Annotation-Directed Run-Time Specialization in C

Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers and Susan Eggers
Department of Computer Science
University of Washington

Abstract

Dynamic-compilation systems have achieved significant speedups on a number of modest-sized applications, and are now turning towards realistic programs. Most dynamic-compilation systems are based on partial-evaluation techniques that are applied at run time, relying on declarative annotations to guide specialization. Our approach to run-time specialization uses simple, expressive annotations to provide polyvariant specialization and division for C programs. Our system, DyC, specializes code at the sub-function level in direct style, which is important for efficiently specializing existing large C programs with only minimal effort by the programmer. To make run-time PE practical for large applications, we use specialization policies to limit the run-time cost and manage space usage. We have completed the design of the binding-time analysis and of the construction of generating extensions, and are currently implementing the system.

Keywords: dynamic compilation, partial evaluation, program specialization, compiler, annotations, run-time specialization

Email: {grant,mock,matthai,chambers,eggers}@cs.washington.edu

Department of Computer Science & Engineering
Box 352350
University of Washington
Seattle, WA 98195-2350
USA

Corresponding Author:

Brian Grant
Phone: +1-206-616-1854
Fax : +1-206-543-2969
grant@cs.washington.edu

N.B. We realized less than three weeks before the conference deadline that we had work that we felt would be interesting to the PEPM community. This paper was furiously written in the time since, and while it represents our best effort, it does not represent our best work. Should the paper be accepted based on the merit of its ideas, we promise the final version will be much more polished. Thank you for your consideration.

Annotation-Directed Run-Time Specialization in C

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

Department of Computer Science
University of Washington

Abstract

Dynamic-compilation systems have achieved significant speedups on a number of modest-sized applications, and are now turning towards realistic programs. Most dynamic-compilation systems are based on partial-evaluation techniques that are applied at run time, relying on declarative annotations to guide specialization. Our approach to run-time specialization uses simple, expressive annotations to provide polyvariant specialization and division for C programs. Our system, DyC, specializes code at the sub-function level in direct style, which is important for efficiently specializing existing large C programs with only minimal effort by the programmer. To make run-time PE practical for large applications, we use specialization policies to limit the run-time cost and manage space usage. We have completed the design of the binding-time analysis and of the construction of generating extensions, and are currently implementing the system.

1 Introduction

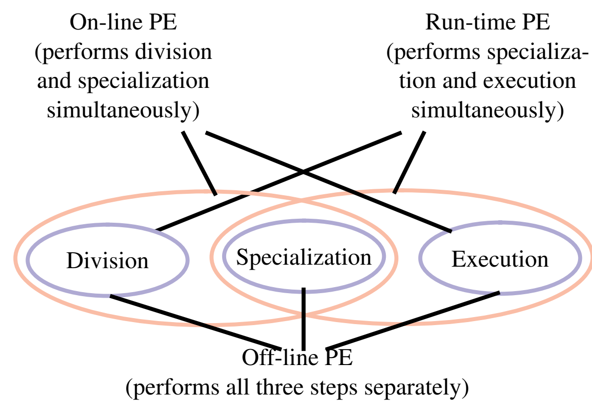
Prototype dynamic-compilation systems have achieved significant speedups on a wide range of modest-sized applications, including simple interpreters, sorting programs, pattern matchers, and network packet filters. Results on these preliminary examples have been encouraging enough that work in the area is ready to move beyond the prototypal stage. Several groups have begun to consider the problem of speeding up large, realistic programs using dynamic compilation.

The types of applications listed above are notably the same as those known to be amenable to partial evaluation (PE). This should not be surprising because PE is effective at eliminating the overheads induced by parameterization and interpretation. Moreover, general-purpose dynamic-compilation systems to date have depended solely upon run-time knowledge of the values of variables to perform optimizations, and these value-specific optimizations are a natural application of program specialization.

Most, but not all, dynamic-compilation efforts to date have focused on run-time program specialization guided by declarative annotations, rather than imperative-style program construction. The declarative systems permit the programmer to annotate ordinary code to identify what code to specialize on which values. The imperative systems, on the other hand, require the programmer to invoke primitives that explicitly construct and compose fragments of code. The declarative systems have the following advantages over imperative systems.

- They can be directly applicable to existing programs.
- They are easier to apply in new programs because they automatically construct the specialized code, cache specialized versions, and respecialize as needed.
- They facilitate debugging by preserving the semantics of annotated statically compilable programs.

Because of these advantages, and because of the power of polyvariant specialization to specialize interpreters, we decided to apply run-time specialization to dynamic compilation. We are particularly interested in applying declarative annotations to existing, realistic, large C programs without change in order to facilitate eventual automatic application of dynamic compilation.



Partial evaluation consists of three steps:

1. Division, computing what is static and what is dynamic,
2. Specialization, generating the specialized code based on the results of division, and
3. Execution of the specialized program.

Off-line PE systems (*e.g.*, Mix [JSS89], Schism [Con93]) perform the three steps separately, division and specialization at compile time and, of course, execution at run time. Because division is performed without the actual static values, off-line PEs must be more conservative in classifying expressions as static than on-line PEs (*e.g.*, Fuse [WCR+91]), which perform division and specialization simultaneously. Neither off-line nor on-line PESs may exploit any run-time values other than that static values provided to their specializers. Run-time PEs (*e.g.*, Fabius [LL96], Tempo [EHK96]), while they still perform division at compile time, can exploit run-time values because they interleave specialization and execution at run time.

FIGURE 1. Static vs. Run-time Partial Evaluation

Current run-time-specialization systems are structured as off-line partial evaluators that perform division at static compile time and specialization interleaved with execution at run time (see figure 1). Previous approaches to run-time specialization [APC+96,CN96,EHK96,LL96] were unsuitable for our domain of interest for the following reasons.

- They employed highly restricted forms of polyvariant division, thus limiting the extent of specialization.
- They forced a function-level granularity of specialization on the user that was inappropriate for imperative languages such as C.
- They did not provide enough control over specialization to limit its run-time cost and manage space usage.

We propose a compact, expressive set of annotations, to be implemented in our new dynamic-compilation system for C, DyC, that address the deficiencies of previous run-time-specialization systems, including our own first-generation system [APC+96].

- The annotations make arbitrary (previously dynamic) program variables static at arbitrary program points, prompting interprocedural and intraprocedural polyvariant specialization and division.
- Our binding-time analysis permits intraprocedural specialization of arbitrary, including unstructured and even irreducible, control-flow graphs in the direct style.
- The annotations provide control over specialization through a combination of default and user-specifiable policies, which are used both to avoid over-specialization and also to overcome some of the current limitations of our analyses.

We introduce our annotations and explain how they specify specialization of C programs in section 2. Sections 3 and 4 discuss how our annotations strike a balance between aggressively applying PE capabilities, and providing the user with flexible means to temper the cost of specialization. In section 5, we discuss the implementation of DyC, including the binding-time analysis and construction of generating extensions. Section 6 compares our system to static PE systems, run-time PE systems, and non-PE-based dynamic-compilation systems.

2 Specialization Annotations

We have designed a set of simple, expressive annotations to direct the run-time specialization of C programs. The annotations initiate intraprocedural and interprocedural polyvariant division and specialization of the annotated functions.

We have found, perhaps surprisingly, that while the annotations are easy to apply and possess the other advantages of declarative approaches to dynamic compilation, they are also as expressive as existing imperative approaches to dynamic compilation such as ‘C [EHK96].

The annotations also provide control over the specialization process by permitting the specification of a number of guiding policies. In static PE, specialization time does factor into the execution time. When performing run-time PE, however, it is important that the benefit gained by running a specialized program is not overwhelmed by the cost incurred producing it. Our annotations give the programmer the control to pay the cost only where he or she expects to reap a benefit.

The polyvariant division of formal parameters of functions to be specialized is explicitly specified by our function annotations in order to facilitate separate compilation, an important pragmatic consideration for C programs. Our approach in this aspect resembles *partial signatures* as proposed in ‘C [EHK96]. Furthermore, the explicit division provides control over which functions are in-lined or specialized and on which arguments to specialize, similarly to Schism’s filters [Con93]. While traditional PE systems typically choose to aggressively in-line function calls and specialize on all static arguments [Ses88], run-time specialization requires tighter control over function specialization because of its high cost.

From these explicit divisions of formal parameters and annotations that make arbitrary program variables static at arbitrary program points, our binding-time analysis computes the polyvariant division for each function. We achieve this program-point specialization in direct style without expensive or complicated program transformations inside the compiler. In particular, we do not have to resort to conversion to continuation-passing style [CD91]. The binding-time analysis performs intraprocedural specialization in direct style on the control-flow graph at the granularity of our three-address intermediate representation, and is capable of handling arbitrary, including unstructured, control flow. This capability is crucial to the goal of applying our annotations to existing C programs automatically.

2.1 Intraprocedural Annotations

Our basic intraprocedural annotation, `makeStatic`, is used to make arbitrary program variables static at arbitrary program points. The `makeStatic` annotation begins polyvariant division and specialization on the variables annotated at the annotation point, which persists to the end of the function annotated. We provide a second annotation, `makeDynamic`, that can be used to halt specialization on variables on a particular path through the control-flow graph by forcing them to become dynamic prior to the end of the function.

2.1.1 Syntax and Semantics

The `makeStatic` annotation has two components, a list of variables that are to be specialized and a list of policies, to be discussed in section 4, that are used to guide specialization (see figure 2). `makeDynamic` takes only a list of variables as arguments. Based on the root set of variables annotated by `makeStatic`, further values and computations are identified as static, all others are, by default, dynamic. This division is computed at compile time by a binding time analysis.

```
makeStatic(<variable-list> : <policy-list>);
makeDynamic(<variable-list>);
dynamicRegion <compound-statement>
```

FIGURE 2. Intraprocedural Annotations

The primary purpose of `makeStatic` is to promote a dynamic variable to static, enabling polyvariant division and specialization on that variable. This promotion generally occurs at the site of the `makeStatic` annotation, for live variables not known to be static, but may also occur elsewhere downstream. If a variable annotated by `makeStatic` is assigned a dynamic value, we automatically promote the variable to be static again. We are able to do this by delaying the specialization of that program point until execution reaches it and the dynamic value becomes available.

Note that we specialize on derived static variables only monovariantly in order to make control over the extent of specialization explicit, a topic discussed in depth in section 4. Furthermore, specialization policies are not inherited by derived static variables. This is one reason why it is necessary that `makeStatic` be able to change specialization policies on variables that are already static. We allow the programmer to induce polyvariant division and specialization on a derived static variable using `makeStatic`, for example. This is demonstrated by the example in figure 3.

The loop in the code fragment, where `n` is static, would be unrolled by static PE systems, as `i` is inferred to be static. In our system, because the specialization cost is incurred at run time, it is important to give the programmer the opportunity to decide when he or she thinks that the unrolling will be beneficial. If the programmer wants to unroll a loop, the loop induction variable must be explicitly annotated by the programmer.

Both the automatic dynamic-to-static promotion and the specialization policies of the static variables are in effect from the `makeStatic` annotation point forward, until it is killed by `makeDynamic` or at the end of the function in which it occurs. "Forward" means forward in the sense of non-back edges in the control-flow graph, which roughly corresponds to textual order. `makeStatic` and `makeDynamic` delimit regions of specialization on particular sets of variables, called *dynamic regions*. Code outside dynamic regions is not specialized.

```
makeStatic(i);
for (i = 0; i < n; i++) {
    /* body without assignment to i */
}
```

FIGURE 3. Annotated Loop Example

`makeStatic` and `makeDynamic` make it possible to specify nested or overlapping dynamic regions, which is useful when variables on which the programmer would like to specialize are derived from both static and dynamic values, or when variables the programmer would like to make static are changed at different rates. The dynamic-region concept is useful to the programmer because it makes the bounds of specialization clear. Therefore, we additionally provide a `dynamicRegion` annotation that limits the scope of any `makeStatic` annotations within it. The `dynamicRegion` annotation is merely syntactic sugar for `makeDynamic` annotations on variables made static within the region, placed on each exit from the region. Therefore, we will henceforth only discuss `makeStatic` and `makeDynamic`.

2.1.2 Direct-Style Specialization

Many PE systems have operated in the context of functional languages applying specialization only at the function level. In imperative languages like C, however, functions are often large and opportunities for specialization arise at the sub-function level. Conditional constructs, such as `if` and `switch`, and loops are particular candidates for specialization, and being able to control their specialization is of paramount importance.

For example, in systems that provide only function specialization, loops must be converted to tail-recursive functions to permit unrolling and specialization of the loop body. Specializing arbitrary sub-regions of functions on different variables would require such systems to separate the code to be specialized into separate functions. More importantly, to avoid losing information at control-flow merges, such systems must convert to continuation-passing style (CPS) [CD91], the disadvantages of which are well known [JGS93], and which would be impractical and costly in a typical optimizing C compiler.

DyC performs polyvariant division and specialization on the control-flow graph in direct style. Our binding-time analysis permits variables to remain static beyond merges of paths each containing definitions of the static variables, and the run-time specializer uses this information to generate a specialized version of the code following the merge for each definition. The binding-time analysis is also capable of computing multiple divisions at each program point following a control-flow merge if different divisions of static and dynamic variables reach it. The code following the merge is then replicated for each division at compile time.

2.2 Function Annotations

In the previous sections we showed that intraprocedural annotations are key to specializing imperative programs without forcing the programmer to rewrite or restructure them. On the other hand, procedural abstraction is a powerful concept also used in imperative languages, encouraging a modular programming style. Because functions are typically used in many different contexts, both polyvariant specialization and division of functions are desirable.

The ability to handle separate compilation is another requirement for function specialization, which is imposed on us by the C programming language. Usually, an application in C consists of a number of different files that are compiled separately. Our annotations for function specialization allow us to specialize functions, even in the presence of separate compilation, by explicitly specifying the divisions of the formal parameters of functions to be specialized or in-lined in both the file containing the definition of a function to be specialized or in-lined and in the files containing calls to it. The programmer must ensure, however, that the annotations of the functions in different files are consistent. This policy is consistent with the requirement that function prototypes be compatible with the function implementation itself, and is easy to guarantee by using header files.

When specializing large applications we also need to be able to restrict function specialization. We only want to specialize functions for which we expect to improve overall run time. Moreover, we want to specialize only on those arguments for which we expect the highest benefit. This was also recognized by Consel, who provided so-called filters to control function specialization in Schism [Con93]. A filter has two parts, both of which are user-defined functions. One decides at binding time whether to unfold or residualize the function which it “filters”. The other determines which of the function arguments are to be propagated as static (provided they are static at the call site) into the function body. In DyC, this decision is made by the programmer.

2.2.1 Function Specialization Annotations

Our function annotations for specialization are designed to express all of the requirements identified above. They support:

- polyvariant division and specialization,
- separate compilation,
- control over arguments used for specialization,
- control of function specialization at the call site,
- caching policies to control the code space requirements, and
- extending the set of functions used for the derivation of static values.

To specialize a function on its arguments, the function prototype in a C header file is annotated as shown in figure 4. This annotation must be included in all files that call the function as well as in the file that defines the function.

specialize <return-type>
  $f(a_1, \ldots, a_n)$ on
  $\left(\left(a_{11} \ldots a_{1n_1}\right), \ldots, \left(a_{k1}, \ldots, a_{kn_k}\right)\right)$

FIGURE 4. Function Specialization Annotation

This annotation will produce *k* specialized versions for `f`, specializing on the list of *k* combinations of formal parameters selected from the formal parameter list. For example the annotation shown in figure 5

```
specialize f(a,b,c)
  on ((a), (c), (a,c));
```

FIGURE 5. Function Specialization Example

will produce 3 different specialized versions of `f`:

1. f_a`(b,c)` [1]specialized on `a`,
2. f_c`(a,b)` specialized on `c`, and
3. f_{ac}`(b)` specialized on `a` and `c`.

Sometimes, C functions take a variable number of arguments indicated by the ellipsis `...`, for example as in `f(n, ...)`. Our annotations currently do not permit specialization on any parameters implicit in the ellipsis. In the examples we studied, we found no need for it, as parameters interesting for specialization were explicitly listed in the for-

1. We use the subscript to indicate the arguments on which the function is specialized.

mal parameter list. However, if necessary, a naming mechanism referring to such parameters could be added.

At at call site calling f we select the specialized version that specializes on the largest number of arguments that are static at the call site. If there are different versions with the same number of arguments, the first listed in the annotation is chosen. For example, suppose f is annotated as follows:

```
specialize f(a, b,c, d) on ((a), (a,b),
(b,c))
```

and at f is called as `f(s1, s2, s3, s4)` where all the arguments are static. Then both f_{ab} and f_{bc} specialize on two arguments. The conflict is resolved by choosing the first specialization listed in the annotation, therefore, f_{ab} would be selected for this call site. f_a would not be selected, even though it is listed first, because it specializes on a lesser number of arguments (one). In some cases, the programmer may want to select a more general, that is less specialized, version. `makeDynamic` can be used to achieve this effect. The programmer simply annotates those arguments with `makeDynamic` that are not be used for specialization. To select f_a in the example, we write:

```
f(s1, makeDynamic s2, makeDynamic s3,
makeDynamic s4)
```

The `makeDynamic` annotation inside the function call only refers to the call itself, it does not change the static-dynamic division of the program otherwise.

2.2.2 Function Annotations for Inlining

Inlining function calls (also called function unfolding) is an important optimization that both reduces the function-call overhead and allows the optimization of the function body in the call context. On the downside, it can lead to large code blowup and in the presence of recursive calls care has to be taken that unfolding terminate. In-lining of recursive function calls can be bounded, for instance, by requiring that it be controlled by a static test, a simple strategy chosen by some PE systems [Ses88].

Our annotations allow the programmer to decide which functions to in-line and also control under which conditions they are in-lined.

```
inline <return-type>
        f(a1, ..., an) on
```

$$\left(\left(a_{11} \dots a_{1n_1}\right), \dots, \left(a_{k1}, \dots, a_{kn_k}\right)\right)$$

FIGURE 6. Function Inlining Annotation

Our inline annotation shown in figure 6 specifies that function `f` be in-lined. It has to be put on the function definition of the function to be in-lined. However, in-lining will only take place if the actual arguments at the call site match one of the combinations listed in the list of combination of arguments for which in-lined should be done. A match can always be forced by using `makeStatic` and `makeDynamic` on the arguments of the function call, analogous to the `specialize` annotation. For in-lining, we require that the source code of the function that is to be in-lined, must be available in the same file as the call site where it is to be in-lined. It will allow for further optimization at the call site because all optimizations can then be done across function boundaries even without interprocedural analysis.

3 Applying Partial Evaluation to Dynamic Compilation

3.1 Off-line vs. on-line specialization

Traditional PE and DC systems both specialize their input programs on the values of a subset of their program variables. The primary difference between the two systems is that in traditional PE, the possible values of variables on which specialization should take place (termed *static* variables) are available at compile time whereas in DC systems, the *identity* of the static variables is known at compile time, but their *values* are available only at run time. In DC systems, therefore, specialization cannot occur until runtime.

The fact that specialization occurs at run time and therefore needs to be fast argues for limiting program analysis, specifically that required for dividing program variables into static and dynamic, to compile time, if possible. Further, since the identity of root static variables is known at compile time, the possibility exists of computing a conservative division of program variables into static and dynamic variables before the program is run. The relatively expensive division phase, followed by a fast specialization which essentially just selects and instantiates divisions found statically, is a characteristic of traditional off-line PE systems. An on-line PE system, on the other hand, would force division and specialization to run concurrently, forcing the program analysis necessary for division to move to run time. We do not believe that the increased accuracy of division afforded by the on-line division can compensate for the drastic increase in cost of performing the program analysis (an iterative, interprocedural dataflow analysis, for general C programs) for division at run time. Further, experience in the PE community suggests that the conservativeness of off-line division relative to the on-line version does not result in significant loss of performance in most applications [JGS93]. Finally, in section 3.3 below, we explain how in some cases, aggressive polyvariant division can compensate for the conservativeness of an off-line division phase. We have therefore chosen to structure our system as an off-line (compile-time) division phase followed by a run-time specialization phase.

Once the decision is made to structure a dynamic compiler as an off-line PE system, a natural question is how aggressive the division and specialization phases should be. The design space ranges from monovariance to restricted polyvariance to very aggressive forms of polyvariance in each case. Existing PE-based DC systems have chosen specialization phases that support restricted forms of polyvariance, and division phases that are monovariant or very restricted in polyvariance. Restricting support for polyvariance can result in simpler specializers, and smaller specialization costs at run time. However, in the next two sections, we show that even very simple applications (such as the bare-bones interpreter of figure 8) require very aggressive polyvariant specialization and division for satisfactory performance gains. Our philosophy therefore is not to restrict the support for polyvariance offered by our system, but rather to present the user with an interface (discussed in section 4) to *control* the level of polyvariance exploited, according to the expected returns.

3.2 The need for polyvariant specialization

Polyvariant specialization, the ability to produce different specialized versions of a program corresponding to different values that a static variable can attain at various program points, is an important capability of traditional off-line PE systems [JSS89], [Con93]. In these systems, a static variable can have multiple possible values at a given program point because the variable is updated differently (albeit in a manner that still leaves it static) on different paths leading up to the program point. An important special-case is when the program point in question is the merge at the head of a loop, and the static variable is an induction variable of the loop. Polyvariant specialization at the loop head then results (provided concerns of termination are satisfied) in the complete unrolling of the loop at specialization time. Loop-unrolling is an important capability because a large class of applications, including graphics rendering engines, database query systems, simulators, language interpreters, have interpretation loops within them, which iterate over data-structures that encode a specification of the problem being solved. Fully unrolling the loop eliminates the overhead of fetching, decoding and dispatching on the values in these data structures. In many cases, the structure being interpreted is not available untill run time, precipitating the need for a DC, rather than a traditional PE, system.

```
        i = s;
LOOP:   if (i==0) goto DONE;
        print(i);
        i--;
        goto LOOP;
DONE:
```

Multiple static definitions of `i` reach the loop head, `LOOP`.

FIGURE 7. The role of polyvariant specialization in loop-unrolling.

Figure 7 presents a simple loop that prints integers from `s` down to 1, and demonstrates how polyvariant specialization results in loop-unrolling[1]. At program point `LOOP`, we can determine at specialization time that `i` can have the sequence of values `s,s-1,...,1` (`i` is therefore polyvariant). Instantiating this sequence of values of `i` in copies of the loop body is what results in complete unrolling of the loop. If the specializer does not have the ability to specialize the loop body on these different values of `i`, as would be the case with a monovariant specializer, the loop could not be unrolled.

In the case of figure 7 , however, the loop-unrolling can be effected by a very restricted form of polyvariance, used in the prototype version or our system [APC+96]. We essentially allowed the specializer to specialize polyvariantly only at the merge points of loop back edges with the loop head; all other specialization was monovariant. This restriction limited unrolling to loops that had a single, unique unrollings at specialization time, since if the induction variable had more than one possible value at the end of an iteration, a monovariant merge rule would term the induction variable dynamic and prevent unrolling of the loop. Interestingly, even with this restriction we were able to achieve significant speedups from unrolling interpretation loops in a wide range of simple applications such as a record sorter, an simplified dispatcher for the SPIN operating system [BSP+95][CEA+96], a sparse matrix multiplication routine, and a stack-based polynomial evaluator. The essential similarity between the loop of figure 7 and these applications is that they are all *oblivious* programs [JGS93], that is, their control flow is entirely determined by static variables. It is easy to see that in oblivious programs:

1. Note that because, the loop is bounded by a static variables and its lone induction variable decreases monotonically, the loop is guaranteed to terminate and can therefore be unrolled fully

- The only program points at which static variable exhibit polyvariance are loop merges, and
- Every loop has an unique unrolling at specialization time.

In order to fully unroll a loop in oblivious programs, therefore, it is sufficient to special-case the loop merge for "single-way" polyvariance.

Unfortunately, many important programs that need specialization (and specifically, loop unrolling) are not oblivious. Consider the main interpretation loop in figure 8 for instance. The code in the loop is non-oblivious; that is, its control flow *does* depend on dynamic variables, in the following two ways:

1. The equal-to-zero test condition at `pp0` is dynamic, since the contents of `valuestack` are dynamic.
2. The `pc` variable, which determines the loop exit becomes dynamic due to the assignment at pp2, again because `valueStack`'s contents are dynamic.

In the case that polyvariant specialization is restricted to loop-head merges, point 1 above implies that pc will be labeled dynamic at the merge at `pp1`. The loop cannot therefore be unrolled at all, notwithstanding the special-case rule at the loop-head merge, since the special-case rule works only if the induction variable is static at the loop-head. General polyvariant specialization would be able to specialize on the two possible different values of `pc` at `pp1`, and would term `pc` static at the merge, therefore allowing unrolling of the loop if the `IGOTO` case (point 2 above) did not exist. The need to unroll loops containing non-oblivious code in common cases such as the interpreter therefore suggests that dynamic-compilation systems should offer general polyvariant specialization.

Conventional polyvariant specialization is of no help in preventing `pc` from becoming dynamic in the manner observed in point 2 above. The problem here is that the possible values taken on by the pc at `pp2` are, in general, not all known at any given point in the program execution: the target of a particular indirect jump may be re-computed at arbitrary times during the program run. The traditional static PE model of a single specialization phase per program run, when all possible values of a static variable at a given program point are known cannot therefore term `pc` static.

In a dynamic compilation system, however, we can extend the traditional specialization model to include *on-demand polyvariant re-specialization*. Specialization need not be restricted to happen once per program run. In fact, every time a variable on which specialization is desired changes value, a new specialization phase can be triggered. Dynamic re-specialization allows the interpretation loop of figure 8 to be unrolled in spite of the `IGOTO`; each time a specialization

```
interpret(byteCodeT byteCodes[],
          int argList[], int numArgs) {
   int valueStack[STACK_SIZE];
   int pc = 0, vsp = 0;
   int arg, opcode;

   /* push arguments */
   for (arg = numArgs; arg > 0; arg--)
      valueStack[vsp++] = argList[arg-1];

      /*DyC annotations for polyvariant
        specialization on bytecodes and pc
        at every downstream use*/
      makeStatic(bytecodes : transitive);
      makestatic(pc);

      for (;;) {
      opcode = byteCodes[pc];
      switch (opcode)
         case ADD:
            valueStack[vsp-1] =
                valueStack[vsp] +
                valueStack[vsp-1];
            vsp--; pc++;
            break;
         case NEQ:
       pp0: if (valueStack[vsp] != 0)
               pc = byteCodes[pc+1];
            else
               pc += 2;
       pp1: break;
         case IGOTO:
       pp2: pc = valueStack[vsp--];
            break;
         /* ... */
         case RETURN:
            return valueStack[vsp];
      }
   }
}
```

FIGURE 8. A simple interpreter that requires sophisticated polyvariant specialization

phase unrolling the loop encounters an `IGOTO` instruction, it will stop unrolling that path. If, after the specialization phase, the interpreter executes the code corresponding to the computed jump in the specialized version of the code, and the code corresponding to the newly computed value of the `pc` has not been specialized yet, a new specialization phase is triggered.

Since dynamic re-specialization supports specialization on variables that can take an arbitrary unbounded set of values at any program point, it is the most powerful form of polyvariant specialization possible. As we discuss in section 5 , the extra power comes with some extra design complexity and run-time overhead. However, in view of the fact that the extra power is necessary for handling common programs such as the interpreter in figure 8 , we have made the default mode of the polyvariant specializer in our system capable of re-specialization. However, in recognition of the fact that more aggressive polyvariant specialization results in higher run-time cost, we allow the user to scale down the level of polyvariance using the policy annotations, as described in section 4 . To the best of our knowledge, no other existing PE-based programming interface for dynamic compilation exposes the full range of specialization in a controllable manner; section 6 on related work discusses this issue further.

3.3 The need for polyvariant division

Polyvariant specialization produces different specialized versions of code for different sets of values taken on by static program variables at a program point. Polyvariant division, on the other hand, allows the possibility of a variable being both static and dynamic at a given program point. Code downstream of the program point can be specialized differently to allow for each of these possibilities. The static PE literature documents various numerous advantages of polyvariant division. Below, we discuss two of these factors that argue for polyvariant division in DC systems as well, and then point out a novel use of polyvariant division especially relevant to run-time DC systems.

In programs where a single function is called from many sites, the arguments of the function may have different dynamic/static bindings at each call-site. In order to trigger specialization of the function according to the dynamic/static bindings at each call-site, (call-site specific function specialization), the PE system needs to perform a polyvariant division on the arguments of the called function. The additional flexibility of call-site specialization has been shown to be beneficial in increasing the amount of specialization in static PE systems for functional languages. Though the frequency of function calls is lower in mainstream imperative languages, the use of a modular programming style with significant code reuse results in functions that are called from different contexts.

In situations like the one illustrated in figure 9 , polyvariant division can compensate for the fact that DC systems are restricted to off-line division. An on-line system would know the value of the guard `s` at specialization time, and would label `x` either static or dynamic at `MERGE`, depending on whether `s` proved true or false respectively. An off-line PE system with monovariant division, on the other hand, would always deduce that `x` is dynamic at the `MERGE`. Polyvariant division, on the other hand, would note that `x` could have either a static or a dynamic binding at `MERGE`, depending on the path taken to get there, and setup the specialization phase to produce the corresponding different versions of specialized code. Section 5 describes how we use replication at the merge as a means of conveying the different possible divisions to the specializer.

```
/* s is static and d dynamic */
if (s)
    x = s;
else
    x = d;

MERGE:  /* ...*/
```

FIGURE 9. A situation where polyvariant division makes up for the conservativeness of offline division

Finally, the high run-time cost of specialization led us to explore means by which our annotations could allow the user to perform specialization conditionally. The intention is that specialization on a variable over a given part of the program is triggered only if some run-time-computed test-condition is met. Figure 10 shows how we use a combination of conditional specialization, and the per-program point granularity of the makeStatic annotation to achieve conditional specialization.

```
{
    ...x...
    /* Say x is dynamic here
       Specialize only if a
       (possibly dynamic)
       condition holds */
    if (condition(x,...))
       makeStatic(x);
 MERGE:
    ...x...
}
```

FIGURE 10. Using per-program point annotations and polyvariant division to effect conditional specialization

Polyvariant division simply conveys to the specializer that `x` is static or dynamic at the merge depending on whether the test condition held or not. Again, in practice, this is implemented by replicating the CFG downstream of `MERGE` into two versions, one with a static `x` and the other dynamic. At run time, the specializer would have to specialize one copy of each replica. In recognition of the excess specialization involved, we allow the user, through the policy part of our annotations, to create replicas only on demand if there is a chance that many useless replicas are created.

As with specialization, though, our philosophy towards the possibly high run-time cost of sophisticated PE capabilities is not to jettison them altogether, but to provide the user with an interface to control them. Further, though the design of the system would most likely be simpler if it offered more restricted capabilities, we expect to encounter enough uses for the more powerful capabilities as to make them a practical necessity.

4 Directing Specialization

In static PE systems, ignoring the impact on instruction-cache locality, the performance of specialized code can always be at least as good as the unspecialized code because the compiler has more information it can use to optimize the specialized code. Furthermore, the cost of the specialization itself does not factor into run-time performance. Therefore, it is always profitable to produce as many versions of specialized functions as possible, to unroll as many loops as possible, and to in-line as many functions as possible.

However, because we perform specialization at run time, we would like to minimize its cost. Moreover, we would like to avoid specialization when its cost outweighs its benefit. The cost outweighs the benefit when the time spent specializing a function (or smaller segment of code) plus the aggregate time spent executing the specialized function (over all calls) is greater than the time required to execute an unspecialized version.

In DyC, we have attempted to make the run-time specializer very fast (to minimize its cost) while still producing high-quality code (to maximize benefit). Yet, some run-time optimizations (*e.g.*, function in-lining) carry high costs and do not always produce profitable specializations. We rely on our annotations and, currently, the programmer to prescribe beneficial specializations. So, we designed the annotations with the goal that they should be easy to apply manually but yet provide adequate control over the specialization invoked. In this vein, we have also exposed cache management to the programmer to provide a way to limit the space consumed by specialized code.

By limiting the extent of specialization induced by our annotations in straightforward and well defined ways and by offering a range of specialization policies, we provide the means to avoid over-specialization, to manage space usage, and to provide information not deducible by our analysis in order to obtain higher performance. These policies also provide a mechanism for experimentation with different specialization strategies, which we hope to eventually apply automatically.

4.1 Controlling the Extent of Specialization

4.1.1 Explicit Scope of the Annotations

We endow the programmer with the power to precisely control the extent of specialization by making the impact of the annotations clear and by providing mechanisms to the programmer to limit their impact.

- The `makeStatic` annotation initiates global specialization which continues to the end of the function in which it appears. We do not permit loop back edges to carry its effects above the annotation site, so `makeStatic` induces specialization forward from the annotation along the control flow of the function.
- The specialization on the formal parameters of a function induced by `specialize` or `inline` affects the entire function.
- The `makeDynamic` annotation may be used to halt specialization on a static variable or formal parameter prior to the end of the function.

Thus, the intraprocedural bounds of specialization are simple and clear. Specialization may extend interprocedurally, however, so rules are needed to determine what specializations are produced. A function may be in-lined or specialized on arguments that are static at the call site, but the `specialize` and `inline` function annotations explicitly specify the extent of interprocedural specialization by indicating which functions should be specialized or in-lined and on which formal parameters they may be specialized. The programmer may prevent specialization on certain arguments at a particular call site by forcing them to be dynamic with `makeDynamic`. If all arguments are made dynamic then the call is not specialized (or in-lined) at all.

4.1.2 Weaker Specialization

In addition to the explicit scope of `makeStatic`, `specialize`, and `inline` as described above, we can limit the extent of their influence by specifying weaker forms of specialization. As described in section 3, the default semantics of `makeStatic`, `specialize`, and `inline` are to invoke polyvariant division and specialization with respect to the specified variables. This default policy is called `promoteStatic`, and also causes the specified variables to be automatically promoted to static again should they become dynamic. So, only the explicit bounds on the annotations limit their extent under the `promoteStatic` policy. We also provide the policies `polyvariant`, which is like

Policy	Multiple Specializations of the Region	Multiple Specializations of All Program Points	Polyvariant Division	Automatic Dynamic-to-Static Promotion
`monovariant`	Yes	No	No	No
`polyvariant`	Yes	Yes	Yes	No
`promoteStatic`	Yes	Yes	Yes	Yes

FIGURE 11. Polyvariance Policies

`promoteStatic` without automatic dynamic-to-static promotion, and `monovariant`.

The `monovariant` policy indicates that although different versions of the dynamic region begun by the `makeStatic` annotation will be created for different incoming values of the specified variables, the region should only be specialized monovariantly with respect to those variables. This is similar to the intraprocedural specialization performed by most PE systems for functional languages, and causes no intra-region code replication. That is, specialization on the variables specified will cause no merges to be split and no loops to be unrolled. The specialization policy for derived static values is `monovariant`, restricting code replication to only those variables that are explicitly annotated. Hence, only loops whose induction variables are annotated with the `polyvariant` or `promoteStatic` policies are unrolled. This policy was chosen, again, for clarity, and also because of the expense of loop unrolling.

The issues of specialization, division, and automatic dynamic-to-static promotion are really independent, and we could have provided distinct sets of policies for each. Or, we could have provided another intermediate policy that indicated polyvariant specialization but monovariant division. However, we think the two extremes, `monovariant` and `promoteStatic`, will be the most useful points in the spectrum, and so provided them plus one intermediate policy. Furthermore, any of the weaker policies may be simulated by `promoteStatic` and judiciously placed `makeDynamic` annotations. The weaker policies are included for convenience.

4.1.3 Conditional Specialization

Being able to specify what loops should be unrolled does not provide enough control in some cases. It is likely that in some situations it will be profitable to unroll a loop and in others it will not. Fortunately, we can use the conditional specialization mechanism described in section 3.3 to conditionally unroll loops. Figure 12 demonstrates how to use conditional specialization to unroll a loop only when the number of iterations is below a certain threshold.

```
makeStatic(n : monovariant);
if (n < UNROLL_LIMIT)
   makeStatic(i : polyvariant);
for (i = 0; i < n; i++) {
   /* ... */
}
```

FIGURE 12. Conditional Loop Unrolling

At the merge following the `if` statement, the polyvariant division on `i` causes two versions of the code that follows to be created. If `n` is less than `UNROLL_LIMIT` then the loop is polyvariantly specialized on `i`, which unrolls the loop. Otherwise, the loop is monovariantly specialized on `i` because `i` is merely a derived static variable. Therefore, when `n` is greater or equal to `UNROLL_LIMIT`, the loop is not unrolled.

4.1.4 Lazy Specialization

Another way in which we can reduce the cost of specialization is to ensure no unnecessary specialization is performed. Eager specialization specializes as much code as possible, which is until another dynamic-to-static promotion is required. Eager specialization is efficient and is beneficial when it is likely that all specialized code produced will be execute. In contrast, lazy specialization postpones specialization until it is known that the specialized code will definitely be executed, which avoids wasted effort but incurs a higher cost than eager specialization in more frequent switches between execution and specialization.

We have provided the capability to lazily specialize any portion of a dynamic region through policies that specify varying degrees of laziness and through additional annotations that indicate code should be eagerly or lazily specialized. The policies with respect to the degree of laziness of specialization we provide include `eager`, `loopLazy`, `poly-`

Policy	Dynamic Branches Specialized Lazily
`eager`	None
`loopLazy`	**Earliest dynamic branches that postdominate discordant loop-head merges**
`polyLazy`	Earliest dynamic branches that postdominate any discordant merges
`lazy`	All

FIGURE 13. Laziness Policies

`Lazy`, and `lazy`. The `eager` policy directs the specializer to be as eager as possible, which is the right policy if it is likely that all of the specialized code will be executed.

The `loopLazy` policy is actually the default, and indicates that specialization should be eager except for loop unrolling and function in-lining, which should be performed lazily, thus postponing what are likely to be the most expensive specializations. The `polyLazy` policy specifies that all polyvariant specialization should occur lazily. That is, no new specialized versions should be generated unless they are needed. The final policy, `lazy`, indicates all code should be specialized lazily.

Code that must be specialized lazily can be specialized as soon as the predicate of the earliest dynamic branch that postdominates the code is evaluated, at which point we can be sure the code will be reached. This observation allows us to specialize larger pieces of code when some lazy specialization is mandated and allows us to use the dynamic branches already present in the code to determine what should be specialized. Figure 13 summarizes how the policies select which dynamic branches should be used to control lazy specialization.

Like the other policies, these are specified for particular static variables. Which policies are actually applied when multiple policies have been specified for different variables depends on which variables are relevant at the dynamic branch in question. The policies of only the live static variables are considered, and lazier policies have precedence over more eager ones.

We also provide a means to override the global laziness policy. The `makeEager` annotation makes the paths from a dynamic decision be specialized eagerly, and `makeLazy` induces lazy specialization for the decision. These annotations may be applied to the `while`, `for`, `if`, and `switch` statements and the `?:`, `||`, and `&&` operators.

4.2 Cache Management

4.2.1 Limiting Memory Consumption

In addition to limiting the amount of specialized code produced as described above, there may be a need to limit the amount of specialized code stored in the code cache. We could provide a way to specify a hard limit on the space allocated to the run-time specializer, but it would be difficult for the programmer to predict how many specialized versions could be stored in the cache, and therefore the impact of the limit on performance. The reason is that because of peephole optimizations, loop unrolling, and function in-lining, the sizes of the specialized versions can vary greatly.

Unfortunately, limiting only the number of specialized versions of code cached for reuse makes it difficult to predict how much space will be consumed by the cache. Because we are currently more interested in studying the performance aspects of dynamic compilation, we have chosen the latter strategy. We provide a mechanism to limit the number of versions cached, rather than a hard limit.

4.2.2 Bounding the Number of Cached Versions

The cache is only accessed at cache lookups, which are required where one of several specialized versions of the code could be needed. The lookup determines whether to retrieve and use a version from the cache, in the event the lookup succeeds, or to respecialize the code on the current values of the static variables and store the new version in the cache. Thus, it is at cache lookups that the cache policy is applied.

Cache lookups are necessary at dynamic-to-static promotions (*e.g.*, at a `makeStatic` annotation) and at certain flow-control merges, called *discordant merges*, where static variables could acquire any of a number of static values on different paths to the merges[1]. We imagine that typically a small bound will be desired on the number of versions cre-

Policy	Which Versions Are Cached
`cachePromotions(1)`	The most recently generated version
`cachePromotions(`*n*`)`	The *n* most recently used versions
`cachePromotions(`*n*`,` *replacement policy*`)`	The *n* versions dictated by the *replacement policy*
`cachePromotions(all)`	**All versions generated**
`cacheSplits(1)`	The most recently generated version
`cacheSplits(`*n*`)`	The *n* most recently used versions
`cacheSplits(`*n*`,` *replacement policy*`)`	The *n* versions dictated by the *replacement policy*
`cacheSplits(all)`	**All versions generated**

FIGURE 14. Cache Policies

ated at dynamic-to-static promotions, but it depends on the size of the working set of values of the static variables of a given program.

At the same time, we expect that programmers will generally not want to bound the number of versions created at discordant merges through the cache-policy mechanism. Discordant merges commonly either yield a small, fixed number of specialized versions (for each version preceding the merge) or are the heads of loops to be unrolled, and the extent of loop unrolling can be bound by conditional specialization.

Therefore, we provide two separate sets of policies to restrict the caching of specialized versions in the two situations. The `cachePromotions(`*n*`)` policy bounds the number of cached versions created at dynamic-to-static promotions of the variables annotated at *n*, $n \geq 1$, versions, and `cacheSplits(`*n*`)` bounds the number of cached versions created at merges with respect to which the variables are discordant. If *n* is `all`, then the number of versions cached is unbounded, which is the default. Figure 17 on page 15 presents an example using cache policies.

These bounds are multiplicative for nested dynamic regions in that they specify the maximum number of versions that may be cached relative to one version of the enclosing dynamic region. Contrarily, when different bounds are specified for different static variables that are all discordant with respect to a particular merge, the maximum of their bounds is used as the bound on the number of versions created at that merge. It is not possible for multiple cache policies to be in effect for a particular dynamic-to-static promotion.

4.2.3 Replacement Policies

For values of n other than one or all, a replacement policy is needed to choose which version to replace when a new version must be stored in a full cache. The replacement policy may be specified as part of the cache policy (*e.g.*, `cachePromotions(`*n*`, fifo)`). Many policies are possible, such as `fifo` (first-in, first-out), `lfu` (least-frequently used), and `lru` (least-recently used). We intend to make `lru` the default policy and to provide the other two as options, but it is difficult to know which policies will be most useful until we apply them to real programs.

Another interesting possibility is to provide replacement policies that could prevent the creation of new specialized versions should the cache be full. In such a situation, a unspecialized version of the code would be executed instead. Such policies would rely on polyvariant division to produce divisions for both specialized and unspecialized versions. A simple example of such a policy would be to store the first *n* versions in the cache and forbid production of any new versions. A more sophisticated policy could keep frequency counts for all combinations of values of static variables, not just the ones corresponding to specialized versions currently in the cache, and not replace a version in the cache until the would-be new version is executed more times than the least-frequently executed version in the cache. This is a type of

1. We perform a reachability analysis to identify merges, called *static merges*, to which only one path is guaranteed to appear in any specialized version of the code. No cache lookups are required at such merges because only one version of the code that follows a static merge will be produced for each version that precedes it.

instrumentation-driven dynamic compilation, which we intend to explore more in the future.

Policy	Which Version Is Replaced
`fifo`	The oldest version in the cache
`lfu`	The least-frequently used version in the cache
`lru`	**The least-recently used version in the cache**

FIGURE 15. Cache Replacement Policies

4.3 Overcoming the Limits of Our Analysis

Our analysis of C programs is currently very limited in many ways. For example, we do not yet perform side-effect analysis or alias analysis, which are both necessary for a fully automatic run-time specializer. Nor are we able to cope with the presence of non-local jumps (*e.g.*, those from `longjmp()` to `setjmp()`), the occurrence of interrupts, or the existence of multiple threads of execution, which we consider to be operating-system-related complications and therefore issues distinct from the features of C that we must address.

To overcome the deficiencies of our analyses we have designed annotations and policies that enact optimizations which are unsafe. By unsafe, we mean our analysis cannot guarantee that the specialized program containing the optimizations is correct. That is, the behavior of the specialized program may not be identical, including termination and exception behavior, to that of the unspecialized program.

We rely on the programmer to assure that the specialization prescribed will produce a correct specialized program though our analyses cannot determine that. To alleviate this burden, we have made all of our default specialization policies safe, and to avoid unintentional use of unsafe annotations, we issue warnings whenever unsafe annotations are used. For example, it is almost never safe to make global variables static due to possible aliases and side effects.

4.3.1 Eliminating Cache Lookups

Run-time constants, variables initialized at run time which remain invariant thereafter, are difficult to identify automatically, requiring interprocedural alias analysis to ensure that the variables are only defined once. So, we provide another cache policy, `cachePromotions(unchecked)`, to indicate that only one specialized version should be produced for the dynamic region because the annotated variables are run-time constants. In this case, no cache lookup is performed and if the static variables do change, the specialized program will not produce the correct result.

We also provide a policy, `cacheSplits(replicate)`, to indicate that a new version should always be created at a discordant merge with no cache lookup. It makes sense to use this policy when the programmer is sure that no duplicate versions will be created, and when loops to be unrolled will reach their static bounds. For example, the `cacheSplits(replicate)` policy is typically used when unrolling loops that simply count up or down to their bounds monotonically. The policy is unsafe with respect to termination if the laziness policy is `eager` and the loop does not terminate based on a static bound, but instead relies on a dynamic exit condition.

Policy	Which Versions Are Cached
`cachePromotions(unchecked)`	The first version generated (no lookup)
`cacheSplits(replicate)`	All, possibly redundant, versions (no lookup)

FIGURE 16. Unsafe Cache Policies

4.3.2 Making Unsafe Operations Static

Only operations that are non-trapping, idempotent, and side-effect-free may be safely made static and executed by the specializer. If *unsafe* operations, such as division and pointer dereferencing, are made static, the behavior of the program may be altered. We provide annotations and policies to permit such operations to be made static. For example, `makeStatic` may be applied to division operations, which are unsafe because they may generate exceptions, to make them static.

To be able to make heap-allocated structures static, we provide the `transitive` dereferencing policy (as opposed to `intransitive`, the default) to make all dereferences of a static pointer variable static as well. This has the effect of making entire pointer data structures static. While transitivity is a highly useful policy, it is unsafe since not only does it make dereferences, which are potentially trapping operations, static, but it also makes the data pointed to static without including their values in cache lookups. The use of this feature is illustrated in the lookup routine from a cache simulator (see figure 17).

```
cacheResult
cacheLookup(void *address, CacheT *cache) {
makeStatic(cache : transitive,
                   cachePromotions(1));
  {
    unsigned blockSize = cache->blockSize;
    unsigned numLines = cache->numLines;
    unsigned tag = (unsigned) address /
        (blockSize * numLines);
    unsigned line = ((unsigned) address /
        blockSize) % numLines;
    setStructure **setArray =
        cache->lines[line]->sets;
    int associativity = cache->associativity;
    int set;

    makeStatic(set : cacheSplits(replicate));
    for (set = 0; set < associativity; ++set)
      if (setArray[set] makeDynamic -> tag ==
          tag)
        return CacheHit;
    return CacheMiss;
  }
}
```

FIGURE 17. Pointer Transitivity

We permit the `transitive` policy can be overridden at a particular dereference using `makeDynamic`, and `intransitive` by `makeStatic`. This capability allows us to deal with partially static structures. We are also considering the addition of annotations on fields of structure definitions, which may prove to be useful but are a markedly different style from our current annotations.

Finally, we provide a `constant` annotation, which indicates that a function is side-effect free and idempotent, and therefore should be evaluated statically when all arguments are static with the result memoized. Such a function is then included in our binding-time analysis in the set of functions that produce static values from static arguments. `constant` has the following form.

```
constant f(<parameter-list>)
```

4.3.3 Exploiting Laziness to Achieve Safety

Lazy specialization may be used to ensure safety with respect to exception behavior and termination. The `loopLazy` policy guarantees that the termination properties of the program are conserved because it causes loops (and recursive function calls) to be unrolled one iteration at a time, permitting dynamic loop bounds to halt unrolling. The `lazy` policy further guarantees that exception behavior will be identical to that of the unspecialized program by ensuring no static code will be executed by the specializer unless it would have been executed in the unspecialized program.

4.4 Conditional Changes of Specialization Policies

We described in section 3 how we use polyvariant division to implement conditional specialization. We have extended the meaning of polyvariant division to include propagation of multiple sets of policies within a dynamic region. In the same way that we compute multiple static/dynamic divisions of the control-flow graph intraprocedurally, we also propagate multiple sets of laziness, caching, and transitivity policies when different policies are specified on different paths. This provides a way to conditionally change specialization policies.

5 Implementation

Our current design of DyC's implementation includes only support for intraprocedural specialization because the Multiflow compiler, the foundation of our system, does not provide a framework for interprocedural analysis. However, we are confident that extending our framework to support interprocedural specialization should be straightforward due to the function annotations. In fact, the most challenging aspects of the implementation thus far have been related to making changes in the code generator to produce correct and efficient specialized dynamic compilers. Therefore, we restrict the discussion that follows to the design of the intraprocedural analysis and specializer.

The work performed by our static compiler can be roughly divided into two parts, the binding-time analysis and the construction of specialized dynamic compilers, also called generating extensions. Each part is outlined below.

5.1 Binding-Time Analysis and Policy Resolution

Our polyvariant binding-time analysis is implemented by an iterative data-flow analysis over the control-flow graph that computes the following information in parallel.

1. Policies are propagated downstream only and not up back edges (as indicated in section 2), which are identified by a depth-first search of the control-flow graph.
2. Static variables are derived from compile-time constants and the root static variables specified by the annotations. We also keep track of which static variables were derived from which other static variables, to be used in step 11.

3. When a static variable is assigned a dynamic value, we mark the dynamic-to-static promotion if its polyvariance policy is `promoteStatic`, and otherwise we make the variable dynamic. Currently, taking the address of a static variable also makes it dynamic because we do not yet perform alias analysis, even under the `promoteStatic` policy.
4. From the static-variable information, static and dynamic branches (*i.e.*, branches with static or dynamic predicates, respectively) are identified.
5. A reachability analysis, as in our first-generation system [APC+96], uses the static-branch information to determine which control-flow merges are static and which are dynamic.
6. Use-definition information and a live-variables analysis are used to identify static variables that are discordant with respect to dynamic merges (*i.e.*, live variables with multiple static definitions). Such variables are made dynamic if their polyvariance policy is `monovariant`, and otherwise their cache policies are used to assign a cache policy to the merge, which is marked discordant.
7. The earliest postdominating dynamic branch of each lazy program point, or the beginning of the dynamic region if one cannot be found, is marked lazy. Which program points are lazy depends on the laziness policy. All program points are lazy if the policy is `lazy`, all discordant dynamic merges are lazy if it is `polyLazy`, and only discordant loop-head merges are lazy under the `loopLazy` policy.
8. The context, which serves as a key for cache lookups, is computed from the set of root static values at each program point.
9. A live-variables analysis is used to determine when dynamic-to-static promotions take place at `makeStatic` annotations (*i.e.*, when they are live and were previously dynamic) and, hence, when variables must actually be added to the context, and also when static variables can be removed from the context prior to their explicit dynamic conversion (*i.e.*, when they are permanently dead).
10. Use-definition information is used to cluster and merge cache lookups. A cache lookup due to a dynamic-to-static promotion at a `makeStatic` annotation may be moved between uses and definitions of the variables promoted above and below the annotation without changing the extent of specialization.
11. Removal of variables from the context are similarly clustered and merged with cache lookups. When a root static variable is removed from the context, all of the static variables derived from it must be made dynamic to ensure all derived static variables are updated whenever the root static variables change.

Following the binding-time analysis, basic blocks with multiple static/dynamic divisions are replicated to establish precisely one division per program point.

We now have a functional binding-time analysis completed, corresponding to steps 1 through 8 above. We have not yet implemented the context and cache-lookup optimizations described in steps 9 to 11. We also do not yet perform the replication necessary to implement polyvariant division.

Our current implementation does not support the `polyvariant` policy, but only `promoteStatic`. The absence of the `polyvariant` policy will allow us to simultaneously propagate the set of `promoteStatic` variables and split merges with different incoming sets to implement polyvariant division as a pass before our monovariant binding-time analysis. This is possible because variables with the `promoteStatic` policy are not affected by dynamic assignments.

5.2 Building Specialized Dynamic Compilers

Based on the information computed by the binding-time analysis, we construct specialized dynamic compilers for each function containing dynamic code. Each specialized dynamic compiler determines when code must be respecialized or replicated, computes derived static values, performs optimizations, and generates code. The following steps are performed in constructing the specialized dynamic compilers.

1. The control-flow graph is partitioned into *specialization units*, which are single-entry subgraphs such that all basic blocks of a unit may be eagerly specialized on the same set of static variables. Thus, we must insert unit partitions on each edge from a dynamic decision marked lazy, wherever the context changes, at other dynamic-to-static promotions, at discordant merges requiring replication, and at merge with predecessors from different units.

   The unit partitioning greatly simplifies the implementation of the specialized dynamic compilers. The units simplify the cache structure and lookup procedures, in part, because only whole units are stored in the cache and cache lookups are required (only) at transitions between units. Additionally, units simplify certain run-time decisions, such as when to stop specializing, and reduce other run-time overheads, such as reducing use of the worklist used in step 7. Finally, units identify subgraphs within which we can safely globally optimize.

2. Units created due to dynamic-to-static promotions or lazy annotations on the control-flow graph are marked lazy, and all others are marked eager. This information is used in determining which units can be dynamically compiled at a given program point (see steps 7 and 8).
3. We then separate static operations, whose operands are all static, from dynamic ones, creating two copies of the control-flow graph of the procedure (both static and dynamic branches appear in both graphs).
4. Next, we generate optimized machine code, called *templates*, for the graph of dynamic operations, and maintain its intermediate representation. We conservatively prohibit movement of instructions across unit boundaries (note that some code movement could have already taken place before our analysis) and speculative hoisting above static decisions in order to ensure instructions are generated only when needed.
5. Subsequently, we convert the graph of static operations of each unit to a DAG such that each path through the DAG will generate a valid specialization. This process eliminates run-time overhead associated with traversing the nodes of the unit. We create the DAG by linearizing the nodes of the graph, but splitting at every static branch. We then merge common tails by sinking them into a single merge node and performing a series of local unifications of identical predecessors of merge nodes.
6. We then insert into the graph of static operations instructions to generate and optimize (e.*g*., with peephole optimizations) the template instructions after their static operands have been computed.
7. Where a template branch to an eager unit is generated, code is inserted to add that unit to a worklist and to store the values of the static variables needed by the unit. The worklist is used by the dynamic compiler to keep track of which units must be specialized and the values on which they must be specialized.
8. On each leaf of the DAG of a specialized dynamic compiler, if the worklist is not empty, another unit is removed from the worklist and its dynamic compiler is invoked. If the worklist is empty the dynamic compiler switches to the code just generated. In the case where a unit has only one eager successor, we can eliminate the worklist manipulation and jump to the successor directly.
9. If the cache policy is not `cacheSplits(replicate)`, a cache lookup must be performed by the dynamic compiler at the beginning of an eager unit. The cache lookup is essentially a hash on the values of the context variables into the code cache. When a cache lookup for an eager unit fails, the dynamic compiler allocates memory and continues to generate a new specialized version of the unit. In the case of a successful cache lookup, the branches into the unit are patched and another unit is removed from the eager-unit worklist.
10. Template branches to lazy units must be replaced with cache lookups when the cache policy is not `cachePromotions(unchecked)` or `cacheSplits(replicate)`, and otherwise with indirect jumps that initially point to the dynamic compiler for the target unit, and then is redirected to the specialized version of the unit by the dynamic compiler. After all branches to lazy units are so converted, code to patch intra-unit branches is inserted.

    When a cache lookup for a lazy unit by specialized code succeeds, the pointer into the table of static values for the unit is loaded and an indirect jump to the specialized unit is performed. A lookup failure induces a switch to the dynamic compiler.
11. The overhead of the cache lookups can be reduced by structuring the cache as a hierarchy of caches. A unit's position in the hierarchy is determined by its context. The consequence is that the cache lookup performed on the transition between two units need only hash on the values of the variables contained in the difference between their context sets, which we believe will typically be small. For each specialized unit, the hierarchical cache contains the code and a pointer into a flat table storing the specialized values. Values for all versions of all units may be stored in the same statically allocated table, for which we reserve a portion of the address space. Because we do not garbage-collect specialized units, the size of the cache and table can only grow.

We have designed nearly all of the pieces of this construction process, but only have the partitioning of the control-flow graph into units implemented. We are implementing the remaining steps.

6 Related work

6.1 Static partial-evaluation systems

As discussed in section 3, our system is modelled closely after off-line partial evaluators [JSS89], [Con93]. In section 3, we explained why we incorporated a host of sophisticated features from traditional static PE systems into our framework. Below, we summarize the ways in which our system diverges from static PE systems.

The *ability* to specialize at run time is the chief capability that a dynamic compilation system such as ours has, that a static PE system does not. This capability has the following implications:

- Our system can specialize on the values of variables that are not available until run time; the domain of applicability of our system is therefore a superset of that of static PE systems.
- Our system is not limited to one specialization run per program execution. The ability to interleave specialization with execution allows DC systems to specialize on variables whose values change as the program executes. Specifically, our system does not need manual binding-time improvements such as The Trick.
- Our system has the option of specializing code in a complete lazy manner; in particular, loops may be unrolled and specialized one iteration at a time, thus guaranteeing that our specializer terminates.

The *necessity* of specializing at run time constrains the design of our system to a specialization phase that is as fast as possible. This constraint has the following implications:

- Our system performs division off-line at static compile time; we abandon on-line PE [WCR+], as a model for our system.
- We are willing to use hints from the user to control the extent, code-caching behavior and the conservativeness of specialization, as discussed in section 5.4. This is a departure from the spirit of traditional PE, where annotations are limited to the *identity* of root static variables that are generally function arguments.
- Since wish to avoid the overhead of manipulating an intermediate representation at run time, our specializer is a source to machine code transformer, rather than source to source as in traditional on-line PE systems. Self-applicability is therefore not one of the goals of our system, though it is an important requirement in many traditional off-line PE systems.

Partial evaluations systems for C, have emphasized automatic specialization for "clean" subsets of C. Anderson [And92],[JGS93] for instance, proposes the aggressive use of call-graph analysis and pointer analysis to ensure that address-taking, side-effecting functions, computed jumps, and partially static data structures affect the accuracy of the BTA as little as possible. Since we are interested in specializing widely-used programs, which often contain "unclean" features such as the casting of void pointers disallowed in Anderson's system, we have decided to allow the user to advice the BTA through policy annotations, and not rely entirely on automatic analyses. On the other hand, in the case the user does not provide sufficient information, or as an aid to the user, the automatic analyses of the type proposed by Anderson will probably be a valuable addition to our system.

6.2 PE-based dynamic compilers

At least two recent systems, Fabius [LL96] and Tempo [CN96] have worked on extending partial-evaluation ideas to achieve dynamic compilation. Fabius is an off-line partial evaluation based run-time code generation system with limited forms of polyvariant specialization and division, for a strictly functional subset of the ML programming language. Tempo is a run-time specializer for C, also structured as an off-line partial evaluator, and also with limited support for polyvariance. In the next two paragraphs, we compare these two systems with ours along two dimensions: the extent to which they support polyvariance in specialization and division and thereby expose the full power of partial evaluation techniques, and the mechanisms they adopt to keep specialization overhead low.

Fabius provides support for polyvariant specialization, including support for re-specializing if the value of a variable changes, through memoization on the programmer-indicated static arguments of its functions. Every time a function is called, a cache check is performed on the values of the static arguments of the function; if no specialization exists for the function for these values, a copy of the function is specialized. However, because memoization occurs only on arguments of functions selected statically by the user, the form of polyvariance provided by Fabius places a heavy burden on the programmer. Consider the following program, for instance:

```
fun foo (s) (d) =
   if (s<=0) then s
   else
   let val v1 = if (d == 0) then (s -1)
                                else (s - 2)
pp1:   val  v2 = v1 >> 2
   in foo (v2) (d-1)
   end
```

The argument currying of arguments of foo indicates that s is to be specialized on, and is therefore a root static variable. d is dynamic.

FIGURE 18. Limited polyvariance in Fabius

The staging analysis (which performs static/dynamic division in Fabius) would term the use of v1 at program point *pp1* in the above piece of code dynamic, because no polyvariant specialization of the binding of the operation v1 >> 2 is possible. Note that in a static PE framework, memoization could take place at every function which took a static value as an argument. The >> function could then be memoized and the binding of v2 could be specialized polyvariantly. When translated to an imperative framework, the fact that polyvariance is supported in Fabius only at user-identified function calls is akin to saying that polyvariant specialization only occurs at program points where the user explicitly requires it. In the context of C, where we expect programs to be large, and points where polyvariant specialization possibly numerous and certainly difficult to identify for the user, we feel that user should at least have the option (the default in our system) of comprehensive polyvariant specialization.

Fabius does not support polyvariant division. In a functional language, polyvariant division manifests itself as the ability to specialize a given function on different subsets of its arguments. In Fabius, a user is allowed to pick one set of root static variables from among the arguments of a function; specialization on the function then occurs only on this one subset of the function arguments.

To the best of our knowledge, the Tempo system provides limited support for polyvariant division and specialization, but no support for dynamic re-specialization.

The primary means by which Tempo and Fabius seek to keep run-time specialization overhead low seems to be by carefully engineering their systems so that the generating extension is as fast as possible. At six instructions per instruction generated at run time, Fabius has demonstrated the capability to create impressively efficient generating extensions for a diverse set of small programs. However, if the regions of the program being specialized are too large, even a low-overhead specializer may incur a large total overhead. We expect this effect to be especially pronounced in our domain of interest, mainstream C programs, where functions are generally large, and profitable specialization often restricted to parts of the function. Further, in such programs, we expect higher-overhead capabilities such as polyvariant specialization and dynamic re-specialization to be of moderately frequent use, but crucial to performance (they aid in multi-way unrolling of loops, for instance). In these programs, therefore, we expect that some mechanism like our annotations, which guide the scope and aggressiveness of specialization will be critical for performance.

6.3 Imperative-style dynamic compilers

Declarative dynamic compilation systems such as ours work on the principle that any program to be generated at run time can be viewed as the result of specializing a program on some subset of its arguments. The values of these arguments may be available only at run time, and may change during the program run. Once the subset of arguments is identified (generally by user annotations), it is the responsibility of the system to deduce derived variables on which specialization should be performed, produce a *generating extension* which will produce the specialized program given values of static variables at run time, and provide for re-specialization in case these values change.

In imperative systems, such as the ‘C system of Engler and colleagues [EHK96], on the other hand, the user hand-writes the generating extension, which manipulates fragments of code identified manually by the user to produce the specialized version. There is no explicit notion that the program being produced at run time is a specialization of the program, being written by the programmer. Specifically, we cannot run the dynamically generated program on a subset of the inputs of the statically written one, and get the same results as the latter does. The generating extension dynamically generates a program, which in general performs an independent computation on a set of arguments possibly disjoint from those of the program that generated it.

Since both declarative and imperative DC systems are, in general, Turing complete languages, they can simulate each other. In the kinds of program transformations they can express, therefore, they are identical. We argue below, however, that the idea of viewing programs produced at run time as specializations of programs written at compile time results in concrete benefits for a programmer, including ease of optimizing existing programs, debugging, and automation of various aspects of dynamic compilation.

Beyond comparing specific examples[1], it is difficult to argue in any systematic manner that the “program specialization” view is more natural than the “program construction” view for a programmer writing a *new* program to generate code at run time. On the other hand, the former approach is clearly easier to use when we consider applying dynamic compilation to **existing applications**: in general declarative systems will require sparse annotations on existing source, whereas imperative systems will require re-writing the source.

Writing in the declarative manner allows the programmer to “turn off” run-time code generation, by disabling all annotations. Since the unspecialized version of the program has the same semantics as the specialized one, the programmer has the option of **debugging** the program with specialization turned off, and enable the “specialization switch” only during performance tuning.

Note that some DyC policies are designated “unsafe” (see section 4.3) because program semantics are *not* guaranteed to be preserved if they are used incorrectly. Debugging in the presence of unsafe annotations can therefore be complicated. By using unsafe annotations, the programmer pays the price of losing the safety guarantee (and related debugging costs), but only when he/she desires the extra performance.

1. Nonetheless, we were pleasantly surprised that we were able to express the sample ‘C programs from [EHK96] in a simple and natural manner using our annotations.

Once the user identifies variables which may be beneficial to specialize on, the declarative framework **automates** the following aspects of dynamic compilation:

1. Identifying derived variables to specialize on, starting from a (hopefully small) root-set specified by the user,
2. Creating the generating extension, and identifying the fragments of the original program that it will act on. Note that identifying the fragments and designing the generating extension is a major responsibility of the programmer in imperative systems,
3. Inserting code to trigger re-specialization if necessary, and creating and managing a cache of specialized code versions at this point, and
4. Enabling conditional specialization by creating generating extensions for each different policy under which a region may be specialized.

In ‘C, the user is forced to implement this entire mechanism manually for each program. Given the complexity of identifying the BTA required step 1, for instance, one would imagine that much of this functionality is provided by library functions. In fact, Engler et al [EHK96] do propose a partial evaluation layer on top of their framework, which we expect will considerably ease the burden on the user of their system. However, we expect that their notion of the layer “syntactic sugar” on their declarative system will probably prove to be an underestimate.

7 Conclusion

We have designed and are currently implementing a dynamic compilation system, DyC, that applies many ideas from partial evaluation to the domain of run-time code generation. Several aspects of our system contribute to its ability to specialize realistic C programs. We use simple but expressive annotations to express what parts of the program to specialize on which variables, without requiring rewriting of the program. We use policies to control the level and cost of specialization and provide control over the number of specialized code versions.

DyC offers a number of novel features that overcome restrictions of previous dynamic compilation systems. Both polyvariant division and specialization are available to the programmer. The annotations provide sub-function level granularity of specialization, important for imperative languages, like C. Additionally, they offer control over specialization to limit its run-time cost and allow the management of code space.

We plan to finish our implementation in the short term and gain experience in the practical implications of our annotations and policies for real world programs. We also plan to do performance studies to assess the performance implications of our annotations. Finally, we are working towards a more automated system, where profiling data and automated tools would initially assist the programmer annotating the program as a first step towards a more automated dynamic compilation framework.

References

[APC+96] J. Auslander, M. Philipose, C. Chambers, S.J. Eggers and B.N. Bershad. Fast, Effective Dynamic Compilation. In *Conference on Programming Language Design and Implementation,* February 1996.

[And92] L.O. Andersen. Self-applicable C program specialization. In *Partial Evaluation and Semantics-Based Program Manipulation*, June 1992.

[And93] L.O. Andersen. Binding-time analysis and the taming of C pointers. In *Partial Evaluation and Semantics-Based Program Manipulation*, June 1993.

[Bon92] A. Bondorf. Improving binding times without explicit CPS-conversion. In *1992 ACM Conference in Lisp and Functional Programming*, June 1992.

[BSP+95] B.N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Symposium on Operating Systems Principles*, November 1995.

[CEA+96] C. Chambers, S.J. Eggers, J. Auslander, M. Philipose, M. Mock, and P. Pardyak. Automatic dynamic compilation support for event dispatching in extensible systems. In *Workshop on Compiler Support for Systems Software*, February 1996.

[Con93] C. Consel. A tour of Schism: a partial evaluation system for higher-order applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation*, June 1993.

[CD89] C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. In *Information Processing Letters*, January 1989.

[CD91] C. Consel and O. Danvy. For a better support of static data flow. In *Functional Programming Languages and Computer Architecture 5th ACM Conference Proceedings*, August 1991.

[CD93] C. Consel and O. Danvy. Partial evaluation: principles & perspectives. In *Symposium on Principles of Programming Languages*, January 1993.

[CN96] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *IEEE International Conference on Computer Languages*, January 1996.

[CHK92] K.D. Cooper, M.W. Hall, and K. Kennedy. Procedure Cloning. In *Symposium on Principles of Programming Languages*, April 1992.

[Dan94] O. Danvy. Partial evaluation for program speedups. In *IFIP Transactions A (Computer Science and Technology)*, A-51, 1994.

[DL92] O. Danvy and J.L. Lawall. Back to direct style. II. First-class continuations. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, June 1992.

[EHK96] D.R. Engler, W.C. Hsieh, and M.F. Kaashoek. ‘C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Symposium on Principles of Programming Languages*, January 1996.

[EP94] D.R. Engler and T.A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[FRN84] J.A. Fisher, J.C. Ruttenberg, and A. Nicolau. Parallel processing: A smart compiler and a dumb machine. In *Symposium on Compiler Construction*, 1984.

[GJ91] C.K. Gomard and N.D. Jones. Compiler generation by partial evaluation: a case study. In *Structured Programming*, 12, 1991.

[GKR95] B. Guenter, T.B. Knoblock, and E. Ruf. Specializing shaders. In *SIGGRAPH '95*, 1995.

[JGS93] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[JSS89] N. Jones, P. Sestoft, and H. Sondergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. In *Lisp & Symbolic Computing*, 2, 1989.

[KEH93] D. Keppel, S.J. Eggers, and R.R. Henry. Evaluating runtime-compiled, value-specific optimizations. Technical Report 93-11-02, University of Washington, Department of Computer Science & Engineering, 1993.

[Kep96] D. Keppel. Runtime code generation. Technical report, University of Washington, Department of Computer Science & Engineering, 1996.

[KR96] T.B. Knoblock and E. Ruf. Data specialization. In *Conference on Programming Language Design and Implementation*, May 1996.

[[LFK+93] P.G. Lowney, S.M. Freudenberger, T.J. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. O'Donnell, and J.C. Ruttenberg. The Multiflow trace scheduling compiler. *Journal of Supercomputing*, 7, 1993.

[LL96] M. Leone and P. Lee. Optimizing ML with run-time code generation. In *Conference on Programming Language Design and Implementation*, May 1996.

[PEK96] M. Poletto, D.R. Engler, and M.F. Kaashoek. tcc: a template-based compiler for 'C. In *Workshop on Compiler Support for Systems Software*, February 1996.

[PMI88] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, (1), winter 1988.

[Ses88] P. Sestoft. Automatic call unfolding in a partial evaluator. In *Partial Evaluation and Mixed Computation*. North-Holland, 1988.

[SZ88] P. Sestoft and A.V. Zamulin. *Annotated Bibliography on Partial Evaluation and Mixed Computation*. North-Holland, 1988.

[WCR+91] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In *Functional Programming Languages and Computer Architecture*, August 1991.