Annotation-Directed Run-Time Specialization in C

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

Department of Computer Science and Engineering
University of Washington
http://www.cs.washington.edu/research/dyncomp/

Abstract

We present the design of a dynamic compilation system for C. Directed by a few declarative user annotations specifying where and on what dynamic compilation is to take place, a binding time analysis computes the set of run-time constants at each program point in each annotated procedure's control flow graph; the analysis supports program-point-specific polyvariant division and specialization. The analysis results guide the construction of a specialized run-time specializer for each dynamically compiled region; the specializer supports various caching strategies for managing dynamically generated code and supports 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. Our design is being implemented in the context of an existing optimizing compiler.

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 specialized to actual run-time behavior. The principal challenge and trade-off 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 would most profitably be applied. Some previous dynamic compilation systems, such as `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. This kind of system offers great flexibility and control to the programmer, at the cost of significant programmer effort and debugging difficulty.

Alternatively, several dynamic compilation systems, including Fabius [Leone & Lee 96], Tempo [Consel & Noël 96], and our own previous system [Auslander *et al.* 96], take a declarative approach, with user annotations guiding the dynamic compilation process. Fabius uses function currying to drive dynamic compilation, in a purely functional subset of ML; Tempo uses function-level annotations, annotations on global variables and structure types, and alias analysis to drive dynamic compilation in C; and our previous system uses intraprocedural annotations to drive dynamic compilation in C. Each of these declarative approaches adapts ideas from partial evaluation, expressing dynamic compilation as off-line run-time specialization, where static values correspond to run-time state on 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 by the programmer) and easier program understanding and debugging (since declarative annotations can be designed to not affect the meaning of the underlying programs). However, declarative systems usually offer less expressiveness and control over the dynamic compilation process than do 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, while still being easy to use. Our system supports the following features:

- program-point-specific rather than function-level specialization,
- support for both polyvariant specialization and polyvariant division* (both of which have practical utility), with the degree of specialization for different variables under programmer control,
- intra- and interprocedural specialization, with caller and callee separately compilable,
- arbitrary nested and overlapping regions of dynamically generated code,
- automatic caching, reuse, and reclamation of dynamically generated code, with cache policies under control of the programmer,
- 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,
- run-time optimizations, including constant propagation, constant folding, strength reduction, conditional branch folding and dead code elimination, loop unrolling and merge splitting, and procedure call specialization.

The next section illustrates many of the capabilities of our system using an annotated bytecode interpreter example. Section 3 describes our run-time specializer and its capabilities, and then

* 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.

In PEPM'97 Proceedings

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

int interp_fn(int bytecodes[], int pc, int arg) {
  int stack[N], sp = 0;
  make_static(bytecodes, pc, sp);
  stack[sp++] = arg;
  for(;;) {
    switch (bytecodes static[pc++]) {
      case CONST:
        stack[sp++] = bytecodes static[pc++];
        break;
      case ADD:
        stack[sp-1] = stack[sp-1] + stack[sp]; sp--;
        break;
      ...
      case LT:
        stack[sp-1] = stack[sp-1] < stack[sp]; sp--;
        break;
      ...
      case IF_GOTO:
        int nextpc = bytecodes static[pc++];
        if (stack[sp--]) {
          pc = nextpc;
        }
        break;
      case GOTO:
        pc = bytecodes static[pc++];
        break;
      case COMPUTED_GOTO:
        pc = stack[sp--];
        break;
      ...
      case RETURN:
        return stack[sp];
    }
  }
}
```

Figure 1: Simple Interpreter Example

```
int count[N];
#define threshold ...
specialize interp_fn(bytecodes, pc, arg)
  on (bytecodes, pc);
int interp_fn(int bytecodes[], int pc, int arg) {
  int stack[N], sp = 0, callee;
  if (++count[pc] >= threshold) {
    make_static(bytecodes, pc, sp);
  } else {
    make_dynamic(bytecodes, pc, sp);
  }
  stack[sp++] = arg;
  for(;;) {
    switch (bytecodes static[pc++]) {
      ... /* same as above */
      case GOSUB:
        callee = bytecodes static[pc++];
        stack[sp] =
          interp_fn(bytecodes, callee, stack[sp]);
        break;
    }
  }
}
```

Figure 2: Interprocedural and Conditional Specialization

sections 4 through 6 present our annotation language, our analysis to compute program-point-specific information, and our approach to producing an (optimized) run-time specializer from the program-point-specific information, respectively. Section 7 compares our system to related work, and section 8 concludes with our plans for future work.

2 Example

Figure 1 presents a simple interpreter of a bytecode program like those in the Smalltalk and Java virtual machines [Goldberg & Robson 83, Lindholm & Yellin 97]. In boldface are the annotations we added to turn the interpreter into a program that produces at run time an interpreter specialized for the particular array of bytecodes, i.e., a run-time compiler.

The main control annotation is `make_static`, whose argument list of variables the system is to treat as *run-time constants* when run-time execution reaches that point. By default, the system will apply 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 `pc` variable is annotated as static. The system will specialize code so that, at each program point in the specialized code, the `pc` variable will have a known run-time constant value. The increments of `pc` in the `switch` body do not cause problems, since the value of `pc` after the increment is a run-time constant, if the value of `pc` before the increment is. The loop head at the top of the `for` loop requires additional work: our system 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 fully.

The references to the contents of the bytecode array are annotated as static references, implying that the contents of the referenced memory location is a run-time constant if its address is.* This enables the system to constant-fold the switch branch within each iteration (since `bytecodes` and `pc` are run-time constants and the loaded bytecode is a run-time constant), selecting just one case arm in each iteration and eliminating the others as dead code. All the code manipulating the `bytecodes` array and the `pc` value itself are also eliminated as dead, once all the interpretation overhead on these data values is constant-folded away. Similarly, at each program point in the unrolled loop the `sp` variable will have a specific run-time constant value, and so all the references and updates to `sp` will be eliminated as dead code, once the index expressions into the `stack` array are replaced with particular run-time constant values. The contents of the `stack` array are not run-time constants, as they depend on the initial `arg` value and subsequent input program execution.

The `IF_GOTO` bytecode rebinds the value of `pc` conditionally 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. By default, because `pc` is annotated as `make_static`, our system will apply polyvariant specialization to the merge and all downstream code, making potentially two copies of the merge and successors, one copy for each run-time constant value of `pc`. For an input program containing 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*. (Our system allows the programmer to specify less aggressive specialization policies for static variables, to provide programmers finer control over the trade-offs between run-time specialization overhead and run-time specialization benefit.)

At each of these *specializable merge points*, by default our system will maintain a cache of all the previously specialized versions, indexed by the values of the static variables at that merge point. When encountering a specializable merge point during run-time specialization, the cache will be examined to see whether a version of that code has already been produced, and, if so, that previous version will be reused. In the interpreter example, the cache checks

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

at the loop head merge have the effect of connecting backward branch bytecodes directly to previously generated iterations, forming loops in the specialized code as desired, and similarly to introduce sharing of iterations if there exist other control flow merge points in the input interpreted program. (Our system allows the programmer to specify alternative caching policies or even that no caching should be used, to provide finer control to the programmer over this potentially expensive primitive.)

The `COMPUTED_GOTO` bytecode assigns the `pc` variable to a dynamic expression. By default, our system will suspend program specialization until run-time execution reaches that program point, at which point the system will resume specialization using the actual value assigned to `pc` at that point. 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, and the specializer consults this cache of previously specialized versions to see whether a previous version can be reused or a new version must be produced. The initial `make_static` entry is also a dynamic-to-static promotion point with an associated cache of versions specialized for different run-time values of the initial static variables. (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.)

A standard issue in specialization is how aggressively to specialize control-flow paths ahead of actually reaching those branches during normal program execution. Aggressive *speculative* specialization has the lowest cost, assuming that all specialized paths will eventually be taken at run time, but it incurs the cost of specializing any path not actually executed at run time, and it can lead to non-termination in the presence of loops or recursion. Alternatively, *demand-driven* specialization only specializes code when it can be proven to be executed at run time, typically by suspending specialization at each successor of a dynamic (non-constant) branch in the program being specialized, resuming specialization 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. Our system allows the programmer to specify policies with variables to control speculative specialization based on those variables, with the default policy introducing suspension points at each specializable loop head.

Figure 2 extends the simple single-procedure interpreter to support interpreting programs made up of multiple procedures. It also illustrates several other of our system's capabilities, in particular how polyvariant division can be exploited to support conditional specialization, and annotations that support interprocedural specialization.

In the modified `interp_fn` routine, a `count` array associates with each `pc` corresponding 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` call from Figure 1 conditional, occurring only when the invocation count of some interpreted procedure reaches a threshold. At the merge after the `if`, `bytecodes`, `pc`, and `sp` are static along one predecessor but dynamic along the other. By default, our system applies polyvariant division to produce two separate versions of the remainder of the body of `interp_fn`, one where the three variables are static and lead to run-time specialization as in Figure 1, and one where they are dynamic and no run-time specialization takes place, leading to regular interpretation of the input at no run-time cost.

The `specialize` annotation directs the compiler to produce an alternate entry point to the `interp_fn` procedure to be used when its first two parameters are run-time constants. At call sites of the `interp_fn` procedure 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 the `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 the `bytecodes` array nor the pc variable 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 version of the unspecialized interpreter.

3 Run-Time Specializer

In this section we describe our run-time specializer. Later sections present our annotation language and describe how annotated programs get translated down into static precompiled code and run-time specializers. Figures 3 and 4 sketch our specializer.

Our run-time specializer 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 is to produce specialized code for a *unit* (a generalization of a basic block that has a single entry but multiple possible exits) given the *context*, i.e., 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 describing units and their connectivity, and generating the initial call to the `Specialize` function with the initial unit and context at the entries to dynamically compiled code.

The `Specialize` function first consults a cache to see if code for the unit and entry context being specialized has already been produced (using the unit's caching policy to customize the cache lookup process), and reuses the existing specialization if so. If not, the unit's `ReduceAndResidualize` function is invoked to produce code for the unit, specialized to the input context. The updated values of the context at each of the program points corresponding to unit exits is 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 the 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 at that edge that will call back into the specializer when executed. Points of dynamic-to-static promotion always correspond to lazy edges between units; code is generated at these lazy edges that will inject the promoted run-time values into the context before invoking the specializer.

The caching structure for units is one of the chief points of flexibility in our system. Each of the variables in the context has an associated policy (`CacheAllUnchecked`, `CacheAll`, `CacheOne`, and `CacheOneUnchecked`, listed in decreasing order of specialization aggressiveness), derived from user annotations and static analysis. `CacheAllUnchecked` variables are considered to be so rapidly changing that there is no value in checking and maintaining a cache of specializations; each time the unit is specialized, a new version of code is produced, used, and thrown away. For `CacheAll` variables, the system caches each

```
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);
        Emit("pc := Specialize(`edge.target_unit`,
                               promoted_context,
                               NULL)");
        Emit("jump pc");
      endif
    endfor
  endif
  /* make the predecessor unit branch to this code */
  Backpatch(backpatch_addr, start_addr);
  return start_addr;
}
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;
  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 3: Run-Time Specializer, Part I

```
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 4: Run-Time Specializer, Part II

combination of those variables 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 combination of values is not likely to recur. The values of `CacheOneUnchecked` 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, by skipping cache lookups and stores if any variable in the context is `CacheAllUnchecked`, and otherwise by first performing a lookup in an unbounded-sized cache based on the `CacheAll` variables (if any), and then (if successful) performing a lookup in the resulting single-entry cache based on the `CacheOne` variables, in turn resulting if successful in the address for the appropriate specialized code. `CacheOneUnchecked` variables are ignored during cache lookup.

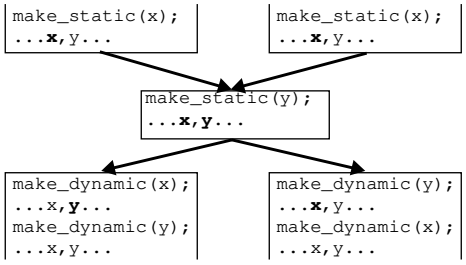
Since invoking the specializer is a source of overhead for run-time specialization, our system performs a number of optimizations of this general structure, principally by producing a specialized version of the `Specialize` function for each unit. Section 6 describes these optimizations in more detail.

4 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 5 and 6). Appendix A specifies the syntax of our annotations, expressed as extensions to the standard C grammar rules [Kernighan & Ritchie 88].

4.1 `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, indicating that each of them is to be treated as a run-time constant at all subsequent program points until reaching either a `make_dynamic` annotation that lists the variable or the end of the variable's scope (which acts as an implicit `make_dynamic` annotation). 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 together 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 in form for dynamic regions is one major difference between our system 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 places `make_dynamic` annotations for the listed variables at each of the exits of the compound statement.

4.2 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 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 get finer control over dynamic compilation for better performance.

Table 1: Policies

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
`cache_all_unchecked`	specialize at merges, assuming that the context is different than any previous or subsequent specialization
`cache_all`	cache each specialized version at merges
`cache_one`	cache only latest version at merges, throwing away previous version if context changes
`cache_one_unchecked`	cache one version, and assume the context is the same for all future executions of this merge
`promote_all_unchecked`	specialize at promotion points, assuming that the promoted value is different than any previous or subsequent specialization
`promote_all`	cache all specialized versions at promotion points
`promote_one`	cache only latest version at promotion points
`promote_one_unchecked`	cache one version, and assume promoted value is the same for all future executions of this promotion
`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

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 indicating 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 polyvariant vs. monovariant specialization policy controls whether merge points should be specialized for different values of

a variable flowing in along different merge predecessors. In contrast, promotion points such as `make_static` always perform polyvariant specialization of the promoted value beginning at the promotion point.

4.3 Common Idioms

We designed the annotations to effect particular optimizations, such as specializing for true run-time constants (variables whose values remain invariant after initialization at run time) or multi-way loop unrolling (specializing a loop along multiple dynamic paths as in Figure 1). Hence, these common optimizations can easily be obtained by the use of concise idioms. For example, a region of code may be specialized for the value of a true run-time constant `x` by using the `promote_one_unchecked` policy:

```
make_static(x:promote_one_unchecked);
```

Alternatively, to conditionally unroll loops shorter than some threshold, the following idiom is used:

```
if (n < threshold) make_static(n,i:eager);
for (i=0; i<n; i++) {
  ...
}
```

We achieve a substantial effect with little annotation because the default policies of the annotations induce the most aggressive (safe) level of specialization (polyvariant specialization and division, and automatic dynamic-to-static promotion). We have also placed a significant burden on our analyses in that they must clean up "sloppy" annotations. In this example, the programmer most likely does not require automatic promotion for `n` or `i`, or polyvariant specialization for `n` at control-flow merges. Furthermore, the programmer annotates `i` when it is dead, and probably also does not wish to specialize code following the loop for `i`. However, the programmer is not penalized for specifying more powerful policies than are required, and our system uses live-variables analysis and performs other optimizations (e.g., unit-boundary clustering, described in section 6.3) to minimize the cost of the specified run-time specialization.

Finally, note that `i` is a derived static variable after it is assigned zero, and no dynamic-to-static promotion of `i` is required. However, the loop would not be unrolled if `i` were not annotated because we treat unannotated derived static variables as if they were set to the weakest level of specialization (monovariant specialization and division and manual promotion), to avoid incurring run-time specialization expense without explicit user direction.

4.4 Partially Static Data Structures

Another common idiom is to perform a memory reference operation (reading a variable, dereferencing a pointer, or indexing an array) whose result is intended to be a run-time constant. This occurs, for example, when manipulating a (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 `promote_one_unchecked` annotation:

```
make_static(t:promote_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 (a variable access, a pointer dereference, or an array index expression) can be prefixed with the `static` keyword, 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 = static* p;
... /* later uses of t are specialized for t's value
          at specialization time */ ...
```

The `static` 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 to construct 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 need for explicit programmer annotations to provide information that the automatic analysis was unable to deduce. Other dynamic compilation systems either include an analysis that operates only within a dynamic region and is unsafe in the face of some legal C programs (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 p variable 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 four `static` prefix annotations on the `bytecodes` array index expressions in the example. Tempo follows this sort of approach, at least for fields of `struct` types. This syntactic sugar may be a worthwhile addition to our system.

4.5 Interprocedural Annotations

Run-time specialization normally applies within the body of a single procedure: calls to a procedure `P` from within a specialized region all branch to the same unspecialized version of `P`. `P` itself may have another specialized region in its body, but this break in the specialized code will cause all the different specialized calls of `P` to merge together, only to be split back apart again by the cache checks at the `make_static` annotation at `P`'s entry. 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 to be treated as run-time constants; a division can specify any of the same policies for listed variables that a `make_static` annotation can specify. As described in section 6, for each division, our system'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, our system 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 of any dynamic region continue to invoke the unspecialized version of `P`.

The `constant` prefix to the `specialize` annotation is an (unsafe) assertion by the programmer than the annotated procedure acts like a pure function, returning the same result given the same arguments without looping forever, making externally observable side-effects, or generating any exceptions or faults. Our system exploits this information by calling a constant function from call sites with all static arguments 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 specialization division where all formals are static, since that would leave a zero-argument call in the specialized code whose result was a dynamic value.

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.

5 Analysis of the Annotations

Given the programmer annotations described in the previous section, our system 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 (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 5.* This output is used to produce the generating extension which invokes the run-time specializer, as described in section 6.

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 annotated with the polyvariant division policy that is either not found (i.e., 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 and any derived static variables computed (directly or indirectly) from an annotated variable. The computed set of static variables will be used to determine which computations and operands are static vs. which are dynamic. In addition, the set of static variables is used to index into the run-time specializer caches, and consequently the analysis also computes the appropriate caching policy for each static variable. (For internal purposes, the analysis tracks the set of root annotated run-time constants from which each static variable was computed, directly or indirectly, as described in subsection 5.3.6.)
- The analysis computes those points requiring 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 annotated with the auto-promotion policy.
- The analysis identifies which merge points require polyvariant specialization (called *discordant merges*), because at least one variable 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.

5.1 Procedure Representation

We assume 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].

* 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 $\mathsf{dom}(\mathsf{f})$ returns the corresponding range element. We use $\times$ to construct cross-product domains. We write $D(\mathsf{p})$ to project from the product p the element corresponding to component domain D, and we write $\mathsf{p}[D \rightarrow \mathsf{v}]$ to compute a new product p whose D element has value v. Pow denotes the powerset domain constructor. Note that $A \rightarrow B \subseteq \mathsf{Pow}(A \times B)$.

Domains:

BTA ≡ Division → DivisionInfo

DivisionInfo ≡ StaticVarInfo × Promotions × DiscordantVars

Division ≡ Var → Policies

Var ≡ *finite set of all variables in scope of procedure being compiled*

Policies ≡ DivisionPolicy × SpecializationPolicy ×
PromotionPolicy ×
MergeCachingPolicy × PromotionCachingPolicy ×
LazinessPolicy

DivisionPolicy ≡ {PolyDivision, MonoDivision}

SpecializtionPolicy ≡ {PolySpecialization, MonoSpecialization}

PromotionPolicy ≡ {AutoPromote, ManualPromote}

MergeCachingPolicy ≡ {CacheAllUnchecked, CacheAll,
CacheOne, CacheOneUnchecked}

PromotionCachingPolicy ≡ {CacheAllUnchecked, CacheAll,
CacheOne, CacheOneUnchecked}

LazinessPolicy ≡
{Lazy, SpecializeLazy, LoopSpecializeLazy, Eager}

StaticVarInfo ≡ Var → CachingPolicy × SourceRoots

CachingPolicy ≡ {CacheAllUnchecked, CacheAll,
CacheOne, CacheOneUnchecked}

SourceRoots ≡ Pow(Var)

Promotions ≡ Pow(Var)

DiscordantVars ≡ Pow(Var)

LiveVars ≡ Pow(Var)

UsedVars ≡ Pow(Var)

MayDefVars ≡ Pow(Var)

Specializations ≡ Proc → SpecializationInfo

Proc ≡ *finite set of all procedures in scope of function being compiled*

SpecializationInfo ≡ IsConstant × Divisions

IsConstant ≡ {Constant, NotConstant}

Divisions ≡ Pow(Division)

Constraints:

BTALegal(bta:BTA) ≡
LegalDivisions(dom(bta)) ∧
∀(d,i)∈ bta.
StaticVars(i)⊇dom(d) ∧
∀v∈ StaticVars(i).
(SourceRoots(v, i)⊆dom(d) ∧
v∉ dom(d) ⇒
CachingPolicy(StaticVarInfo(i)(v)) =
CacheOneUnchecked) ∧
Promotions(i)⊆dom(d) ∧
DiscordantVars(i)⊆PolySpecializationVars(d)

LegalDivisions(ds:Pow(Division)) ≡
∀d_1,d_2∈ ds. d_1=d_2 ∨ SeparateDivisions(d_1,d_2)

SeparateDivisions(d_1:Division, d_2:Division) ≡
PolyDivisionVars(d_1)≠PolyDivisionVars(d_2) ∨
∀v∈ PolyDivisionVars(d_1). d_1(v)≠d_2(v)

PolyDivisionVars(d:Division) ≡
{ v∈ dom(d) | DivisionPolicy(d(v)) = PolyDivision }

PolySpecializationVars(d:Division) ≡
{ v∈ dom(d) | SpecializationPolicy(d(v)) = PolySpecialization }

StaticVars(i:DivisionInfo) ≡ dom(StaticVarInfo(i))
SourceRoots(v:Var, i:DivisionInfo) ≡
if v∈ StaticVars(i) then SourceRoots(StaticVarInfo(i)(v)) else ∅

Figure 5: Domains

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)
CostNode ::= Var := Const
MoveNode ::= Var := Var
UnaryNode ::= Var := UnaryOp Var
BinaryNode ::= Var := Var BinaryOp Var
LoadNode ::= Var := * Var
StaticLoadNode ::= Var := `static`* 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 5.

5.2 Prepasses

Our analyses will need to identify those program points where a variable in the scope of analysis 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, 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 are the set of variables that have an earlier definition and a later use (but may temporarily be dead at this point).

Finally, we assume that the interprocedural specialization directives have been processed and represented in the Specializations domain that maps each annotated procedure to a set of divisions given in the `specialize` annotation and specifies 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.

5.3 The Main Analysis

Figures 6, 7, and 8 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 5.2 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 fixpoint 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 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 from the set of annotated variables and any (at least temporarily) dead variables from the set of run-time constants, to avoid unnecessary polyvariant division or specialization. Once a new set of divisions and associated

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(∅, uvs_1, mds_1, d, i) }),
   Merge(lvs_2, { (d_{out},i_{out}) |
    (d,i) ∈ bta ∧ (d_{out},i_{out}) = ProcessStmt(∅, 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
    if *this is a static merge* then Merge(lvs, bta)
    else Merge(lvs, { (d_{out},i_{out}) |
        (d,i) ∈ bta ∧
        pvs = {x_1,...,x_n} ∩ PolySpecializationVars(d) ∩ lvs ∧
        mvs = ({x_1,...,x_n} – pvs) ∩ lvs ∧
        d_{out} = ForgetDeadVars(uvs, d – { (x,p')∈d | x∈mvs }) ∧
        i_{out} =
          if pvs = ∅ then i[DiscordantVars→∅]
          else
            ({ (v, (mp, {v})) | (v,p)∈d_{out} ∧
              mp = if v∈pvs then MergeCachingPolicy(p)
                   else CachingPolicy(StaticVarInfo(i)(v)) },
            ∅, pvs) })

Figure 6: Flow Functions, Part I

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.

5.3.1 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 constant variables. 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.

* 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.

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)}) ∧
    i_{out} = MakeStatic(x, d_{out}, i[DiscordantVars→∅]) })

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 }) ∧
    i_{out} = i[DiscordantVars→∅] })

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

BTA_{OpNode} [[`x := static* p`]] lvs uvs mds sp bta ≡
  Merge(lvs, { (d_{out},i_{out}) |
    (d,i) ∈ bta ∧
    (d_{out},i_{out}) = ProcessAssignment(
      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(∅, 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(
      x,
      {y_1,...,y_n}⊆StaticVars(i) ∧
        f∈dom(sp) ∧ IsConstant(sp(f)) = Constant,
      ∪$_{y_i \in \{y1,...,yn\}}$ SourceRoots(y_i,i),
      uvs, mds, d, i) })

Figure 7: Flow Functions, Part II

```
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 ≡
  let si = { (v, (p,rvs))∈StaticVarInfo(i) | v∈lvs } in
    i[StaticVarInfo→
      { (v, (p,rvs))∈si | rvs⊆dom(d) } ∪
      { InitialBinding(v, d) |
        (v, (p,rvs))∈si ∧ v∈dom(d) ∧ ¬(rvs⊆dom(d)) }]
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}, ∅)
ProcessAssignment(v:Var, rhs_is_static:bool, rvs:SourceRoots,
                  uvs:UsedVars, mds:MayDefVars,
                  d:Division, i:DivisionInfo
                  ):Division × DivisionInfo ≡
  if rhs_is_static
  then ProcessStmt({(v,(CacheOneUnchecked,rvs))}, mds, uvs, d, i)
  else ProcessStmt(∅, mds ∪ {v}, uvs, d, i)
ProcessStmt(static_assigns:StaticVarInfo, dyn_assigns:Pow(Var),
            uvs:UsedVars, d:Division, i:DivisionInfo
            ):Division × DivisionInfo ≡
  (d_out,i_out) where
    ps = MayPromotedVars(d, dyn_assigns)
    d_out = ForgetDeadVars(uvs,
            ForgetDynVars(dyn_assigns − ps, d))
    ps_out = ps ∩ dom(d_out)
    si = StaticVarInfo(i)
    si' = si − { (v,vi)∈si | v∈dom(static_assigns) } ∪ static_assigns
    si_out = ProcessDynAssigns(
             si', dom(static_assigns), dyn_assigns, d_out)
    i_out = (si_out, ps_out, ∅)
MayPromotedVars(d:Division, vs:Pow(Var)):Promotions ≡
  { v∈vs | v∈dom(d) ∧ PromotionPolicy(d(v)) = AutoPromote }
ProcessDynAssigns(si:StaticVarInfo, svs:Pow(Var), dvs:Pow(Var),
                  d:Division):StaticVarInfo ≡
  si − { (v, (p,rvs))∈si | v∈dvs ∨ (v∉dom(d) ∧ rvs∩(svs∪dvs)≠∅) }
    ∪ { InitialBinding(v, d) | v∈dom(d) ∧ v∈dvs }
ForgetDeadVars(uvs:UsedVars, d:Division):Division ≡
  { (v,p)∈d | v∈uvs }
ForgetDynVars(vs:Pow(Var), d:Division):Division ≡
  { (v,p)∈d | v∉vs }
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 8: Helper Functions

5.3.2 `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, replacing the policies associated with the variable in some division if already present. 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.

5.3.3 Assignment and Store Nodes

The various forms of assignment nodes all have similar analysis, dependent only on whether or not 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 as `static` by the programmer. A call to a procedure annotated by the programmer as `constant` yields a run-time constant if all its arguments are. A store instruction has no definitely assigned result variable, only potential side-effects as described by the MayDefVars set.

These properties are summarized into a (singleton or empty) set of variables definitely assigned run-time constant values and 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 the system to insert a dynamic-to-static promotion automatically if they ever get assigned a dynamic value), and those that aren't auto-promoted, which are dropped 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.)

5.3.4 Merge Nodes

Ignoring the definition and analysis of static merges for the moment, 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 5.2). For those discordant variables that the programmer annotated as run-time constants with a polyvariant specialization policy, the analysis will mark this merge as a discordant merge 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. (As usual, 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.

Static merges are merges where at most one of the merge's predecessors can appear 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; hence each of the input divisions is passed through unchanged without introducing any cache check points. Subsection 5.4 explains how static merges are identified.

5.3.5 Other 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.

5.3.6 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 discordant merges, 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, which are checked during cache lookups (unless the user specifies explicitly that no checking is required). 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 whatever caching policy with which the variable was annotated.

Assignments to root annotated variables can break the assumptions that some derived run-time expression is a function of some set of root annotated variables. In such a 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 restored to their regular explicit PromotionCachingPolicy value. The analysis tracks the set of root annotated variables on which a derived run-time constant depends, and 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).

5.3.7 Additional Lattice Meet Operations

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

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

Elements of Policies are met pointwise. Elements of individual policy domains are totally ordered, with elements listed earlier in the set of alternatives for a domain in Figure 5 ordered less than elements listed later; for example:

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

Thus, the lattice meet operator for a particular policy domain returns its minimum argument, e.g.:

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

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

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

$$\begin{aligned}
& si_1 \cap_{\text{StaticVarInfo}} si_2 \equiv \\
& \quad \{ (v, (p,rvs)) \mid v \in \text{dom}(si_1) \cup \text{dom}(si_2) \ \wedge \\
& \qquad\qquad p = p_1 \cap_{\text{CachingPolicy}} p_2 \ \wedge \\
& \qquad\qquad rvs = rvs_1 \cup rvs_2 \\
& \qquad \text{where } p_2 = \text{if } v \in \text{dom}(si_2) \text{ then CachingPolicy}(si_2(v)) \\
& \qquad\qquad\qquad\qquad \text{else CacheOneUnchecked} \\
& \qquad\qquad p_1 = \text{if } v \in \text{dom}(si_1) \text{ then CachingPolicy}(si_1(v)) \\
& \qquad\qquad\qquad\qquad \text{else CacheOneUnchecked} \\
& \qquad\qquad rvs_1 = \text{if } v \in \text{dom}(si_1) \text{ then SourceRoots}(si_1(v)) \text{ else } \varnothing \\
& \qquad\qquad rvs_2 = \text{if } v \in \text{dom}(si_2) \text{ then SourceRoots}(si_2(v)) \text{ else } \varnothing \ \}
\end{aligned}$$

$$vs_1 \cap_{\text{Promotions}} vs_2 \equiv vs_1 \cup vs_2$$

$$vs_1 \cap_{\text{DiscordantVars}} vs_2 \equiv vs_1 \cup vs_2$$

5.4 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 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. Reachability conditions are computed at the same time as the *BTA* information, since it depends on the *BTA*'s division and static variable analysis and influences the *BTA* analysis's treatment of merge nodes. Further details on reachability analysis can be found in an earlier paper [Auslander *et al.* 96].

6 Generating the Run-Time Specializer

Given the output of the *BTA* analysis, our compiler 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 are performed:

- 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. Points can begin to be replicated at entry to specialized functions (producing several distinct functions), and at merge points where different divisions combine. Replicated paths can remerge at points where divisions cease to differ and are joined by the Merge function.
- The compiler identifies which branch successor edges need to be lazy specialization edges. Subsection 6.1 discusses this in more detail.
- The compiler identifies the boundaries of the units manipulated by the run-time specializer (described in section 3). Unit boundaries primarily correspond to dynamic-to-static promotion points, demotion points (where variables are removed from the set of annotated variables), discordant merges, and lazy branch successor edges. The first three cases are cache lookup points, and the last case avoids speculative specialization. This process is described in more detail in subsection 6.2 below. A clustering algorithm then attempts to merge boundaries together to minimize their cost, as described in subsection 6.3. The `Unit` and `UnitEdge` specializer data structures are generated at the end of this process.
- 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 6.4 discusses some issues with this separation in more detail. We apply standard compiler optimizations, including instruction scheduling and register allocation, to each subgraph separately. (We perform higher-level target-independent optimizations such as common-subexpression elimination and loop optimizations before our *BTA* analysis.) Performing these regular compiler optimizations over both statically compiled and dynamically compiled code is crucial for generating high-quality code [Auslander *et al.* 96].

- Finally, each unit's `ReduceAndResidualize` function is produced. First, the control-flow and the reduce operations of the `ReduceAndResidualize` function are derived from the static control-flow subgraph, after removing all dynamic branches from the static subgraph; this process is described in more detail in subsection 6.5. Then 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 6.6 below. The resulting subgraph forms the `ReduceAndResidualize` function for the unit, and the dynamic subgraph is thrown away.

Some optimizations of the calls to the run-time specializer are discussed in subsection 6.7.

6.1 Computing Lazy Branch Successors

Laziness policies on variables indicate the extent of speculative specialization after dynamic branches that should be performed. A branch successor is a lazy edge 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 discordant 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 discordant loop head merge node where at least one of the discordant variables is annotated with the LoopSpecializeLazy policy.
- 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 points where it is determined that the downstream program point will eventually be executed.

Once the dominator information relating program points is computed, a linear scan over the dynamic branches, discordant merges, and specialized calls serves to compute the lazy edge information.

6.2 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.

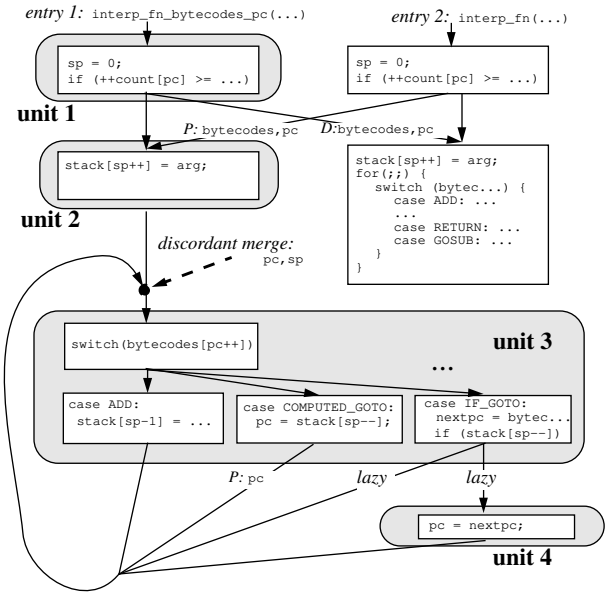


Figure 9: Specialization Units for Figure 2

- 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 predecessor point(s) is a unit boundary, since different degrees of polyvariant specialization or of cache retention can occur.
- 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 induces a unit boundary.
- Each edge labelled as a lazy edge introduces a unit boundary.

In addition, units are constrained to be single-entry regions, so if any units would otherwise have multiple entry points, additional unit boundaries are inserted at control-flow merges of paths with different unit entries. It is possible for a program point to be a boundary in more ways than one; only a single boundary results.

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 9 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.

The specialized entry point begins unit 1, and the true branches of the tests merge at the code to be specialized forming unit 2, which is created due to 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 and sp, which have definitions both inside and outside the loop, are discordant at its head. A promotion of pc is required on the back edge from the COMPUTED_GOTO case after pc is assigned a dynamic stack location. The successors of the dynamic branch in the IF_GOTO case are made *lazy* as required by the (default) loop_specialize_lazy policy because the branch determines execution of different paths to the discordant 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 discordant 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.

6.3 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 demotion 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 moved legally. Discordant-merge and lazy-edge boundaries cannot be moved, so their range is a single program point. Promotion and demotion boundaries can move to any control-equivalent [Ferrante *et al.* 87] program point bounded by earlier and later uses of any promoted or demoted variable, except that 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 we make a linear scan through this sorted list. We remove the range that ends first in the list, remove all other ranges that overlap with the first range, 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 demotions 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 demotions, 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.

We have proved that this algorithm for coalescing boundary ranges is optimal, given the restricted kinds of ranges produced in the first step (the restriction to control-equivalent program points is key).

Different kinds of boundaries incur different kinds of costs. Eager boundaries incur cost only at specialization time. Lazy-edge boundaries incur cost at run-time, but only once the first time that boundary is executed, since the edge is patched to branch directly to the specialized successor code when first invoked. Promotion boundaries require run-time cost each time they are executed. We do not wish to cluster boundaries with different kinds of cost together if that would increase overall expense; for example, we do not wish to cluster an eager cache lookup boundary with a lazy edge to form a lazy cache lookup that would incur run-time cost at each execution. A simple strategy is to treat each kind of boundary separately in the clustering algorithm, running the clustering algorithm three times, one for each class of boundary. A more sophisticated strategy would allow eager caching to be combined with lazy caching, and lazy edges to be combined with lazy caching, only avoiding merging eager caching with lazy edges.

6.4 Separating Static and Dynamic Operations

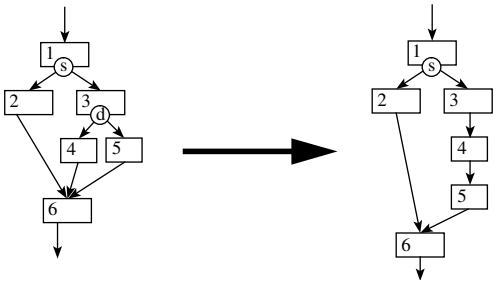
For most straight-line operations, it is clear whether the operation is static or dynamic. However, call instructions are trickier.

- 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 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 static arguments (as determined by the division of the callee), and returning a static procedure address. This is followed by a dynamic call invoking the static procedure address and passing 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 will address them.

6.5 Linearization within Units

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 is computed from the static subgraph. 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; similarly, 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. The following diagram illustrates how the dynamic branches are linearized:



* Some other systems, such as Tempo, perform interprocedural binding time analysis and so can deduce that the result of the a specialized function is static. If we were to extend our system 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.

In the presence of arbitrary unstructured control flow of mixed static and dynamic branches, this linearization process may require some code duplication to avoid maintaining specialization-time data structures and overhead. Details of our linearization algorithm are available in an expanded version of this paper [Grant *et al.* 97].

6.6 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 single static counterpart. The mapping is updated as the static subgraph is linearized and some blocks are replicated, and as the subgraphs are optimized through instruction scheduling. To integrate the two subgraphs, for each dynamic block, code is appended to each corresponding static block to emit the instructions of the dynamic block, after the static code has computed any run-time constants used in the dynamic instructions.

The code to emit a dynamic instruction embeds the values of any run-time constant operands into the generated instruction (either as a short integer immediate field or as a load from a global table to a scratch register for large integers, floating-point numbers, and pointers). The emitting code also performs any peephole optimizations of the generated instruction based on the run-time constant value, such as replacing multiplications by constants with sequences of shifts and adds.

6.7 Optimizing Specializer Interactions

Each initial promotion point entering 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 3 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.
- Calls to the `Specialize` function corresponding to lazy edges with no change in cache context or promotions are dynamically overwritten to be direct jumps (or fall-throughs) to the dynamically generated code for the target unit.
- Demotions corresponding to the ends of dynamic regions are compiled into direct jumps to statically compiled code.

7 Comparison To Related Work

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

- Our system 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 loop unrolling.
- Our system 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.
- Our system allows dynamic-to-static promotions to occur within dynamically compiled code. Tempo requires such promotions to occur only at the entry point.
- Our system 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.
- Our system 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.
- Our system supports separate compilation while still being able to specialize call sites and callee functions. Tempo requires the whole module being specialized to be analyzed and compiled as a unit.
- Our system is currently under construction; Tempo is implemented and usable.

Fabius [Leone & Lee 95] is another dynamic compilation system based on partial evaluation. Fabius is more limited than our system 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 our system, it does not explicitly construct an explicit dynamic subgraph that can then be optimized.

Compared to our previous system [Auslander *et al.* 96], our current system 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 our system. In particular, C-mix is a partial-evaluation system for C that provides program-point polyvariant specialization [Andersen 92], but not polyvariant division. C-mix copes directly with unstructured code, but it appears to lack reachability analysis to identify static merges [Andersen 94]. C-mix also includes support for automatic interprocedural call graph, alias, and side-effect analysis, although partially static data structures are not supported.

Schism's filters permit choices about whether to unfold or residualize a function and which arguments to generalize, 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. Our system'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, for example, to prevent unbounded unfolding and unbounded specialization. Both off-line partial evaluators, such as Schism, and on-line 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, as our system does.

`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, our system's ability to perform arbitrary and conditional polyvariant division and specialization enables it perform a wide range of optimizations with very little user intervention, and our system 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 loop unrolling within a tick expression [Poletto *et al.* 97].) Also, tick expressions cannot contain other tick expressions, so nested (not to mention 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 our system by specializing a procedure based on the length and values in its `varargs` pseudo-argument.

A declarative system such as ours 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.

8 Conclusions

We have designed 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 partial evaluation process when needed. Our support for arbitrary program-point-specific polyvariant division and specialization is a key component of our system'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 and nestable dynamic-to-static promotion and demand-driven specialization.

We are currently in the process of implementing this design, in the context of the Multiflow compiler [Lowney *et al.* 93]. Once complete, we plan to focus on gaining experience applying dynamic compilation to sizeable, real application programs and extending our system to provide some form of automatic alias and side-effect analysis, interprocedural binding-time analysis, and additional run-time optimizations.

Acknowledgments

We are grateful to the anonymous PEPM'97 referees for their suggestions for refocusing this paper, and Charles Consel for his help in understanding Tempo and some of the related issues in partial evaluation. We also thank 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 92] L.O. Andersen. Self-Applicable C Program Specialization. pages 54–61, June 1992.

[Andersen 94] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, 1994. 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, St. Petersburg, Florida, January 1996.

[Consel 93] C. Consel. A Tour of Schism: A Partial Evaluation System for Higher-Order Applicative Languages. 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. volume 1110 of *Lecture Notes in Computer Science*, pages 54–72, 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, Austin, Texas, January 1989.

[Engler & Proebsting 94] D. R. Engler and T. A. Proebsting. DCG: An Efficient, Retargetable Dynamic Code Generator. In *Sixth International Conference on Architectural Support for Programming Languages a nd Operating Systems (ASPLOS-VI)*, 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, St. Petersburg, Florida, 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*

Lanaguge and its Implementation. Addision-Wesley, Reading, MA, 1983.

[Grant *et al.* 97] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. Annotation-Directed Run-Time Specialization in C. Technical Report UW-CSE-97-03-03, University of Washington, April 1997.

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

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

[Lindholm & Yellin 97] T. Lindholm and F. Yellin. Inside the Java Virtual Machine. *Unix Review*, 15(1):31–32, January 1997.

[Lowney *et al.* 93] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. ODonnell, and J. C. Ruttenberg. The Multiflow Trace Scheduling Compiler. *Journal of Supercomputing*, 7(1), May 1993.

[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*, page To Appear, June 1997. In Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.

[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, St. Petersburg, Florida, January 1996.

[Weise *et al.* 91] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In *Functional Programming & Computer Architecture*, June 1991.

[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
  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:
  `cache_all_unchecked`
  `cache_all`
  `cache_one`
  `cache_one_unchecked`

promotion-caching-policy:
  `promote_all_unchecked`
  `promote_all`
  `promote_one`
  `promote_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 */
  `static *` *expression*

primary:
  `...` /* same as in regular C */
  `static` *identifier*
  primary `static [` *expression* `]`
  lvalue `static .` *identifier*
  primary `static ->` *identifier*