvs uvs mds sp bta ≡
Merge(lvs, { (d_{out},i_{out}) |
(d,i) ∈ bta ∧
(d_{out},i_{out}) = ProcessAssignment(lvs,
`x`, `p`∈StaticVars(i), SourceRoots(`p`,i), uvs, mds, d, i)})

BTA_{OpNode} ⟦`*p := y`⟧ lvs uvs mds sp bta ≡
Merge(lvs, { (d_{out},i_{out}) |
(d,i) ∈ bta ∧
(d_{out},i_{out}) = ProcessStmt(lvs, ∅, uvs, mds, d, i))}

BTA_{OpNode} ⟦`x := f(`y_1`,...,`y_n`)`⟧ lvs uvs mds sp bta ≡
Merge(lvs, { (d_{out},i_{out}) |
(d,i) ∈ bta ∧
(d_{out},i_{out}) = ProcessAssignment(lvs, `x`,
{y_1,...,y_n}⊆StaticVars(i) ∧
`f`∈dom(sp) ∧ IsConstant(sp(f)) = Constant,
$\cup_{y_i \in \{y1,\ldots,yn\}}$ SourceRoots(y_i,i), uvs, mds, d, i)})

Figure 12: Flow Functions, Part I

```
BTA_Entry: EntryNode→LiveVars→UsedVars→Specializations→BTA
BTA_Entry ⟦enter P⟧ lvs uvs sp ≡
  let ds = (if P∈dom(sp) then Divisions(sp(P)) else ∅) ∪ {∅} in
    Merge(lvs, { (d, (s, ∅, ∅, ∅)) |
          d'∈ds ∧
          d = ForgetDeadVars(uvs, d', ∅) ∧
          s = { InitialBinding(v, d) | v∈dom(d) } })

BTA_Branch: BranchNode→LiveVars×LiveVars→UsedVars×UsedVars
  →MayDefVars×MayDefVars→Specializations→BTA→BTA×BTA
BTA_Branch ⟦test x⟧ (lvs_1,lvs_2) (uvs_1,uvs_2) (mds_1,mds_2) sp bta ≡
  (Merge(lvs_1, { (d_out,i_out) |
    (d,i) ∈ bta ∧ (d_out,i_out) = ProcessStmt(lvs_1, ∅, uvs_1, mds_1, d, i)}),
   Merge(lvs_2, { (d_out,i_out) |
    (d,i) ∈ bta ∧ (d_out,i_out) = ProcessStmt(lvs_2, ∅, uvs_2, mds_2, d, i)}))

BTA_Merge: MergeNode→LiveVars→UsedVars→MayDefVars
                   →Specializations→Pow(BTA)→BTA
BTA_Merge ⟦merge(x_1, ..., x_n)⟧ lvs uvs mds sp btas ≡
  let bta = ∪ btas in
        Merge(lvs, { (d_out,i_out) |
          (d,i) ∈ bta ∧
          pvs = {x_1,...,x_n} ∩ PolySpecializationVars(d) ∩ lvs ∧
          smvs = {x | x∈{x_1,...,x_n}∧ merge for x is static in division d}∧
          mvs = ({x_1,...,x_n} – pvs – smvs) ∩ lvs ∧
          d_out = ForgetDeadVars(uvs, d – { (x,p)∈d | x∈mvs },
                                  StaticVarInfo(i)) ∧
          si = KillDanglingDerivedVars(d,StaticVarInfo(i),mvs) –
               { (v, vp) ∈ StaticVarInfo(i) | v∈mvs } ∧
          si' = si – { (v, vp) ∈ si | v∈pvs } ∪
                { (v, (mp, {v})) | (v,p)∈d_out ∧ v∈pvs ∧
                                      mp = MergeCachingPolicy(p) } ∧
          i_out = ComputeDemoted(lvs,d_out, i,(si', ∅, pvs, ∅)) } )
```

Figure 13: Flow Functions, Part II

the initial iteration of analysis of loops, is the empty set (no divisions).*

In general, each flow function computes a new, updated set of divisions from the inflowing set(s) of divisions. We remove any permanently dead variables (those no longer in the UsedVars set)† from the set of annotated variables, Division, and any, at least temporarily, dead variables (those no longer in the LiveVars set) from the set of run-time constants, StaticVarInfo, to avoid unnecessary polyvariant division or specialization. Once a new set of divisions and associated information is computed, divisions that no longer differ in the policies of any variables annotated as leading to polyvariant division are merged together into a single division. Thus the degree of polyvariant division can vary from program point to program point.

* We follow the conventions of dataflow analysis in solving for *greatest* fixpoints and initializing information along edges to the *top* of the lattice. In this paper we do not bother to more formally define the lattice ordering and meet operations, since we have given an explicit flow function for merge nodes and defined the top lattice element, and simple iterative or worklist-based analyses need nothing more. A soundness proof for our analysis would of course require a more formal treatment. Since the domain of analysis is finite and each analysis function is monotonic, termination of analysis is assured.

† We do not remove permanently dead variables from Division if any static variables derived from them are still live, because doing so would require us to kill those derived static variables, as described in subsection 6.3.6.

```
Merge(lvs:LiveVars, bta:BTA):BTA ≡
  MergePartitions(lvs, Partition(bta))
Partition(bta:BTA):Pow(BTA) ≡
  { { (d,i)∈bta | DivisionSelector(d) = ds } |
    ds ∈DivisionSelectors(bta) }
DivisionSelectors(bta:BTA):Divisions ≡
  { DivisionSelector(d) | (d,i)∈bta }
DivisionSelector(d:Division):Division ≡
  { (v,p)∈d | v∈PolyDivisionVars(d) }
MergePartitions(lvs:LiveVars, btas:Pow(BTA)):BTA ≡
  { (d,i) | bta ∈ btas ∧
          d = ∩_Division dom(bta) ∧
          i = FilterStaticVars(lvs, d, ∩_DivisionInfo range(bta)) }
FilterStaticVars(lvs:LiveVars, d:Division, i:DivisionInfo
                ):DivisionInfo ≡
  if dom(d) = ∅ then i[StaticVarInfo→∅]
  else let si = { (v, (p,rvs))∈StaticVarInfo(i) |
                  v∈lvs ∪Derived(v,StaticVarinfo(i)} in
    i[StaticVarInfo→
      KillDanglingDerivedVars(d,si,{ r | (v, (p, rvs)) ∈ si ∧
                                          r∈rvs–dom(d) })]
Derived(v: Var, si: StaticVarInfo):Pow(Var) ≡
  {v' | (v', (p', rvs')) ∈si ∧ v ∈rvs'∧ v ≠ v' }
ComputeDemoted(lvs:LiveVars, d: Division, i, i':DivisionInfo
              ): DivisionInfo ≡
  let  svf = StaticVars(FilterStaticVars(lvs, d, i'))
       svi = StaticVars(i), svo = StaticVars(i') in
         i'[DemotedVars→(svi – svo) ∪ (svo – svf)]
InitialBinding(v:Var, d:Division
              ):Var × (CachingPolicy × SourceRoots) ≡
  (v, (PromotionCachingPolicy(d(v)), {v}))
MakeStatic(v:Var, d:Division, i:DivisionInfo):DivisionInfo ≡
  if v∈StaticVars(i) then i
  else (StaticVarInfo(i) ∪ {InitialBinding(v, d)}, {v}, ∅, ∅)
Pure(op:Op):bool ≡
  returns true iff op is idempotent and cannot raise an exception or fault;
  most operators are pure; div and malloc are canonical impure operators
```

Figure 14: Helper Functions, Part I

6.3.1 Entry Nodes

The analysis of the procedure entry node creates the initial division(s), including at least the empty unspecialized division with no run-time constants. For a specialized procedure, each of the divisions listed in the `specialize` annotation introduces an additional specialized division in the analysis. For each division, the set of run-time constants is initialized to the set of annotated variables, with each variable's initial caching policy taken from its specified PromotionCachingPolicy.

6.3.2 `make_static` and `make_dynamic` Nodes

The analysis of a `make_static` pseudo-instruction adds a new static variable to each of the existing divisions, and replaces the policies associated with the variable if it is already present in some division. If the variable was not already a run-time constant in some division, then the `make_static` instruction introduces a dynamic-to-static promotion. The `make_dynamic` instruction simply removes the annotated variable from each of the inflowing divisions; as described above, this may cause divisions to merge and run-time static variables derived from the newly dynamic variable to be dropped.

6.3.3 Assignment and Store Nodes

The various forms of assignment nodes all have similar analyses, dependent only on whether the right-hand-side expression is a run-time constant expression. Compile-time constants are trivially run-time constants. A unary or binary expression yields a run-time constant, if its operands are run-time constants and if the operator is a pure function (e.g., it cannot trap and always returns the same result given the same arguments). A load instruction yields a run-time constant iff its address operand is a run-time constant (which includes fixed values, such as the address of a global or local variable) and it is annotated with `@` by the programmer. A call to a procedure annotated by the programmer as `constant` yields a run-time constant if all its arguments are run-time constants. Since a call annotated with `@` is identical, we have omitted that case. A store instruction has no definitely assigned result variable, only potential side-effects, as described by the MayDefVars set.

The effect of these nodes is summarized into two sets. The first is a (singleton or empty) set of variables definitely assigned run-time constant values; the other is a set of variables possibly assigned dynamic expressions (comprised of the assigned variable if the right-hand-side expression is dynamic, as well as any variables in the MayDefVars set). The definitely static variables are added to the set of run-time constant variables. The possibly dynamic variables are divided into those annotated with the auto-promote policy (which instructs DyC to insert a dynamic-to-static promotion automatically if they ever get assigned a dynamic value), and those that aren't auto-promoted (which DyC drops from the set of annotated variables and the set of run-time constants, if present in either). As with the analysis of any node, dropping variables from the set of annotated variables can cause divisions to merge.

6.3.4 Merge Nodes

The analysis of a merge node must deal with *discordant variables* that have potentially different definitions along different predecessors (these variables were identified by a prepass and stored with the merge node, as described in section 6.2). For those discordant variables that the programmer annotated as run-time constants with a polyvariant specialization policy, the analysis will mark this merge as discordant in those variables, triggering specialization of the merge and downstream code. Any other discordant variables are dropped from the set of annotated variables and run-time constants, if present. (Again, this dropping of variables from the annotated set may cause divisions to merge.) Derived run-time constants are implicitly monovariantly specialized, since they were not explicitly annotated as polyvariantly specialized by the programmer. The caching policy for all discordant variables at the merge is set to those variables' merge caching policy.

This analysis can be improved for the case of a *static merge.* A static merge is a merge where at most one of the merge's predecessors can be followed at specialization time, because the predecessors are reached only on mutually exclusive static conditions. Since only one predecessor will be specialized, the merge node won't actually merge any branches in the specialized code and only one definition of each static variable will reach the merge when the residual code is executed. In fact, all that is required is to ensure that only one definition of a static variable can reach the merge at execution time, either because there is only one reaching definition, or potentially different definitions are only along predecessors with mutually exclusive static reachability conditions. Such variables are not included in the set of discordant variables. Subsection 6.4 describes the reachability analysis used to identify static merges.

```
ProcessAssignment(lvs:LiveVars, v:Var,
                  rhs_is_static:bool, rvs:SourceRoots,
                  uvs:UsedVars, mds:MayDefVars,
                  d:Division, i:DivisionInfo
                  ):Division × DivisionInfo ≡
  if rhs_is_static
  then ProcessStmt({(lvs, v,(CacheOneUnchecked,rvs))}, mds,
                  uvs, d, i)
  else ProcessStmt(lvs, ∅, mds ∪ {v}, uvs, d, i)
ProcessStmt(lvs:LiveVars, static_assigns:StaticVarInfo,
            uvs:UsedVars, dyn_assigns:Pow(Var),
            d:Division, i:DivisionInfo
            ):Division × DivisionInfo ≡
  (d_out,i_out) where
    ps = MayPromotedVars(d, dyn_assigns)
    d' = ForgetDynVars(dyn_assigns – ps, d)
    si = StaticVarInfo(i)
    si' = si – { (v,vi)∈si | v∈dom(static_assigns) } ∪ static_assigns
    si_out = ProcessDynAssigns(
            si', dom(static_assigns), dyn_assigns, d')
    d_out = ForgetDeadVars(uvs, d', si_out)
    ps_out = ps ∩ dom(d_out)
    i_out = ComputeDemoted(lvs, d_out, i, (si_out, ps_out, ∅, ∅))
MayPromotedVars(d:Division, vs:Pow(Var)):Promotions ≡
  { v∈vs | v∈dom(d) ∧ PromotionPolicy(d(v)) = AutoPromote }
KillDanglingDerivedVars(d:Division,si:StaticVarInfo,
                        mvs:Pow(Var)):StaticVarInfo ≡
  { (v, (p,rvs))∈si | (rvs∩mvs)=∅ } ∪
  {(v, (p',rvs')) | (v, (p,rvs))∈si ∧ (rvs∩mvs)≠∅ ∧ v∈dom(d) ∧
                rvs'=(rvs-mvs)∪{v} ∧
                p'=p ∩_CachingPolicy
                  (∩_CachingPolicy v' ∈ rvs∩mvs CachingPolicy(si(v'))) }
ProcessDynAssigns(si:StaticVarInfo, svs:Pow(Var), dvs:Pow(Var),
                  d:Division):StaticVarInfo ≡
  KillDanglingDerivedVars(d,si,(svs∪dvs)) – { (v, vp)∈si | v∈dvs }
    ∪ { InitialBinding(v, d) | v∈dom(d) ∧ v∈dvs }
ForgetDeadVars(uvs:UsedVars, d:Division,si:StaticVarInfo
               ):Division ≡
  { (v,p)∈d | v∈uvs ∨ v∈∪_v' ∈ dom(si) SourceRoots(si(v'))}
ForgetDynVars(vs:Pow(Var), d:Division):Division ≡
  { (v,p)∈d | v∉vs }
```

Figure 15: Helper Functions, Part II

6.3.5 Branch and Return Nodes

The analysis of a branch node simply replicates its incoming information along both successors (as always, after filtering the set of variables to exclude those that are no longer live along that successor). Return nodes need no analysis function, since there are no program points after return nodes, and we do not currently do interprocedural flow analysis of annotations.

6.3.6 Caching Policies and Derivations of Static Variables

At each program point, the analysis computes a caching policy for each variable. This policy is used to control indexing into the run-time specializer's caches of previously specialized code. Annotated variables at promotion points (and at the start of analysis of a division of a specialized function) are given the user-specified PromotionCachingPolicy value. At specializable merge points, a discordant variable is changed to use the variable's MergeCachingPolicy value.

Derived run-time constants are given the CacheOneUnchecked policy. This ensures that unannotated run-time constants are never used in cache lookups and consequently do not lead to additional specialization beyond that explicitly requested by the user. This unchecked caching policy is safe, as long as each derived run-time constant is a pure function of some set of annotated variables. An annotated variable can be assigned a static expression, in which case it is treated (more efficiently) as a derived run-time constant with a CacheOneUnchecked policy, instead of its annotated caching policy.

Assignments to root annotated variables violate the assumption that a derived run-time expression is a function of a set of root annotated variables. In this case, the derived run-time constants need to be dropped from the set of static variables, and annotated derived run-time constants need to be assigned new cache policies; currently, we meet the cache policies of their prior root variables. The analysis tracks the set of root annotated variables, SourceRoots, on which a derived run-time constant depends; whenever a root variable is (possibly) assigned to or is removed from the division, all dependent run-time constants are dropped (or restored to their regular caching policy, if roots themselves). This distinction between root and derived variables is a significant source of complexity in the analysis.

6.3.7 Computation of Demotions

At each program point the analysis computes the set of demoted variables. A variable can be demoted in two ways: (1) if it was static before the point but is dynamic after the point ($svi - svo$ in the equations), or (2) if it becomes static at the node but is dropped from the set of static variables right after the node because of filtering of live variables ($svo - svf$ in the equations).

6.3.8 Additional Lattice Meet Operations

The Merge helper function uses the lattice meet operators for the Division and DivisionInfo domains. The lattice meet operator $\cap_{\mathrm{Division}}$ over elements of Division indicates how to combine different annotations for a set of variables in the same division, and is defined as follows:

$$d_1 \cap_{\mathrm{Division}} d_2 \equiv \{ (v,p) \mid v \in \mathrm{dom}(d_1) \cap \mathrm{dom}(d_2) \wedge p = d_1(v) \cap_{\mathrm{Policies}} d_2(v) \}$$

Elements of Policies are met point-wise. Elements of individual policy domains are totally ordered, with elements listed earlier in the set of alternatives for a domain in Figure 11 ordered less than elements listed later; for example:

$$\mathrm{AutoPromote} \leq_{\mathrm{PromotionPolicy}} \mathrm{ManualPromote}$$

Thus, the lattice meet operator for a particular policy domain returns its smallest argument, for example:

$$\mathrm{AutoPromote} \cap_{\mathrm{PromotionPolicy}} \mathrm{ManualPromote} = \mathrm{AutoPromote}$$

This rule has the effect of picking the strongest policy of any of the merging divisions.

The lattice meet operator $\cap_{\mathrm{DivisionInfo}}$ over elements of DivisionInfo is defined as the pointwise meet over its component domains, which are defined as follows:

$$\begin{aligned}
& si_1 \cap_{\mathrm{StaticVarInfo}} si_2 \equiv \\
& \quad \text{let } si_{new} = \\
& \quad \{ (v, (p,rvs)) \mid v \in \mathrm{dom}(si_1) \cap \mathrm{dom}(si_2) \wedge \\
& \qquad\qquad p = p_1 \cap_{\mathrm{CachingPolicy}} p_2 \wedge \\
& \qquad\qquad rvs = rvs_1 \cup rvs_2 \\
& \qquad \text{where } p_2 = \mathrm{CachingPolicy}(si_2(v)) \\
& \qquad\qquad p_1 = \mathrm{CachingPolicy}(si_1(v)) \\
& \qquad\qquad rvs_1 = \mathrm{SourceRoots}(si_1(v)) \\
& \qquad\qquad rvs_2 = \mathrm{SourceRoots}(si_2(v)) \} \\
& vs_1 \cap_{\mathrm{Promotions}} vs_2 \equiv vs_1 \cup vs_2 \cap \mathrm{dom}(si_{new}) \\
& vs_1 \cap_{\mathrm{DiscordantVars}} vs_2 \equiv vs_1 \cup vs_2 \cap \mathrm{dom}(si_{new}) \\
& vs_1 \cap_{\mathrm{Demotions}} vs_2 \equiv vs_1 \cup vs_2
\end{aligned}$$

6.4 Reachability Analysis

We identify static merges by computing a *static reachability condition* at each program point for each division. A static reachability condition is a boolean expression (in conjunctive normal form) over the static branch outcomes that are required in order to reach that program point. A static branch is a branch whose test variable is identified as a run-time constant by the *BTA* analysis. A static merge is one whose predecessors have mutually exclusive static reachability conditions. A merge is static for a particular variable x with respect to a given division iff at most one possible definition reaches the merge, or different incoming potential definitions are along mutually exclusive predecessors. Reachability conditions are computed at the same time as the *BTA* information, since they depend on the *BTA*'s division and static variable analysis and influence the *BTA* analysis's treatment of merge nodes. Further details on reachability analysis can be found in an earlier paper [Auslander et al. 96][*].

7 Creating the Generating Extensions

Given the output of the *BTA* analysis, DyC statically constructs the code and static data structures that, when executed at run time, will call the run-time specializer with the appropriate run-time-constant arguments to produce and cache the run-time specialized code, i.e., the generating extensions. The following steps, shown in Figure 7, are performed:

- **Split divisions:** The compiler statically replicates control-flow paths, so that each division receives its own code. After replication, each program point corresponds to a single division. Replication starts at entries to specialized functions (producing several distinct functions), and at merge points where different divisions combine. Replicated paths remerge at points where divisions cease to differ and are joined by the Merge function.
- **Identify lazy edges:** The compiler identifies which branch successor edges should be lazy specialization edges. Subsection 7.1 discusses this in more detail. Lazy points due to dynamic-to-static promotions are trivially identified.
- **Identify units:** The compiler identifies the boundaries of the units manipulated by the run-time specializer (described in section 4). Unit boundaries primarily correspond to dynamic-to-static promotion points, *eviction* points (where variables are evicted from the set of annotated variables), specializable merge points, and lazy branch successor edges. The first three cases are cache lookup points, and the last case avoids

[*] Our earlier paper presents the reachability analysis for a monovariant binding-time analysis; the analysis also uses a slightly more conservative rule for determining static merges than the one described here.

speculative specialization. This process is described in more detail in subsection 7.2, below. A clustering algorithm then attempts to merge boundaries together to minimize their cost, as described in subsection 7.3. The `Unit` and `UnitEdge` specializer data structures are generated at the end of this process.

- **Separate static & dynamic subgraphs:** The compiler separates the static operations (OpNodes whose right-hand-side expressions were computed to be static by the *BTA* analysis) and the dynamic operations into two separate, parallel control-flow subgraphs; in earlier work we called these subgraphs "set-up code" and "template code," respectively [Auslander et al. 96]. Subsection 7.4 discusses some aspects of this separation in more detail. Our method of determining the control flow of the static subgraph, after all dynamic branches have been removed from it, is described in subsection 7.5.
- **Insert explicators:** The compiler inserts explicators in the dynamic subgraph for all variables in the Demotions set at each program point. For Demotions sets at merge nodes, each assignment must be inserted on each predecessor edge to the merge where the now-dynamic variable was previously static.
- **Insert DC operations:** The operations needed to complete the implementation of `Specialize`, such as cache lookups, memory allocation, and branch patching, are inserted into the static and dynamic subgraphs before they are passed through the backend of the compiler. Some optimizations of the calls to the run-time specializer are discussed in subsection 7.7.
- **Integrate:** Finally, each unit's `ReduceAndResidualize` function is completed. The control-flow and the reduce operations of the `ReduceAndResidualize` function are derived from the static control-flow subgraph. The residualize operations are introduced by translating the operations and dynamic branches of the dynamic subgraph into code to emit the dynamic instructions (perhaps with run-time-constant operands) in the static subgraph; this process is described in more detail in subsection 7.6 below. The resulting subgraph forms the `ReduceAndResidualize` function for the unit, and the dynamic subgraph is thrown away.

7.1 Computing Lazy Branch Successors

Laziness policies on variables indicate the extent of speculative specialization that should be performed after dynamic branches. Based on these policies, successors of some dynamic branches are determined to be lazy edges, each of which corresponds to a one-time suspension and resumption of specialization at run time.

A branch successor edge is lazy iff its test variable is dynamic and at least one of the following conditions holds:

- At least one of the run-time constants at the branch is annotated with the Lazy policy,
- The branch successor edge *determines execution* (as defined below) of a predecessor edge of a later specializable merge node, where at least one of the discordant variables is annotated with the SpecializeLazy policy,
- The branch successor edge determines execution of a predecessor edge of a later specializable loop-head merge node, where at least one of the discordant variables is annotated with the LoopSpecializeLazy policy, or
- The branch successor edge determines execution of a later call to a specialized division of a procedure, and some run-time constant live at the call is not annotated with the Eager policy.

We say that a branch successor edge determines execution of a program point iff the edge is postdominated by the program point, but the branch node itself is not, i.e., the branch successor is (one of) the earliest point(s) where it is determined that the downstream program point will eventually be executed. Once the (post)dominator information relating program points is computed, a linear scan over the dynamic branches, specializable merge points, and specialized calls serves to compute the lazy edge information.

7.2 Unit Identification

Each interaction with the run-time specializer, including cache lookup points and demand-driven specialization points, introduces a unit boundary. To identify the boundaries based on cache lookup points, we first compute the *cache context* at each program point from the set of static variables at that point, as follows:

- If any static variable is annotated with the CacheAllUnchecked policy, then the cache context is the special marker replicate.
- Otherwise, the cache context is the pair of the set of variables annotated with the CacheAll policy and the set of variables annotated with the CacheOne policy. (The set of variables annotated with CacheOneUnchecked do not contribute to the cache context.)

Given the cache context and the other program-point-specific information, unit boundaries are identified as follows:*

- Any point where the cache context differs from the cache context at a predecessor point is a unit boundary, since different degrees of polyvariant specialization or of cache retention can occur. In practice, this rule can be relaxed since, except at promotion points, these boundaries are not required for correctness. Unit-boundary clustering (see the next subsection) also helps to mitigate the impact of the many boundaries this rule can insert.
- A non-empty Promotions set at a program point corresponds to a dynamic-to-static promotion point, and introduces a unit boundary.
- A non-empty DiscordantVars list corresponds to a specializable merge point, and introduces a unit boundary.
- Each edge labelled as a lazy edge introduces a unit boundary.

In addition, units are constrained to be single-entry regions. To ensure this, additional unit boundaries are inserted at control-flow merges of paths (including loop back edges) from different units. These unit boundaries can be omitted, however, if all paths from different units have mutually exclusive static reachability conditions (the same way it is determined that multiple static definitions are not truly discordant; see section 6.4). This eliminates the overhead associated with crossing the omitted unit boundaries (discussed in the next subsection), and permits program points to be shared among multiple units, at the cost of larger generating extensions.

The `UnitEdge` data structure records whether each unit edge should be specialized eagerly or lazily. A unit boundary is eager, unless it is a promotion point (which must be suspended until the computed run-time value is available) or a lazy edge.

Figure 16 illustrates the units (shown in gray) that are identified for the interpreter example in Figure 2. The two entry points correspond to the specialized and unspecialized divisions of the `interp_fn` function. The unspecialized entry point and the false branches of both the specialized and unspecialized versions of the conditional-specialization tests lead to unspecialized, statically compiled code. Demotions (indicated by *D*) of `bytecodes` and `pc` are required on the edge from the specialized test as they are evicted from the set of annotated variables.

* Note that a program point can be a boundary in more than one way.

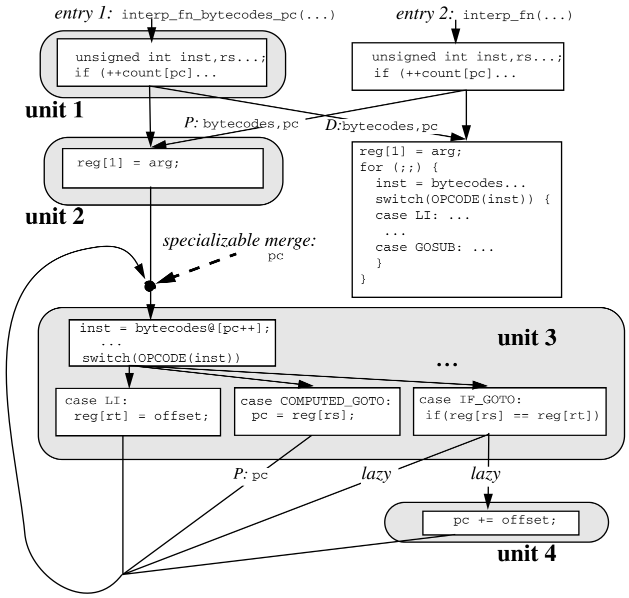


Figure 16: Specialization Units for Figure 2

The specialized entry point begins unit 1. The true branches of the tests merge at the code to be specialized, forming unit 2, which is created by the dynamic-to-static promotion (indicated by *P*) of `bytecodes` and `pc` on the edge from the unspecialized test. Unit 3, which contains the loop body to be specialized, is created because `pc`, which has definitions both inside and outside the loop, is discordant at its head. A promotion of `pc` is required on the back edge from the `COMPUTED_GOTO` case after `pc` is assigned an address location. The successors of the dynamic branch in the `IF_GOTO` case are made *lazy* as required by the (default) LoopSpecializeLazy policy, because the branch determines the execution of different paths to the specializable loop head. The false branch extends to the loop head, so no new unit is required, but the true branch creates the fourth unit.

The specializable loop head will include a specialization-time cache lookup, the edges carrying promotions will correspond to run-time cache lookups, and the lazy edges will become one-time call-backs to the specializer.

7.3 Clustering Unit Boundaries

A unit boundary introduces run-time specialization overhead – to package up the run-time-constant context from the exiting unit's `ReduceAndResidualize` function, to execute the run-time specializer and any cache lookups, and to invoke the target unit's `ReduceAndResidualize` function (unpacking the target's run-time context). In some circumstances, series of unit boundaries can be created with little if any work in between, for instance when a series of annotated static variables become dead, leading to a series of eviction points and corresponding unit boundaries.

To avoid excessive unit boundaries, we attempt to combine multiple boundaries whenever possible.* We have developed a boundary clustering algorithm that works as follows:

- First, for each boundary, we construct the range over the procedure where that boundary can be legally moved. Specializable merge points and lazy-edge boundaries cannot be moved, so their range is a single program point.† Promotion and eviction boundaries can move to any control-equivalent [Ferrante et al. 87] program point that are bounded by earlier and later uses of any promoted or evicted variable; however, promotion points cannot move above earlier definitions.‡ We delay inserting the single-entry-producing unit boundaries until after all the other boundaries have been clustered, so they do not participate in the clustering algorithm.
- Second, we sort the boundary ranges in increasing order of their ends, and then make a linear scan through this sorted list. We remove the range that ends first in the list (call this a *kernel range*), remove all other ranges that overlap with the first range (call the union of these ranges a *cluster*), and find the intersection of these ranges. This resulting intersection is the program region where all of these boundaries can be placed. We prefer earliest possible points for evictions and later points for promotions, as these will reduce the amount of specialized code. We choose either the start or end of the intersection range, based on the relative mix of promotions and evictions, and insert a single boundary for all the merged ranges at that point.** Then we continue processing the sorted list of boundary ranges, until the list is exhausted.

This algorithm for coalescing boundary ranges produces the minimum number of unit boundaries possible, given the restricted kinds of ranges produced in the first step (the restriction to control-equivalent program points is key). To prove this, note that we produce a cluster iff we detect a kernel range, so that the number of clusters is equal to the number of kernels. Since kernels never overlap, no clustering scheme could place two kernels in the same cluster. The number of kernels is therefore also the minimum number of clusters required, implying that our algorithm produces no more clusters and, therefore, no more boundaries than necessary.

Because unit boundaries are also caching points, moving them can increase or decrease the amount of code reuse. Thus, clustering sometimes trades-off reuse for fewer boundary crossings. It may be desirable to limit the length of the ranges so that boundaries sufficiently far away from each other are not coalesced, or otherwise to prevent different types of boundaries that are relatively distant from each other from being clustered together. For example, it may not be beneficial to combine distant boundaries due to evictions and promotions, since eviction boundaries must occur earlier and promotion boundaries later, in order to maximize reuse.

More elaborate versions of the clustering algorithm could permit coalescing of unit boundaries beyond control-equivalent regions, but this would require more than a straightforward extension to the algorithm presented above. The ranges would no longer be strictly linear. Moving boundaries below branches or above control-flow merges would create identical boundaries on all paths from the branches or to the merges. Moving boundaries in the opposite direction could only be permitted if identical boundaries existed on all the paths.

* An obvious alternative to clustering is simply to introduce fewer boundaries when possible, such as at eviction points. It would be interesting to compare the impact of these two techniques in real applications.

† Except at loop heads, cache lookups due to specializable merge points could be permitted to be moved down by the clustering algorithm. This would decrease the number of boundaries, but would also decrease the amount of code reuse.

‡ Definitions and uses are mobile as well, so a fair range of motion should be possible while still respecting data and control dependences.

** One need not place the boundaries only at the end points of the intersection ranges. One could choose the final position for a boundary by selecting an offset within its intersection range that is scaled by the ratio of the numbers of evictions and promotions.

7.4 Separating Static and Dynamic Operations

For most straight-line operations, it is clear whether the operation is static or dynamic. However, call instructions are trickier.

- A call to a regular unspecialized function (or to the unspecialized version of a specialized function) is treated as a dynamic operation and appears only in the dynamic subgraph.
- A call to a `constant` function (or one annotated with `@`) with static arguments is treated as a regular static computation, appearing only in the static subgraph.
- A call to a particular specialized division of a function has both static and dynamic components. To implement this, the call operation is split into two separate calls, one static and one dynamic. The static version of the call invokes the statically compiled generating extension for the selected division of the callee, taking as arguments the division's static arguments, and returning a static procedure address. This is followed by a dynamic call that invokes the static procedure address and passes the remaining arguments to produce a dynamic result.[*] The static call will be moved to the static subgraph, and the dynamic call will appear in the dynamic subgraph.

Control-flow nodes, including branches and merges, initially are replicated in both the static and the dynamic subgraphs. Later transformations can optimize them.

7.5 Determining Control Flow of the Static Subgraph

Once each unit has been identified and split into separate static and dynamic control-flow subgraphs, the control-flow structure of the unit's `ReduceAndResidualize` function is computed. Static and dynamic branches in the unit receive different treatment. A static branch is taken at specialization time, and does not appear in the dynamically generated (residual) code; accordingly, only one of its successors produces dynamically generated code. Consequently a static branch appears as a regular branch in the final `ReduceAndResidualize` function, selecting some single successor to pursue and residualize. A dynamic branch, on the other hand, is emitted as a regular branch into the dynamically generated code, and both its successors must be residualized. Consequently, no branch appears in the `ReduceAndResidualize` function at a dynamic branch, and the successors of the dynamic branch are linearized instead.

Figure 17 illustrates how the dynamic branches are linearized. Numbered boxes represent basic blocks and circles represent branches. The circle enclosing an `s` represents a static branch and the one containing a `d` represents a dynamic branch.

In the presence of arbitrary, unstructured control flow with mixed static and dynamic branches, this linearization process may require some code duplication to avoid maintaining specialization-time data structures and overhead. Our algorithm first splits all static control paths[†] within the unit, linearizing dynamic branches by topologically sorting their successors, then re-merges the common tails of the static paths bottom-up. The time required by the algorithm can be exponential in the maximum number of sequential

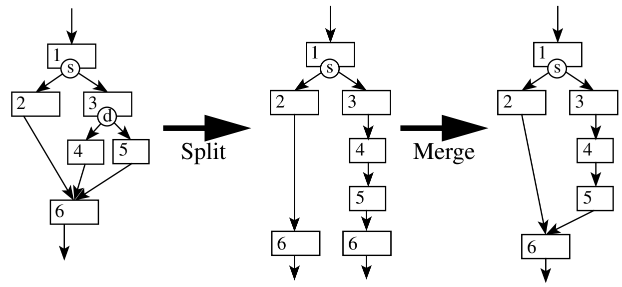


Figure 17: Linearization

static branches on any static control path within a single unit, which we expect to be a small number in practice.

Linearization causes what were originally alternative code segments to be executed sequentially. We must ensure that the segments executed earlier do not alter the initial static state expected by subsequent alternative segments. This could be achieved by saving the static state at each dynamic branch and restoring it before executing each branch successor. This is the approach we have taken in order to propagate the static context between units. However, within a single unit, a more efficient solution is possible by converting static variables to static-single-assignment (SSA) form [Cytron et al. 89]. SSA form ensures that only one assignment is made to each variable, which implies that state changes made by segments that occur earlier in the linearized unit are made to variables not read by alternative segments. In this case, the SSA form is easy to compute, because issues arising from loops and aliasing can be safely ignored due to DyC's restrictions on the form of units (i.e., units cannot contain static loops) and its prohibition of static stores. If these restrictions were eased, however, an alternate solution may have to be found.

7.6 Integrating Dynamic Code into Static Code

To produce the final code for a unit's `ReduceAndResidualize` function, we take the linearized static control-flow graph which computes all the static expressions, and blend in code to generate the dynamic calculations with the appropriate run-time constants embedded in them. To accomplish this, our system maintains a mapping from each basic block in the dynamic subgraph to a set of corresponding basic blocks in the static subgraph. When splitting apart static and dynamic operations, the mapping is created, with each dynamic block mapping to its static counterpart(s).[‡] The mapping is updated, as the static subgraph is linearized and some blocks are replicated, and as the subgraphs are optimized through instruction scheduling. The two subgraphs are integrated, one dynamic block at a time. First, the static code computes any run-time constants used in the block's dynamic instructions. Then, code to emit the dynamic block is appended to its corresponding static block.

The code to emit a dynamic instruction embeds the values of any small run-time constant operands into the immediate field of the emitted instruction. If the run-time constant is too large to fit in the immediate field, code is emitted to load it from a global table into a scratch register. The emitted instruction then reads the scratch register to access the run-time constant. The emitting code also performs any peephole optimizations that are based on the run-time constant value, such as replacing multiplications by constants with sequences of shifts and adds.

[*] Tempo performs interprocedural binding-time analysis and so can deduce that the result of a specialized function is static. If we were to extend DyC to support interprocedural analysis of annotations, then the static half of the call would return both a procedure address and the static result value, and the dynamic half would return no result and be invoked only for its side-effects.

[†] A static control path includes all dynamically reachable basic blocks, given particular decisions for all static conditional branches. Each static branch can appear on a static control path at most once, because units cannot contain static loops.

[‡] Unit linearization may create multiple instances of a basic block in the static subgraph, as mentioned in section 7.5.

7.7 Optimizing Specializer Interactions

Each initial promotion point at the entrance to a dynamic region is implemented by generating a static call to the run-time specializer, passing the run-time values of the cache context at that program point. Section 4 described the run-time specializer as if a single general-purpose specializer took control at this and all other unit boundaries. Our system optimizes this pedagogical model as follows:

- The `Specialize` function is specialized for each `Unit` argument. All the run-time manipulations of the `Unit` and `UnitEdge` data structures are eliminated, the unit's `ReduceAndResidualize` function is inlined, and the processing of outgoing lazy unit edges is inlined. If the cache policy for any of the unit's context variables is CacheAllUnchecked, then the cache lookup and store calls are omitted.
- Rather than recursively call `Specialize`, a `pending-list` is used to keep track of unprocessed (eager) unit edges. Furthermore, the overhead of pushing and popping the static context on and off of the `pending-list` can be avoided for one successor of each unit, which eliminates more than half of this overhead in dynamic regions without dynamic `switch` statements.
- Ends of dynamic regions are compiled into direct jumps to statically compiled code.

8 Experience with DyC

We have implemented the core functionality of the system in the context of the Multiflow compiler [Lowney et al. 93]. Only the function annotations, the CacheOne policy, unit-boundary clustering, and unit linearization have not yet been fully implemented. We have encountered a number of practical difficulties in the implementation, particularly in the implementation of the annotations. Most of these problems related to naming, i.e., establishing a correspondence between the variables that the programmer sees in the source code and their internal representation in the compiler; this issue is discussed in subsection 8.1.

Despite the challenges, we achieved good results with a larger application than previously had been dynamically compiled by other general-purpose dynamic-compilation systems. Subsection 8.2 describes our positive experiences with this and other applications. On the other hand, as we applied DyC to various programs, we encountered several weaknesses in our current design, and these are discussed in subsection 8.3.

8.1 Challenges in Implementing the Annotations

In the Multiflow compiler, all computations are represented as operations whose operands are virtual registers called *temporaries*. Temporaries are created on demand by the compiler and their names bear no correspondence to source-level variable names. At different program points, a source variable may correspond to different temporaries, and optimizations such as induction-variable simplification or variable expansion* may even create multiple simultaneously live temporaries corresponding to a single variable. Since the programmer annotates source variables, our implementation computes a source-variable-to-temporary correspondence at each program point. This correspondence relation is used to apply the *BTA* rules to those temporaries that correspond to annotated source variables and any temporaries derived from them.

Several standard compiler optimizations make maintaining this correspondence difficult. For example, copy propagation can result in the annotated variable (i.e., its corresponding temporary) being replaced by another non-annotated temporary, typically resulting in less specialization than desired by the programmer. In the following source code:

```
   make_static(x);
   x = y;
   if (d) x = x + 1; else x = x + 2;
M: .. x ../* no further uses of y */
```

variables `x` and `y` are represented by temporaries `tx` and `ty`, respectively:

```
   make_static(tx);
   tx = ty;
   if (td) tx = tx + 1; else tx = tx + 2;
M: .. tx ..
```

Multiflow's copy propagation and temporary renaming phase transform this into:

```
   make_static(tx);
   if (td) ty = ty + 1; else ty = ty + 2;
M: .. ty ..
```

Since the source variable corresponding to temporary `ty` is not annotated, the `make_static` annotation on `x` is effectively lost, leading to less specialization in the program than expected by the programmer. We combat this problem by attempting to maintain the source-variable-to-temporary correspondence through Multiflow's many optimization phases, with varying degrees of success.

Induction-variable simplification can similarly cause loop-induction variables to be replaced with temporaries that do not obviously correspond to annotated (or any) source variables. Because the specialization annotation on the individual variable has been lost, the loop may not be unrolled as desired. To avoid this problem, we currently disable this optimization at some cost in code quality.

Variable expansion, which is performed by the Multiflow compiler during loop unrolling, exacerbates the problem of lost annotations. Since several temporaries are created and are modified independently in the loop body, the source-variable-to-temporary correspondence cannot be easily established. To get around this problem, we currently disable (compile-time) loop unrolling in some cases as well.

8.2 Preliminary Experiences with Applications

We have applied DyC to a few kernels previously used as benchmarks for other dynamic compilation systems, and have obtained speedups and overhead comparable to these systems. The kernels are typically 100-200 lines of C code with dynamic regions of size 10-25 lines. Our dynamic-compilation overhead ranged between about 20 and 200 cycles per instruction generated, on the Digital Alpha 21164.

The automation provided by our system also allowed us to experiment with dynamically compiling a larger program, the `mipsi` architectural simulator for the MIPS R3000 architecture. The simulator consisted of approximately 9100 lines of C with a dynamic region roughly 400 lines long. We were able to dynamically compile the simulator by converting a few global variables to local variables, and then adding just three lines of annotations, very similar to those in Figure 1. Nearly all of DyC's functionality was exercised, including polyvariant specialization, automatic dynamic-to-static promotion, and automatic caching. This resulted in constant folding, constant branch removal, load elimination, call elimination, and multi-way complete loop

* Variable expansion creates `n` copies of a variable in the body of a loop that is unrolled by a factor of `n`, one for each unrolled body, and combines the values at the loop exits to produce the value that the original variable would have had. Creating `n` copies reduces the dependences in the loop body, thereby enabling potentially better instruction schedules.

unrolling. The reachability analysis also proved useful in several instances by preventing derived static variables defined under static control from being dropped from the set of run-time constants at static merges. (Tempo was recently used to dynamically specialize an interpreter comparable in size to `mipsi` [Thibault et al. 98].) All of these optimizations yielded a speedup ranging from 2 to 5, depending on the input program, at an overhead of approximately 200 cycles per instruction generated.

8.3 Areas Requiring Improvement

As we applied DyC to `mipsi` and to the small benchmarks, we encountered a number of weaknesses of our current design. These weaknesses did not reduce specialization opportunities, but made the system less automatic than we had hoped. Inadequate support for global variables and partially static data structures may be DyC's most serious shortcoming. Most programs we wish to dynamically compile require specialization for static or partially static data structures, and `mipsi` used global variables as well. The `@` annotation allows DyC to perform dereferences at specialization time. If the annotated data structures are actually invariant, then this approach works fine; otherwise, it is insufficient. For example, in `mipsi` we had to manually copy global variables to annotated local variables whenever their values may have changed. Unfortunately, extending DyC to be capable of performing static stores would require significant changes to our context-management strategy, caching mechanism, and unit-linearization scheme. Also, additional annotations (or interprocedural analysis) would be required to position explicators for statically written memory locations.

Additional analyses, for example, to automatically determine when cache lookups and lazy branches could be safely eliminated, would be useful. Such analyses would reduce the need to use the unsafe caching and laziness policies, which we used extensively in the small benchmarks to achieve the greatest possible performance with the least overhead. At the other end of the ease-of-use spectrum, an invalidation-based caching and dispatching mechanism could also reduce the cost of safety. For dynamic regions or specialized functions using an invalidation-based cache policy (hypothetically, InstallOne, InstallAll, or InstallAllUnchecked), one specialization would be installed as the currently valid version and it would be invoked with direct jumps or calls until invalidated. Following invalidation, the next execution of the region or function would fall back on DyC's existing caching schemes (CacheOne, CacheAll, or CacheAllUnchecked, respectively), and the version retrieved from the cache (or the newly specialized version) would be installed as the current one. Such a scheme could improve performance for applications in which it could be easily determined when to invalidate the current specialized version of each dynamic region.

9 Comparison To Related Work

Tempo [Consel & Noël 96], a compile-time and run-time specialization system for C, is most similar to DyC. The two systems differ chiefly in the following ways:

- DyC may produce multiple divisions and specializations of program points, with the degree of division and specialization varying from point to point. Tempo supports only function-level polyvariant division and specialization, with no additional division or specialization possible within the function, except for some limited support for complete loop unrolling.
- DyC performs analysis over arbitrary, potentially unstructured control-flow graphs. Tempo converts all instances of unstructured code to structured form [Erosa & Hendren 94, Consel et al. 96], which introduces a number of additional tests and may also introduce loops.
- DyC allows dynamic-to-static promotions to occur anywhere within dynamically compiled code. Tempo requires such promotions to occur only at the entry point.
- DyC allows the programmer to specify policies to control division, specialization, caching, and speculative specialization. Tempo does not provide user controls; the client program must perform its own caching of specialized code if desired. A Java front-end to Tempo has been designed, however, that provides automatic caching and policies to govern replacement in the cache; users may also implement their own policies [Volanschi et al. 97].
- DyC relies on the programmer to annotate memory references as static. Tempo performs an automatic alias and side-effect analysis to identify (partially) static data structures. Tempo's approach is more convenient for programmers and less error-prone, but it still is not completely safe, relies on the programmer to correctly describe aliasing relationships and side-effects of parts of the program outside of the module being specialized, and may benefit from explicit user annotations wherever the analysis is overly conservative. However, a strong benefit of Tempo's approach is that static writes to memory are possible.
- DyC supports separate compilation while still being able to specialize call sites and callee functions for the values of their static arguments, but performs no interprocedural analysis. Tempo performs interprocedural side-effect and binding-time analyses, can also specialize functions for the values of static global variables, and can identify static return results of residual functions. However, it requires the whole module being specialized to be analyzed and compiled as a unit.
- Tempo also supports compile-time specialization.

In our view, DyC's focus on intraprocedural specialization, automatic caching and dispatching, control over specialization, and low run-time overhead is fairly complementary to Tempo's focus on interprocedural specialization, support for partially static data structures, and uniform support for compile-time and run-time specialization.

Fabius [Leone & Lee 95, Leone & Lee 96] is another dynamic compilation system based on partial evaluation. Fabius is more limited than DyC or Tempo, working in the context of a first-order, purely functional subset of ML and exploiting a syntactic form of currying to drive dynamic compilation. Only polyvariant specialization at the granularity of functions is supported. Given the hints of curried function invocation, Fabius performs all dynamic compilation optimizations automatically with no additional annotations; by the same token, the trade-offs involved in the dynamic compilation process are not user-controllable. Fabius does little cross-dynamic-statement optimization other than register allocation, since, unlike DyC, it does not explicitly construct an explicit dynamic subgraph that can then be optimized.

Compared to our previous system [Auslander et al. 96], DyC has a more flexible and expressive annotation language, support for polyvariant division and better support for polyvariant specialization, support for nested and overlapping dynamic regions, support for demand-driven (lazy) specialization, support for interprocedural specialization, a much more efficient strategy for and optimizations of run-time specialization, and a more well-developed approach to caching of specialized code.

Outside the realm of dynamic compilation, other partial evaluation systems share characteristics with DyC. In particular, C-mix [Andersen 92b, Andersen 94] is a (compile-time) offline partial-evaluation system for C. Its analyses differ from DyC's in the following ways:

- C-mix provides program-point polyvariant specialization, but only function-level polyvariant division.
- While DyC computes point-wise divisions, C-mix's divisions are uniform; that is, it assigns only one binding time, static or dynamic, to each variable and does not permit variables to change from static to dynamic or vice-versa. However, C-mix's analysis runs in near-linear time and is efficient enough to apply interprocedurally, while DyC's intraprocedural analysis has exponential (worst-case) complexity.
- C-mix copes directly with unstructured code, but it appears to lack reachability analysis to identify static merges [Andersen 94].
- C-mix handles partially static structures by splitting the structures into separate variables.
- C-mix includes support for automatic interprocedural call graph, alias, and side-effect analyses.
- C-mix also provides annotations for controlling code growth by limiting specialization with respect to certain variables and for overcoming the limitations of its conservative analysis; however, its annotations provide less control than DyC's. C-mix always polyvariantly specializes control-flow merges, and provides the `residual` annotation to make a variable dynamic in order to prevent explosive code growth due to multi-way loop unrolling. In contrast, DyC provides control over code growth by permitting variables to be specialized monovariantly or by specializing lazily on demand. C-mix's `pure` annotation corresponds to `constant`, and `unfold` fills the role of the `inline` pragma provided by most modern optimizing compilers.

Andersen's *dynamic basic blocks* (DBBs) [Andersen 92a] serve the same purpose as specialization units, to reduce overhead in the specializer; however, their boundaries are determined entirely differently. DyC's specialization units differ from C-mix's dynamic basic blocks in the following ways:

- DBBs are bounded by (and may not contain) dynamic control flow. On the other hand, DyC's units are designed to include dynamic control flow (via linearization).
- C-mix does not automatically insert specialization points (and thus begin new DBBs) at specializable merge points in order to enable code sharing. Unit boundaries are required wherever a new variant of the code must be begun, at both dynamic-to-static promotions and specializable merge points. Unit boundaries are also inserted where cache lookups could enable sharing (i.e., at eviction points).
- DBBs may overlap. Units currently cannot overlap, though that restriction could be relaxed, as described in section 7.2.

Schism's filters permit choices about whether to unfold or residualize a function and which arguments to generalize (i.e., make dynamic), given binding times for the function's parameters [Consel 93]. Because filters are executed by the binding-time analysis, only binding-time information can be used to make decisions. DyC's conditional specialization can use the results of arbitrary static or dynamic expressions to control all aspects of run-time specialization.

Filters can be used to prevent unbounded unfolding and unbounded specialization. Both offline partial evaluators, such as Schism, and online specializers, such as Fuse [Weise et al. 91], look for dynamic conditionals as a signal that unbounded unfolding or specialization could occur and specialization should be stopped. Run-time specializers have an additional option, which is to temporarily suspend specialization when dynamic conditionals are found in potential cycles and insert lazy callbacks to the specializer; currently, only DyC exploits this option.

`C extends the ANSI C language to support dynamic code generation in an imperative rather than annotation-based style [Engler et al. 96]. The programmer must specify code to be generated at run time, substitute run-time values and combine code fragments (called tick expressions), perform optimizations, invoke the run-time compiler, manage code reuse and code-space reclamation, and ensure correctness. In return for this programming burden, `C would seem to offer greater expressiveness than a declarative, annotation-based system. However, DyC's ability to perform arbitrary and conditional polyvariant division and specialization enables it to perform a wide range of optimizations with very little user intervention, and DyC offers capabilities not available in `C. For instance, `C cannot (multi-way) unroll loops with dynamic exit tests, because jumps to labels in other tick expressions are not permitted. (`C recently added limited support for automatic single-way complete loop unrolling within a tick expression [Poletto et al. 97].) Also, tick expressions cannot contain other tick expressions, so nested and overlapping dynamic regions cannot be supported. Both of these weaknesses would appear to prevent `C from handling the simple interpreter example in Figure 1. `C can support run-time compiled functions with a dynamically determined number of arguments, but it may be feasible to achieve at least some of this behavior in DyC by specializing a procedure based on the length and values in its `varargs` pseudo-argument. One advantage that `C does have is that the programmer can easily implement a variety of dispatching mechanisms, which may be important in exploiting certain opportunities for dynamic compilation, such as data decompression [Keppel 96].

A declarative system such as DyC allows better static optimization of dynamic code than an imperative system such as `C, because the control flow within a dynamic region is more easily determined and conveyed to the rest of the optimizing compiler. Optimization across tick expressions is as hard as interprocedural optimization across calls through unknown function pointers [Poletto et al. 97].* Finally, programs written in declarative systems can be easier to debug: since (most of) the annotations are semantics-preserving, programs can simply be compiled ignoring them. Debugging the use of unsafe annotations is still challenging, however.

10 Conclusions

We have presented the design of DyC, an annotation-based system for performing dynamic compilation that couples a flexible and systematic partial-evaluation-based model of program transformation with user control of key policy decisions. Our annotations' design resulted from a search for a small set of flexible primitive directives to govern dynamic compilation, suitable for use by both human programmers and tools (such as a semi-automatic dynamic-compilation front-end). With the exception of support for static data structures, we believe that our `make_static` annotation provides the flexibility we require in a concise, elegant manner. By adding policy annotations, users can gain fine control over the dynamic compilation process when needed. Our support for arbitrary program-point-specific polyvariant division and specialization is a key component of DyC's flexibility, enabling, for instance, multi-way loop unrolling and conditional specialization, as illustrated in the interpreter example. We exploit the unusual capabilities of run-time specialization in the forms of arbitrary dynamic-to-static promotion and demand-driven specialization.

We have implemented the core functionality of the system in the context of an optimizing compiler. Our initial experience in using DyC has been promising – DyC has obtained good speedups (over statically compiled code) with low run-time overhead, and required

* If run-time inlining through function pointers were available in DyC, analysis across those calls would be of comparable difficulty.

little modification of source programs. The majority of our system's functionality has been used in the single large program with which we have experience. Once the full implementation is complete, we plan to focus on applying dynamic compilation to other sizeable, real application programs. We will use these applications to further evaluate DyC's design and implementation. We also plan to extend DyC with additional run-time optimizations, such as run-time inlining and register allocation (via register actions).

Acknowledgments

We are grateful to Charles Consel for his help in understanding Tempo and some of the related issues in partial evaluation. We also thank the anonymous referees for finding several errors and suggesting other improvements to the paper, David Grove for feedback on earlier drafts of this paper, Charles Garrett for his implementation work on our dynamic compiler, John O'Donnell and Tryggve Fossum for the source for the Alpha AXP version of the Multiflow compiler, and Ben Cutler, Michael Adler, and Geoff Lowney for technical advice in altering it. This work is supported by ONR contract N00014-96-1-0402, ARPA contract N00014-94-1-1136, NSF Young Investigator Award CCR-9457767, and an NSF Graduate Research Fellowship.

References

[Andersen 92a] L.O. Andersen. C Program Specialization. Technical Report 92/14, DIKU, University of Copenhagen, Denmark, May 1992.

[Andersen 92b] L.O. Andersen. Self-Applicable C Program Specialization. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation '92*, pages 54–61, June 1992. Published as Yale University Technical Report YALEU/DCS/RR-909.

[Andersen 94] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1994. Published as DIKU Research Report 94/19.

[Auslander et al. 96] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, Effective Dynamic Compilation. *SIGPLAN Notices*, pages 149–159, May 1996. In Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation.

[Consel & Noël 96] C. Consel and F. Noël. A General Approach for Run-Time Specialization and its Application to C. In *Conference Record of POPL '96: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, January 1996.

[Consel 93] C. Consel. A Tour of Schism: A Partial Evaluation System for Higher-Order Applicative Languages. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation '93*, pages 145–154, 1993.

[Consel et al. 96] C. Consel, L. Hornof, F. Noël, J. Noyé, and N. Volanschi. A Uniform Approach for Compile-Time and Run-Time Specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany,February 1996*, LNCS 1110, pages 54–72. Springer-Verlag, 1996.

[Cytron et al. 89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, January 1989.

[Engler & Proebsting 94] D. R. Engler and T. A. Proebsting. DCG: An Efficient, Retargetable Dynamic Code Generator. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 263–273, October 1994.

[Engler et al. 96] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A Language for High-Level, Efficient, and Machine-Independent Dynamic Code Generation. In *Conference Record of POPL '96: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 131–144, January 1996.

[Erosa & Hendren 94] A.M. Erosa and L.J. Hendren. Taming Control Flow: A Structured Approach to Eliminating goto Statements. In *Proceedings of 1994 IEEE International Conference on Computer Languages*, pages 229–240, May 1994.

[Ferrante et al. 87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[Goldberg & Robson 83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addision-Wesley, 1983.

[Jones et al. 93] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[Keppel 96] David Keppel. *Runtime Code Generation*. PhD thesis, University of Washington, 1996.

[Kernighan & Ritchie 88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language (second edition)*. Prentice Hall, 1988.

[Leone & Lee 95] M. Leone and P. Lee. Optimizing ML with Run-Time Code Generation. Technical report CMU-CS-95-205, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 1995.

[Leone & Lee 96] M. Leone and P. Lee. Optimizing ML with Run-Time Code Generation. *SIGPLAN Notices*, pages 137–148, May 1996. In Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation.

[Meyer 91] U. Meyer. Techniques for Partial Evaluation of Imperative Languages. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation '91*, pages 94–105, June 1991. Published as SIGPLAN Notices 26(9).

[Poletto et al. 97] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation. *SIGPLAN Notices*, pages 109–121, June 1997. In Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.

[Sirer 93] Emin Gun Sirer. Measuring Limits of Fine-Grain Parallelism. Princeton University Senior Project, June 1993.

[Steensgaard 96] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Conference Record of POPL '96: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[Thibault et al. 98] Scott Thibault, Charles Consel, and Gilles Muller. Safe and Efficient Active Network Programming. Technical Report Research Report 1170, IRISA, January 1998.

[Volanschi et al. 97] E. N. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. *SIGPLAN Notices*, 32(10):286–300, October 1997.

[Weise et al. 91] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic Online Partial Evaluation. In J. Hughes, editor, *Record of the 1991 Conference on Functional Programming Languages and Computer Architecture*, LNCS 523, pages 165–191, Cambridge, MA, 1991. Springer-Verlag.

[Wilson & Lam 95] R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. *SIGPLAN Notices*, pages 1–12, June 1995. In Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.

Appendix A Grammar of Annotations

statement:
```
  ...  /* same as in regular C */
  make_static ( static-var-list ) ;
  make_dynamic ( var-list ) ;
  make_static ( static-var-list ) compound-statement
```

static-var-list:
  static-var
  static-var , *static-var-list*

static-var:
  identifier policies$_{opt}$

policies:
  : *policy-list*

policy-list:
  policy
  policy , *policy-list*

policy:
  division-policy
  specialization-policy
  promotion-policy
  merge-caching-policy
  promotion-caching-policy
  laziness-policy

division-policy:
```
poly_divide
mono_divide
```

specialization-policy:
```
poly_specialize
mono_specialize
```

promotion-policy:
```
auto_promote
manual_promote
```

merge-caching-policy:
```
m_cache_all_unchecked
m_cache_all
m_cache_one
m_cache_one_unchecked
```

promotion-caching-policy:
```
p_cache_none_unchecked
p_cache_all
p_cache_one
p_cache_one_unchecked
```

laziness-policy:
```
lazy
specialize_lazy
loop_specialize_lazy
eager
```

var-list:
  identifier
  identifier , *var-list*

external-definition:
  ... /* same as in regular C */
  specialize-definition

specialize-definition:
  constant$_{opt}$ specialize *identifier* (*var-list*)
    on *specialize-list* ;

specialize-list:
  (*static-var-list*)
  (*static-var-list*) , *specialize-list*

expression:
  ... /* same as in regular C */
  @ * *expression*

primary:
  ... /* same as in regular C */
  @ *identifier*
  primary @(*expression-list*$_{opt}$)
  primary @[*expression*]
  lvalue @. *identifier*
  primary @-> *identifier*