Design and Implementation of Annotations for Dynamic Compilation
Quals Report

Markus U. Mock

February 10, 1997

Abstract

Traditional compilers produce all executable code at compile time. Dynamic compilation produces some executable code at run time, tapping additional opportunities for optimization by doing value-specific optimization on values that only become known at run time. There are different approaches to dynamic compilation, from completely manual, imperative approaches to fully automatic dynamic compilation. Annotation-based approaches are an intermediate point between these two extremes. They are easy to use for the programmer and can still provide many opportunities for optimization if they are well-designed. We present the design of simple, expressive annotations that allow us to express optimizations that were found to be useful in a previous system. They are based on two basic primitives *makeStatic* and *makeDynamic* that seem to be sufficient to dynamically compile a wide range of applications ranging from numerical codes to interpreter programs.

1 Introduction

While traditional compilers produce all executable at compile time, dynamic compilers defer compilation of some code until run time, possibly interleaved with code execution. Using dynamic compilation, value specific optimizations such as constant propagation, constant folding, strength reduction and full loop unrolling can be done with values that become known only at runtime, thus enabling further optimization of programs compared to traditional compilers.

Important applications that are likely to benefit from these additional opportunities for optimizations include interpreters, graphics rendering machines and numerical applications. Common to these applications is that some of the data structures they manipulate are slowly changing: the program being interpreted, the scene being rendered and constant matrices or vectors, respectively. Using these data structures that are, at least for some time, constant, a dynamic compiler can apply value-specific optimizations to produce a program specialized for these values. The specialized codes will have to do less computations, and repeated runs will amortize the cost of dynamic compilation.

In many cases, systems level code can also benefit from dynamic compilation. For example, some operating systems let an application specify a packet filter, which is a little program that is run to filter incoming network packets and pass only those the application is interested in to the application. The packet filter is usually constant for some time, which makes dynamic compilation for the particular filter possible to speed up the packet selection. Yet another benefit of dynamic compilation is that it can be an enabling technology for very high-level language features, such as dynamic dispatching of message sends in object-oriented languages [CUL89].

A key issue for dynamic compilation is how to identify where and when to compile code dynamically. There are different approaches to this problem. In the declarative, annotation-based approach, the programmer uses annotations specifying what to compile at runtime, but does not need to write the code that generates the dynamically produced code. In the imperative approach the programmer writes additional code that produces the dynamic code at run time. In a completely automatic approach — as yet unrealized — the compiler and runtime system automatically identify what parts of the code should be dynamically compiled and the dynamic code is produced without further programmer intervention.

The advantages of the annotation-based approach are that annotations are simple and require minimal programmer effort. Because of their declarative nature, existing code does not have to be rewritten. Moreover, the dynamic compiler can choose to ignore them, thus facilitating debugging.

Imperative approaches require substantial effort by the programmer and are therefore more tedious. Debugging of code specifically written for dynamic compilation becomes hard as there is no simple way to fall back to the unmodified code. On the other hand, imperative approaches give full control over the code being produced at run time so that any conceivable optimization can be expressed.

The big advantage of the automatic approach is that no programmer intervention is required. At the same time this is a problem because identifying what to compile dynamically and for what values is still an open problem.

An annotation-based approach can try to achieve a reasonable compromise between complete automation and manual, imperative approaches. To be able to do that, annotations should be expressive. They must be able to express which parts of the program to compile dynamically and for which variables and functions value-specific optimizations should be done. Additionally, they must be able to control both time and space costs incurred by dynamic compilation and be implementable with low run-time cost to make dynamic compilation profitable in many cases. Also, optimization of dynamically produced code should be possible in order to produce quality code.

2 Related Work

As the different approaches to dynamic compilation have different advantages and disadvantages, we would like to keep the good things and overcome the shortcomings of previous approaches. Hence, before describing our annotations

```
if (y <= 42)
   x = y;

M: x = x + 17;
```

Figure 1: Polyvariant vs. Monovariant Division.

we will look at the features of previous dynamic compilation systems and other related work (namely partial evaluation).

2.1 Partial Evaluation

Partial Evaluation (PE) does program optimization by specializing a program for specific values. It shares the same goal as dynamic compilation: speeding up programs by exploiting that certain values are known at specialization time. The values known at specialization time are called *static* values, while values that become known only later are called *dynamic* values. Computations based only on static values are done once at specialization time, reducing the amount of computation at run time. The code doing the computations based on dynamic values is called the *residual program*, which is executed at run time.

To identify which values are static and which are dynamic, PE does an analysis of the program, called *Binding Time Analysis* (BTA). It computes a division of the program into static and dynamic values. Usually, the programmer identifies certain root static values from which other static values are computed by the BTA. *Off-line* PE does the BTA before specialization time. *On-line* PE computes the program division at specialization time when the concrete values of static variables have already become known, which sometimes allows it to produce better code.

PE has developed different degrees of program specialization [JGS93]. *Monovariant specialization* produces only one specialization for a given static program variable. *Polyvariant specialization*, on the other hand, allows for multiple versions of specialized code for a given static program variable. For each different value of the variable that is specialized, polyvariant specialization produces a separate version of code. *Polyvariant division* permits multiple divisions of the program allowing for different kinds of specializations for the same program variable. *Monovariant division*, in contrast, computes only one division of the program. Consider the example, given by Jones et al. [JGS93], and shown, in a slightly modified way, in figure 1:

If, initially, `x` is static and `y` dynamic, a monovariant division of the program will compute that both `x` and `y` are dynamic at point `M`, because at this point `x` could depend either on a dynamic computation (`x := y`) or a static one (`x`). A polyvariant division computes a set of divisions for each program point. In the example, two division are computed: $\{(D, D), (S, D)\}$, denoting that either both variables are dynamic or that `x` is static and `y` dynamic.

Much of the partial evaluation work has been carried out in the context of functional languages. For that reason, PE typically does program specialization and division at function granularity, i.e., specialized versions of a function are produced for specific argument values. [JGS93]

2.2 Our First Generation System

Our previous prototype system is an annotation-based system. Code inside a dynamic region, delimited by the *dynamicRegion*() annotation, is compiled dynamically. At the beginning of the region the programmer identifies certain values as static (root values) from which further values are derived as static based on an analysis phase that is carried out at compile time. The code in the region is monovariantly specialized unless the annotation *dynamicRegion key*() is used, in which case polyvariant specialization is done for the region. The programmer has to give an explicit key that is used to discriminate between the different specialized versions. A special *unrolled* annotation is used to achieve complete loop unrolling.

A BTA, carried out at compile time derives more static values from the root static values given in the *dynamicRegion* annotation and from compile-time constants. All non-trapping, side-effect free, idempotent operators produce static values if all arguments are static. The BTA is carried out on the control flow graph (CFG) of the program, which allows our first generation system to deal with arbitrary, unstructured control flow. Unstructured control flow is quite common in C code where `break` and `continue` statements allow the early exit from structured control flow constructs and where `goto` statements can be freely used.

The first generation system uses a *reachability analysis* to identify static merges. Static merges are control flow merges that are guaranteed to be entered always through the same path. This allows the use of a less conservative merge rule when computing derived static values. The reachability analysis works by keeping track of static branch outcomes. At the start of the region the reachability condition is `true`. At each static branch, each branch outcome is added (using logical and) to the reachability condition for that program point. For example, if the branch condition is `x > 0`, the condition $x > 0 \rightarrow T$ is added on one side and $x > 0 \rightarrow F$ on the other, where T and F stand for true and false, respectively. At merge points reachability conditions are or-ed together. If at a merge the incoming reachability conditions are mutually exclusive, the merge is static, otherwise dynamic. If a merge is static, a static variable that has different definitions on the incoming paths can still be considered static after the merge because only one path can ever be taken. If the merge is dynamic, we conservatively have to assume that the variable is no longer static after the merge.

There is a mutual dependency between the analysis of static/dynamic values and reachability conditions, because only static conditions are used in the computation of the reachability conditions which in turn influence what values are identified as static. Therefore, the BTA is run interleaved with the reachability conditions analysis.

In the first generation system, dynamic code is produced from two components. *Setup code* computes the values of static variables that are used for specialization at setup (= specialization) time. *Template code* contains the code for computations that are dependent on dynamic values and also holes for the

```
/* assume pc, bytecodes and vsp are static, valueStack dynamic */
switch(bytecodes[pc] {
   ...
   case EQ:
      if (valueStack[vsp] == valueStack[vsp-1])
         pc = bytecodes[pc+1];
      else
         pc = pc + 2;
    M:
    break;
    ...
}
```

Figure 2: Multi-way Unrolling.

static values computed during setup time. At setup time the holes are filled (*stitched*) with the actual values and then the stitched template code is run.

The ability to deal with arbitrary control flow is one of the main advantages of the first generation system. The reachability analysis computes a program division with more static values than a BTA without it, potentially reducing the number of dynamic computations. Template code can be optimized at compile time producing better dynamic code at specialization time. Because stitching of holes is simple, run-time costs of dynamic compilation can be low.

On the downside, the annotations used in the first generation system are somewhat ad hoc. Loop unrolling needs a special rule and nesting of regions is impossible requiring that all static values be introduced at region entry. There is only limited control over caching at the beginning of region by the *key*() mechanism. Polyvariance is restricted to the beginning of the region; there is no polyvariance at dynamic merges and also no polyvariant division. This is significant because an important application of polyvariance at dynamic merges is multi-way loop unrolling, i.e., unrolling of a loop with more than one path through the loop. This made it impossible, for instance, to optimize a simple dynamic dispatch interpreter [CEA$^+$96].

The code fragment in figure 2 shows the dynamic merge M that caused the problem. Depending on the dynamic condition `valueStack[vsp] == valueStack[vsp-1]` the `pc` will be assigned either `bytecodes[pc+1]` or `pc + 2`, both of which are static.[1] Merge M is a dynamic merge, because it is not guaranteed be entered always from the same branch of the `if`. Therefore, the merge rule makes `pc` dynamic at M. However, to unroll the interpreter loop for the different possible values of `pc` it is necessary that `pc` be static. Because of the dynamic condition different paths through the interpreter are possible, and the unrolled loop will not be straight-line code, but rather a directed graph. The first generation system is unable to handle this case. Consequently, it cannot dynamically compile any interpreter where, for a specific interpreter program, different paths through the interpretation loop are possible, e.g., the dynamic dispatcher [CEA$^+$96] or

[1] The *static* annotations has to be used for loads through static pointers (here `bytecodes`) because these are not assumed to be transitive, i.e., static pointers are not assumed to point to static data. This is a safe default. In our first generation system [APC$^+$96] static pointers by default pointed to static data which in many cases lead to surprises for the programmer.

the bytecode interpreter discussed later in section 3.1.

2.3 Imperative style systems such as 'C

In annotation-based, declarative systems it is the responsibility of the system to create a *generating extension* which will produce specialized code given the values of static variables at run time and provide for respecialization in case these values change. In imperative systems, such as the 'C system [EHK96], the user has to write the generating extension himself. 'C provides so-called tick-expressions that are used to construct dynamic code.

While this approach is versatile and allows the programmer to generate arbitrary code and apply any optimizations to it, it is also error-prone and tedious. Particularly, debugging becomes hard because is is impossible to debug the code with dynamic compilation turned off.

2.4 Fabius

Fabius [LL96] is a dynamic compilation system based on off-line partial evaluation of a functional subset of the ML programming language. Fabius provides support for polyvariant specialization, including specialization if the value of a variable changes (respecialization).

Fabius uses a binding time analysis to compute the static/dynamic division of the program. The analysis is relatively straightforward because referential transparency and single-assignment semantics, properties of the functional language it uses, simplify its task. In particular, no alias or side-effect analysis is required.

Polyvariance in Fabius is limited to function parameters and does not extend to assignments inside functions that are controlled by dynamic conditions. Therefore, the programmer has to restructure the program in such a way that the points that require polyvariant specialization coincide with function boundaries. Another shortcoming of the Fabius system is that it does not allow control over the amount of cached code, apart from a maximum code space limit for all dynamically generated code. Hence, it is impossible to restrict space usage for a particular function, a useful capability given that the specialization of some functions is more beneficial than that of others. Polyvariant division is also impossible in Fabius because a given function can only be specialized on one subset of its arguments.

2.5 Tempo

Tempo [CN96] is a run-time specializer for C, structured as an off-line partial evaluator. It does its analysis phases on the abstract syntax tree (AST) produced from the C source code. After an alias and side-effect analysis a binding time analysis is performed that computes the static/dynamic division based on programmer annotations. Then, based on the results of this phase, an *action analysis* is performed. For each program construct it computes which program transformation (called *specialization action*) should be done. An abstract interpreter produces a tree grammar from the action annotated program. The tree grammar is used to generate templates, like our first generation system. However, these templates are in the source language, C.

Currently, Tempo does not seem to provide support for dynamic specialization of a variable whose value changes. It also does not provide polyvariant division. Because it does its analysis on ASTs rather than CFGs, it appears that the computed static/dynamic division must be more conservative than the one computed by our first generation system which operates on CFGs [APC+96]. Like Fabius, Tempo doesn't support any control mechanisms for the caching of dynamically generated code.

3 Annotations

We now present the design of our annotations. First, we'll state what they can do. Then, by means of a few examples, we'll give an intuition of their semantics followed by a description of their exact syntax. Finally, in section 4, we will discuss their semantics in terms of the required analyses and their output.

We use two basic annotations to control dynamic compilation: *makeStatic* and *makeDynamic*. Code inside a dynamic region (delimited by *makeStatic* and *makeDynamic*) is compiled dynamically. A dynamic region does not span across functions, i.e., *makeStatic* and *makeDynamic* are intraprocedural annotations. The annotation *specialize* is provided to specialize functions as a whole. Specialized versions are called at those call sites where the static actuals match the variables designated for specialization in the annotation. A *constant* annotation is used to give the BTA additional information about function properties. Specialization *policies* are provided to control the degree of specialization, permitting monovariant specialization, polyvariant specialization and polyvariant division.

3.1 Intuitive Meaning

3.1.1 *makeStatic* and *makeDynamic*

The two basic primitives are *makeStatic*() and *makeDynamic*(). *makeStatic* takes as arguments the variable(s) that are to be specialized and a list of specialization policies. A listed variable is used for specialization until *makeDynamic* removes it from the set of variables for which code is specialized (= *specialization variables*). For the the exact syntax the reader is referred to the syntax section 3.2.

makeStatic($x : p$) adds variable x to the variables used for specialization. p is the specialization policy that is to be used for this variable. From this point on until removed by a *makeDynamic* or implicitly at the end of the current function, code will be specialized based on values of x guided by specialization policy p. p could be, for instance, *monovariant*, requiring only monovariant specialization for the variable. Hence, *makeStatic* introduces a static value whereas *makeDynamic* removes it. A list of available policies is shown in tables 1, 3, and 2. The default policy is always indicated in bold. Note that in table 1 "multiple specialization of a region" describes the behavior of the

Policy	Multiple Specializations of region?	Multiple Specializations at merges?	Polyvariant division?	Automatic dynamic-to-static promotion?
monovariant	yes	no	no	no
polyvariant	yes	yes	yes	no
promoteStatic	yes	yes	yes	yes

Table 1: Specialization Policies.

policy at a *makeStatic* annotation that begins a new dynamic region.

Consider the following example, a simple bytecode interpreter which is shown in figure 3.
It computes the series of the famous $3a+1$ problem.[2] Figure 4 shows the C code computing the series and figure 5 shows the corresponding bytecode program for which the interpreter will be specialized. The code shown in figure 6 would be produced by dynamically compiling the interpreter with the shown bytecode program as argument. We show C code, although machine code will be produced in reality.

The interpretation overhead and unused operations in the interpreter get removed in the dynamically produced code resulting in code that is fairly close to standard compiled code for the original C program.

3.1.2 Specialization Policies

The bytecode interpreter example shows the use of the policy *polyvariant*. At the end of the `switch` there's a dynamic merge for variable `pc` which would normally render `pc` dynamic after the merge because there are different definitions of `pc` reaching the merge (we call such merges *discordant*). The policy *polyvariant* will keep `pc` static after the merge resulting in an unrolling of the interpreter loop. This works by code duplication after the merge. For each distinct value that `pc` will take on at the merge, the subsequent code will be specialized separately. Therefore, as `pc` goes through its different values, different loop bodies are produced specialized on the separate `pc` values. This effectively unrolls the loop. In the unrolled loop, then, `pc` is a run-time constant value, which is why it is not explicitly present in the unrolled code.

An even more aggressive form of specialization is provided by the policy *promoteStatic*. It provides polyvariant specialization with additional dynamic-to-static promotion. The difference compared to *polyvariant* is shown in the example in figure 7.
`d` is a dynamic condition, therefore it is possible that `x` have either value $x+1$ or $x+2$ after `M1`, i.e., the code after `M1` needs to be specialized polyvariantly. Both *polyvariant* and *promoteStatic* do this, so at `M1` `x` will still be static. At `M2`, however, `x` is assigned a dynamic value (program input). If we want to keep

[2]It is not known if: $\forall a \in\mathcal{N}\exists k \in\mathcal{N}$: $f^k(a)=1$, where

$$f(a) = \begin{cases} a/2 & \text{if } a \text{ is even} \\ 3a+1 & \text{if } a \text{ is odd} \end{cases}$$

```
int interpret(byteCodeT byteCodes[], int numArgs, int arglist[]) {

    int valueStack[STACKSIZE];
    int pc = 0, vsp=0;
    int arg, opcode;

    /* push arguments */
    for (arg=numArgs; arg ¿ 0; arg–)
        valueStack[vsp++] = argList[arg-1];
    vsp–; /* adjust to top element */
    makeStatic(byteCodes:polyvariant);
    makeStatic(pc:polyvariant); /* polyvariant specialization on byteCodes and pc */

    for (;;) {
        opcode = static byteCodes[pc];
        switch (opcode) {
            case ADD:
                valueStack[vsp-1] = valueStack[vsp] + valueStack[vsp-1];
                vsp–; pc++;
                break;
            case SUB:
                valueStack[vsp-1] = valueStack[vsp-1] - valueStack[vsp];
                vsp–; pc++;
                break;
            /* analogous cases for other arithmetic operators MUL. DIV */
            case DUP:
                valueStack[vsp+1] = valueStack[vsp];
                pc++;vsp++;
                break;
            case PUSH:
                valueStack[vsp++] = static byteCodes[pc+1];
                pc += 2;
                break;
            case POP:
                vsp–;
                pc++;
                break;
            case ODD:
                if (valueStack[vsp–] %2 == 1)
                    pc = static byteCodes[pc+1];
                else
                    pc +=2;
                break;
            case NEQ:
                if (valueStack[vsp–] != 0)
                    pc = static byteCodes[pc+1];
                else
                    pc +=2;
                break;
            case GOTO:
                pc = static byteCodes[pc+1];
                break;
            case WRITE:
                printf("%d\n",valueStack[vsp]);
                pc++;
                break;
            case RET:
                return valueStack[vsp];
        }
    }
}
```

Figure 3: Bytecode Interpreter.

```
/* print out series:
   a(0), a(1), a(2) ...
   where a(n+1) = 3 * a(n) +1; if a odd
                  a(n) / 2;    if a even
   until a(n) == 1 (if ever)

*/
int f(int a) {
   printf("%d\n",a);
   for (;;) {
      if (a == 1)
         return a;
      if (a %2 == 1)
         a = 3 * a +1;
      else
         a = a /2;
    }
}
```

Figure 4: 3a+1 C Program.

```
L0: WRITE
DUP
PUSH 1
SUB
NEQ L1
L1: DUP
ODD L2
PUSH 2
DIV
GOTO L0
L2: PUSH 3
MUL
PUSH 1
ADD
GOTO L0
```

Figure 5: 3a+1 Bytecode Program.

```
L0:
printf("%d\n",valStack[0]);
valStack[1] = valstack[0];
valStack[2] = 1;
valStack[1] = valstack[1] - 1;
if (valStack[1]  != 0) goto L1;
return valStack[0];
L1:
valStack[1] = valStack[0];
if (valStack[1] %2 == 1) goto L2;
valStack[1] = 2;
valStack[0] = valStack[0] / valStack[1];
goto L0;
L2:
valStack[1] = 3;
valStack[0] = valStack[1] * valStack[0];
valStack[1] = 1;
valStack[0] = valStack[1] + valStack[0];
goto L0;
```

Figure 6: Dynamically Produced Code For the 3a+1 Bytecode Program.

```
makeStatic(x:promoteStatic);
... /* uses of x */
if (d)
   x = x + 1 ;
else
   x = x + 2;
M1:
... /* uses of x */
M2: x = read(); /* read from input */
y = x + 42; /* x still static */
...
```

Figure 7: Dynamic-to-Static Promotion. `d` is a Dynamic Condition.

```
makeStatic(x:monovariant);
makeStatic(y:polyvariant);
makeStatic(z:promoteStatic);

/* x, y, z static */

if (s)               /* s static condition */
   x = y = z = 1;
else
   x = y = z = 2;

P1: /* x, y, z static */

if (d)              /* d a dynamic condition */
   x = y + z;
else
   /* only uses of x, y z */

P2: /* dynamic merge, x no longer static
       y and z still static
       continue specializing on y and z */

if (d)
   y = z = 3;
else
   y = z = 4;

P3: /* discordant merge
       y and z still static */

y = z = read();

P4: /* only z still static */
```

Figure 8: Effect of Different Policies.

`x` static after `M2` we have to delay specialization of code after `M2` until the read is executed at run time. *promoteStatic* will do this, whereas *polyvariant* will not.

A third specialization policy, *monovariant*, is also provided. In contrast to *promoteStatic* and *polyvariant*, it will not continue specialization of variables past any dynamic merge with multiple reaching definitions for the specialization variable, i.e., specialization for monovariantly specialized variables will be turned off at such merges. In the example in figure 8 the differences between the policies are shown.

Polyvariant Division Sometimes it is useful to specialize code when a certain condition is true at run time. Polyvariant division is therefore included in the *polyvariant* and *promoteStatic* policies. The example in figure 9 shows how this feature might be used. Code after `M` is specialized on variable `x` only if $l < x < u$. Our system will produce two versions for the code after `M` at compile time, one where `x` is used for specialization and one where it is not. At run time one or the other will be selected depending on the outcome of the test.

```
if ((x < u) && (x > l))
   makeStatic(x:polyvariant);
M:
```

Figure 9: Polyvariant Division.

Policy	Which Versions Are Cached
cachePromotionOne	most recently generated version
cachePromotionAll	all versions generated
cachePromotionUnchecked	first generated version

Policy	Which Versions Are Cached
cacheSplitOne	most recently generated version
cacheSplitAll	all versions generated
cacheSplitReplicate	new version each time

Table 2: Cache Policies.

3.1.3 Cache Policies

In the bytecode interpreter example 3 `pc` is polyvariantly specialized, producing separate versions of the loop body for different values of `pc`. When we encounter the same `pc` value again (for the same interpreted program) we would like to reuse the previously specialized code. That way, we pay the overhead of specializing for a particular value only once. For that reason, we cache specialized code. We use cache policies to provide control over the amount of code that is being cached.

Caching happens at two places. First, caching is done at dynamic-to-static promotions, i.e., at a *makeStatic* annotation that promotes a dynamic value to a static value or at a dynamic-to-static promotion for a variable that is specialized with the *promoteStatic* policy. At these points we cache the code specialized for the particular value of the specialization variable. Second, we also need to cache at discordant merges where a specialization variable can take on different values depending on the path through which the merge is entered. The policies *cachePromotionAll*, *cachePromotionOne* and *cachePromotionUnchecked* are used to control the amount of caching at the dynamic-to-static promotion points, whereas *cacheSplitAll*, *cacheSplitOne* and *cacheSplitReplicate* control caching at discordant merges.

With *cachePromotionAll* specialized code for all different values of the specialized variable are generated and cached, i.e. the cache is allowed to grow as big as necessary. *cachePromotionOne*, on the other hand, will keep only the latest specialized version of code in the cache, replacing the previous one, if necessary. In both cases the cache is checked to see if the specialized code for the particular value is already in the code cache. *cachePromotionUnchecked* is like *cachePromotionOne* except that it will always keep the version of code that was produced first. It is provided to allow the programmer to make a performance/safety trade-off because checking the cache incurs some run-time overhead. Using *cachePromotionUnchecked* the programmer can turn the

```
makeStatic(i:cacheSplitReplicate);
sum = 0;
for (i=0; i<C,i++){
  sum += a[i];
}
```

Figure 10: Unrolling Static Loops, C is a Run-time Constant.

cache check off, i.e., the first specialized version that was put in the cache will be used without a check thereafter. In our bytecode interpreter example we could use `makeStatic(bytecodes:cachePromotionUnchecked)` if we know that we will call the interpreter only with one program in a particular run of the interpreter. The programmer implicitly asserts that only one specialized version will ever be necessary for the specialized variable. If this assertion is not satisfied, the resulting specialized code will likely be wrong. Therefore, *cachePromotionUnchecked* is an unsafe annotation and has to be used with care.

The *cacheSplitAll* policy applies to discordant merges. After the merge versions of specialized code will be generated and cached for all different values of the variable being specialized. Moreover, a cache check guarantees that for each value at most one specialized version is generated and then reused. *cacheSplitOne* will also do a cache check, however, it will keep only the latest version in the cache. As an unsafe caching policy for merges *cacheSplitReplicate* is provided. It will always produce a new version of code at a discordant merge, even if the same code the been produced before. Consequently, no cache check is necessary. It is typically used to unroll fully static loops, where each value of the induction variable is only encountered once. Figure 10 shows an example of the use. A fully static loop is a loop that is controlled only by static conditions. This implies that given one value of the induction variable the subsequent value of the induction variable is unique. Such loops can be completely unrolled at specialization time. If each value of the induction variable occurs only once, generating a new specialized version for the different values of the induction variable will generate the loop body only once for each value of the induction variable. The effect of *cacheSplitReplicate* in this case will be identical to *cacheSplitAll* except that the cache check is omitted when the loop body is specialized, producing a cheaper loop unrolling.

The caching policy decisions are made at dynamic-to-static promotions and at merges. However, our annotations are on variables. Our system uses a propagation analysis to compute the caching policies at the dynamic-to-static promotions and merges from these annotations. The details of this analysis are specified in section 4.1.1.

3.1.4 Laziness Policies

The policies mentioned thus far allow us to control the degree of specialization and the number of code versions that will be cached. They do not control when these specialized versions are produced. We provide a third type of policies

Policy	Which Dynamic Branches Specialized Lazily?
eager	none
looplazy	earliest dynamic branches postdominated by discordant loop-head merges
polyLazy	earliest dynamic branches postdominated by any discordant merges
lazy	all

Table 3: Laziness Policies.

called *laziness policies*, that give control over this aspect. In general, we would like to specialize all the code in a dynamic region at once. Specializing a region only partially, then executing the program, and then resuming specialization, requires maintaining state information about specialization, which carries a cost. Sometimes, however, it is necessary to delay specialization. In the example shown in figure 7, the code after the `read` can only be specialized after the read operation has been executed. Also, dynamic regions can only be specialized when they are entered, as the values used in *makeStatic* annotations must be available. In general, we must delay specialization at these dynamic-to-static promotion sites.

At dynamic branches, on the other hand, we have a choice. We can specialize code for both sides right away (we call this *eager* specialization) or we can wait until a branch is actually taken. If it is likely that both sides of the dynamic branch will eventually be executed, eager specialization is advisable. If, however, one side might never be taken, eager specialization would produce more code than necessary, incurring an unnecessary cost both in space and run time. Moreover, in the presence of loops, eager specialization of dynamic branches controlling loop exits can lead to unbounded specialization.

The policies we provide to control the amount of eager specialization are summarized in table 3. *Eager* will specialize both branches of a dynamic branch, regardless if taken or not. *Lazy* is the other extreme, specializing all dynamic branches lazily, i.e., when they are taken at run time. Controlling specialization of dynamic branches inside of loops is especially important, because eager specialization inside a loop can lead to unbounded code blowup. The policy *loopLazy* is provided to specialize all code eagerly except for loop unrolling. *polyLazy* specifies that all polyvariant specialization is to be done lazily.

The eager/lazy decisions are made at dynamic branches. The annotations, on the other hand, are on variables, As in the case of caching annotations, we use a propagation analysis to compute the policies at the dynamic merges from the user annotations. The analysis is described in section 4.1.1.

3.1.5 Procedural Annotations

To optimize function calls inside of a dynamic region the annotations *specialize* and *constant* are provided. *specialize* is used to produce code for a function specialized for a subset of its parameters. Annotating the bytecode interpreter from figure 3 `specialize int interpret(byteCodeT byteCodes[], int numArgs, int arglist[]) on ((byteCodes));` in the header file will cause it to be specialized polyvariantly in its first argument, i.e., for each different value of the first argument a separate version of `f` will be generated during specialization. Specialization will be done when the value of the argument becomes available, i.e., when the function is called. A specialized version will only be called from inside a dynamic region and only if the actual arguments for which the function is to be specialized are static at the call site. If a specialized version of the function is called, a cache lookup at the call site will select the version for the actual argument value(s). To specialize the function body the static/dynamic division is computed by implicitly annotating the arguments that are specialized with $makeStatic(monovariant)$ at the top of the function body. Thus, the function will be polyvariantly specialized for its arguments, but only monovariantly inside the body at discordant merges and no dynamic-to-static promotion inside the function will be done. The annotated function and the function call can be in different files, only the same header file with the annotation on the function declaration needs to be included. Therefore, our annotation supports separate compilation. At compile time for each set of arguments a generating function will be produced that, when run, will produce specialized versions for the selected set of arguments and the actual argument values at run time. Because the *specialize* annotation allows the programmer to specify a set of arguments (for exact syntax see section 3.2), e.g., *specialize* ... *on* $(a,b),(b,c))$ our annotation also provides polyvariant division.

If a function is a mathematical function, i.e., if it is side-effect free and idempotent the *constant* annotation can be used to enable additional optimizations. If all arguments of a mathematical function are static, the call can safely be constant-folded. Because the result of the call will be static, more values can potentially be identified as static, improving code quality.

In the example, shown in figure 11, function `prod` is specialized either on all its arguments `n, v, m`, or its first and last argument `n, m`, or its second and last argument `v, m`. At call site `L1` all three arguments to `prod` are static, therefore, the version specialized for all three arguments will be called. Because the function is annotated as *constant*, the function call can also be constant-folded in the executable code. Moreover, because `i` is annotated with *makeStatic*, the loop gets unrolled, and because `s` is also static at the beginning of the loop the whole summation in the loop will be constant-folded away too. At `L2` only the first and last argument are static, therefore, a specialized version for the first and last argument will be called. Because not all arguments are static at this call site, the function call can not be constant-folded.

3.2 Syntax

makeStatic and *makeDynamic* are integrated into *C* syntax as new statement types:

```
statement:
   ...
   makeStatic-statement
   makeDynamic-statement

   makeStatic-statement:
        makeStatic (identifier-list [: policy-list]) ;
   makeDynamic-statement:
        makeDynamic (identifier-list);
```

```
/* this is in the header file: */
specialize constant int prod(int n, double v[], double* m[]) on ((n,m), (v,m),
(n,v,m));

/* this in the C-code file: */
double sum(int v[], int w[], double* m[]) {
   double s = 0;

   makeStatic(w,m,i);
   for (i=1;i<MAX;i++)
L1:   s += prod(i,w,m);

L2:
   s = s + prod(1,v,m);
   return s;
}
```

Figure 11: Use of *specialize* and *constant*.

```
policy-list:
   policy-list , policy
   policy
policy:
   specialization-policy
   eagerness-policy
   caching-policy

specialization-policy:
   monovariant
   polyvariant
   promoteStatic

laziness-policy:
   eager
   loopLazy
   polyLazy
   lazy

caching-policy:
   cachePromotionOne
   cachePromotionAll
   cachePromotionUnchecked
   cacheSplitOne
   cacheSplitAll
   cacheSplitReplicate

expression:
...
   static-expression:
      static unary-expression
```

All policy names are non-reserved keywords. Default policies are *polyvariant*, *loopLazy* and *cachePromotionAll*/*cacheSplitAll*.

The *static* annotation is integrated into C expression syntax. During type-checking we check that it is correctly used for pointer dereferencing.

As syntactic sugar for pairs of *makeStatic* / *makeDynamic* statements, the following construct is provided:

```
statement:
   dynamicRegion-statement
```

```
dynamicRegion-statement:
   dynamicRegion compound-statement
```

For instance:

```
dynamicRegion {
   makeStatic(x);
   ...
}
```

is equivalent to:

```
   makeStatic(x);
   ...
   makeDynamic(x); /* along all exits from the block */
}
```

Annotations for Function Specialization The *specialize* and *constant* annotations are integrated into C declarators:

```
declarator:
   ...
   specialize [pointer] direct-declarator (identifier-list) on
               (identifier-list-list)

   constant [pointer] direct-declarator ([identifier-list])

identifier-list-list:
   (identifier-list) , identifier-list-list
   (identifier-list)
pointer:
   * [type-qualifier-list]
   * [type-qualifier-list] pointer
```

specialize, *constant* and *on* are non-reserved keywords. Making the list of identifiers optional in the *constant* annotation enables the constant-folding of functions with zero arguments, e.g., functions that return a run-time constant configuration, such as *getpid*(), which returns the process identifier of a UNIX process. A function without arguments can obviously not be specialized on its arguments, therefore, the identifier list is not optional in the *specialize* annotation.

4 Semantics

We now present a precise definition of the semantics of our annotations through a set of analyses that compute information at the level of the CFG from the programmer's annotations in the C source. The back end of our dynamic compilation system needs the following information for its program transformations, described in [GMP+]:

- static/dynamic division information: which computations are static, which are dynamic?
- laziness information: which dynamic branches are lazy, which are eager?
- caching information: where do we need cache checks and what key is used? What is the size of the cache?

Consequently, the output of our analyses is a CFG decorated with this information.

4.1 Semantic Rules for the Analysis

Dynamically compiled code will only be produced for dynamic regions. Dynamic regions are created by *makeStatic* and *makeDynamic* annotations. Each program point has a set of root variables which are designated for specialization. Those program points where this set is empty are considered outside any dynamic region, while program points with a non-empty set are considered in a dynamic region. Code inside a dynamic region is specialized for these variables and all static values derived from these root variables.

For each of the specialization policies there is a set of the root variables that are specialized with that policy:
$S(promoteStatic)$, $S(polyvariant)$, and $S(monovariant)$
A statement is inside a dynamic region iff

$$S(promoteStatic) \cup S(polyvariant) \cup S(monovariant) \neq \{\}$$

The elements of these sets are tuples: $\langle X, Cp, Cs, L\rangle$ where X is a variable, Cp is a cache promotion policy, Cs is a cache split policy, and L is a laziness policy.

For each program point there will be at most one tuple $\langle X, Cp, Cs, L\rangle$ for variable X. Hence, in the following we will often identify the whole tuple $\langle X, Cp, Cs, L\rangle$ with variable X, i.e., when we refer to X in some set we mean the whole tuple $\langle X, Cp, Cs, L\rangle$.

Each program point has a flag ($promotionFlag$) which is set if a dynamic-to-static promotion happens at this point (initialized to F). There is another flag that indicates whether the computation is static or dynamic. Each edge in the CFG has a flag $lazyFlag$ set to either $lazy$ or $eager$ which is computed from the laziness policies. How this is done is explained later.

4.1.1 Computation of Specialization Policies

For each program point and each specialization policy, we compute the set of variables that are used for specialization at this point. These sets are denoted as $M \equiv S(monovariant)$, $P \equiv S(polyvariant)$, and $R \equiv S(promoteStatic)$ for the $monovariant$, $polyvariant$ and $promoteStatic$ policy respectively. Initially, all these sets are empty. We use the flow function SP_{flow} to indicate the effect of instructions and merges on these sets. We compute these sets in conjunction with the computation of the set of run-time constant values cs, which is described in section 4.1.2.

Flow Equations We consider the following set $Inst$ of instructions in our flow equations:

$$
\begin{array}{lcl}
Inst & \rightarrow & makeStatic(x:\ Policy) \\
 & | & makeDynamic(x) \\
 & | & x := y\ [op\ z] \\
 & | & x := f(y_1, \ldots, y_n) \\
 & | & x := [static]\ *y \\
 & | & *x := y \\
 & | & x = \&y \\
 & & \\
Policy & \rightarrow & promoteStatic \\
 & | & polyvariant \\
 & | & monovariant
\end{array}
$$

These are the flow functions:

$$SP_{flow} : Inst \rightarrow \langle Vars, Vars, Vars \rangle \rightarrow \langle Vars, Vars, Vars \rangle$$

$$
\begin{array}{lcl}
SP_{flow}[\![makeStatic(x : promoteStatic)]\!]\langle M,P,R \rangle & = & \langle M - \{x\}, P - \{x\}, R \cup \{x\} \rangle \\
SP_{flow}[\![makeStatic(x : polyvariant)]\!]\langle M,P,R \rangle & = & \langle M - \{x\}, P \cup \{x\}, R - \{x\} \rangle \\
SP_{flow}[\![makeStatic(x : monovariant)]\!]\langle M,P,R \rangle & = & \langle M \cup \{x\}, P - \{x\}, R - \{x\} \rangle \\
SP_{flow}[\![makeDynamic(x)]\!]\langle M,P,R \rangle & = & \langle M - \{x\}, P - \{x\}, R - \{x\} \rangle \\
SP_{flow}[\![x := y]\!]\langle M,P,R \rangle & = & \begin{cases} \langle M,P,R \rangle & \text{if } y \in cs \\ \langle M - \{x\}, P - \{x\}, R \rangle & \text{otherwise} \end{cases} \\
SP_{flow}[\![x := y\ op\ z]\!]\langle M,P,R \rangle & = & \begin{cases} \langle M,P,R \rangle & \text{if } \{y,z\} \subseteq cs, \\ & op \text{ idempotent,} \\ & \text{and side-effect} \\ & \text{free} \\ \langle M - \{x\}, P - \{x\}, R \rangle & \text{otherwise} \end{cases} \\
SP_{flow}[\![x := f(y_1, \ldots, y_n)]\!]\langle M,P,R \rangle & = & \begin{cases} \langle M,P,R \rangle & \text{if } f \text{ } constant\text{, and} \\ & \{y_1, \ldots, y_n\} \subseteq cs \\ \langle M - \{x\}, P - \{x\}, R \rangle & \text{otherwise} \end{cases} \\
SP_{flow}[\![x := *y]\!]\langle M,P,R \rangle & = & \langle M - \{x\}, P - \{x\}, R \rangle \\
SP_{flow}[\![x := static\ *y]\!]\langle M,P,R \rangle & = & \langle M,P,R \rangle \\
SP_{flow}[\![*x = y]\!]\langle M,P,R \rangle & = & \langle M,P,R \rangle \\
SP_{flow}[\![x = \&y]\!]\rangle M,P,R \rangle & = & \begin{cases} error & \text{if } y \in M \cup P \cup R \\ \langle M,P,R \rangle & \text{otherwise} \end{cases}
\end{array}
$$

The last rule disallows taking the address of a variable that is used for specialization. The reason for that can be seen in the example in figure 12. It would be necessary to identify at point L that x (may) be written at this point to insert a cache check and respecialize in case x was changed. Another less conservative rule could be used by doing an alias analysis to identify points where a variable in $M \cup P \cup R$ might be written.

Note that at each program point, a variable is in at most one of the sets $S(monovariant), S(polyvariant), S(promoteStatic)$. We refer to this property as the **Disjunction Property**.

Meet Operation At a control flow merge the meet operation computes the new tuple of specialization policies that are in effect after the merge point. At a

```
makeStatic(x);
p = &x; /* error */
L: *p = read();
```

Figure 12: "&" Disallowed for Specialization Variables.

dynamic merge, all variables that are specialized only monovariantly (i.e., with policy *monovariant*) are killed from the set $S(monovariant)$ and no longer used for monovariant specialization. Because the meet function uses the information whether a merge is static or dynamic, the computation of the specialization policies needs to run interleaved with the computation of the run-time constant values that computes this information and is described in 4.1.2.

For the merges only incoming non-back-edges, i.e., advancing or cross edges [ASU85], are considered. The intuition behind this is that policy annotations on variables used for specialization are propagated in roughly textual order downstream and should not extend back beyond the program point of the annotation. $\sqcap_0$ is used to select the non-back-edge(s).

For the *polyvariant* policy and the *promoteStatic* policy, polyvariant division is done by splitting the CFG in such a way that the tuple of specialization policies in each split CFG fulfills the disjunction property. Conceptually, the splitting is done by duplicating the successor instruction and assigning a different tuple of specialization policies to each. The split successor instructions are merged in the following instruction. There, further splitting might occur if different specialization policies are merging again.

The meet function is expressed in three stages: $\sqcap_0$ selects incoming non-back-edges. $\sqcap_1$ is used to apply the rule that monovariantly specialized variables are killed from their set $S(monovariant)$ at dynamic merges. Then, $\sqcap_2$ is applied to compute the merged policy. Because of splitting, the rule is recursive. Its termination is guaranteed because the number of sets that differ in the tuples $\langle M_1, P_1, R_1\rangle$ and $\langle M_2, P_2, R_2\rangle$ is strictly decreasing in the recursive call, so that eventually either the first case or one of the last two cases will apply.

If the set of variables specialized with *monovariant* differs in the incoming paths, the merge rule guarantees that all of them continue to be used for monovariant specialization, except for those that in either path were used for specialization with policy *polyvariant* or *promoteStatic*. This explains the definition of δ. When the $S(polyvariant)$ or $S(promoteStatic)$ sets differ in the two incoming paths the CFG is split into two CFGs, CFG_1 and CFG_2. $\sqcap_2$ returns different tuples for the two different CFGs. For instance, $\sqcap_2(\langle M_1, P_2, R_1\rangle, \langle M_2, P_2, R_2\rangle)$ is used for CFG_1 if $M_1 = M_2$ and $P_1 \neq P_2$, whereas for CFG_2 the tuple $\sqcap_2(\langle M_1, P_2, R_1\rangle, \langle M_2, P_2, R_2\rangle)$ is used.

Define $\delta \equiv M_1 \cup M_2 - (P_1 \cup P_2 \cup R_1 \cup R_2)$.

$$\sqcap_0(T_1, T_2) \quad = \quad \begin{cases} \sqcap_1(T_1, T_2) & \text{if both } T_1 \text{ and } T_2 \text{ come in on non-back-edges} \\ \sqcap_1(T_1, T_1) & \text{if } T_1 \text{ comes from a non-back-edge} \\ \sqcap_1(T_2, T_2) & \text{if } T_2 \text{ comes from a non-back-edge} \end{cases}$$

$$\sqcap_1(\langle M_1, P_1, R_1\rangle, \langle M_2, P_2, R_2\rangle) \quad = \quad \begin{cases} \sqcap_2(\langle \{\}, P_1, R_1\rangle, \langle \{\}, P_2, R_2\rangle) & \text{if the merge is dynamic} \\ \sqcap_2(\langle M_1, P_1, R_1\rangle, \langle M_2, P_2, R_2\rangle) & \text{if the merge is static} \end{cases}$$

where $\sqcap_2$ is defined as follows:

$$\sqcap_2(\langle M_1, P_1, R_1\rangle, \langle M_2, P_2, R_2\rangle) =$$

$$\begin{cases} \langle M_1, P_1, R_1\rangle & \text{if } M_1 = M_2 \wedge P_1 = P_2 \wedge R_1 = R_2 \\ \sqcap_2(\delta, P_1, R_1\rangle, \langle\delta, P_2, R_2) & \text{if } M_1 \neq M_2 \\ \sqcap_2(\langle M_1, P_1, R_1\rangle, \langle M_2, P_1, R_2\rangle) & \text{if } M_1 = M_2 \wedge P_1 \neq P_2 \text{ and } CFG_1 \\ \sqcap_2(\langle M_1, P_2, R_1\rangle, \langle M_2, P_2, R_2\rangle) & \text{if } M_1 = M_2 \wedge P_1 \neq P_2 \text{ and } CFG_2 \\ \langle M_1, P_1, R_1\rangle & \text{if } M_1 = M_2 \wedge P_1 = P_2 \wedge R_1 \neq R_2 \text{ and } CFG_1 \\ \langle M_1, P_1, R_2\rangle & \text{if } M_1 = M_2 \wedge P_1 = P_2 \wedge R_1 \neq R_2 \text{ and } CFG_2 \end{cases}$$

4.1.2 Computation of run-time constants

Let cs be the set of runtime constants at each program point. These sets are computed as an iterative data flow analysis using the following flow and meet functions. For our analysis we assume the program to be in SSA [CFR+89] form. In SSA form each variable has at most one definition. To guarantee this property at control flow merges, it uses so-called ϕ-functions, which select one of multiple reaching definitions.

From the set cs we compute at each program point a flag that indicates whether the instruction at the program point is a dynamic or a static computation. If all used values in the computation are static and the the written value is also static, the computation is static, otherwise it is dynamic.

Flow Equations The flow equations are identical to the first generation system flow functions ($C_{flow_{old}}$) for ordinary instructions, but at each program point add $\cup S(promoteStatic)$ to do dynamic-to-static promotion for designated variables. The promotion flag $promotionFlag$ is set using the following rule:

$$promotionFlag = \begin{cases} T & \text{if } C_{flow_{old}}[\![ins]\!]cs \neq C_{flow}[\![ins]\!]cs \\ F & \text{otherwise} \end{cases}$$

These are the new flow equations:

$C_{flow} : Inst \rightarrow Constants \rightarrow Constants$
$C_{flow}[\![x := k]\!]cs = cs \cup \{x\} \cup S(promoteStatic)$ where k is a compile-time constant

$$C_{flow}[\![x := y\ op\ z]\!]cs = \begin{cases} cs \cup \{x\} \cup S(promoteStatic) & \text{if } \{y, z\} \subseteq cs \text{ and } op \text{ idempotent} \\ & \text{and side-effect free,} \\ cs - \{x\} \cup S(promoteStatic) & \text{otherwise} \end{cases}$$

$$C_{flow}[\![x := f(y_1, \ldots, y_n)]\!]cs = \begin{cases} cs \cup \{x\} \cup S(promoteStatic) & \text{if } \{y_1, \ldots, y_n\} \subseteq cs \text{ and } f \\ & \text{annotated as } constant, \\ cs - x \cup S(promoteStatic) & \text{otherwise} \end{cases}$$

$C_{flow}[\![x := *y]\!]cs = cs - \{x\} \cup S(promoteStatic)$

By default static pointers are non-transitive.

$C_{flow}[\![x := static * y]\!]cs = cs \cup \{x\} \cup S(promoteStatic)$

$C_{flow}[\![*x := y]\!]cs = cs \cup S(promoteStatic)$

$$C_{flow}[\![x := \phi(x_1, \ldots, x_n)]\!]cs = \begin{cases} (cs - \{x_1, \ldots, x_n\}) \cup \{x\} \cup S(promoteStatic) & \text{if } \{x_1, \ldots, x_n\} \subseteq cs, \\ & \text{and the merge is static} \\ (cs - \{x, x_1, \ldots, x_n\}) \cup S(promoteStatic) & \text{otherwise} \end{cases}$$

If the merge is static[3] only one definition is guaranteed to reach the merge point, therefore we can keep x in the set of constants. If the merge is dynamic the ϕ-function might select different definitions at different times, so the x has to be killed.

$C_{flow}[\![makeStatic(x : \ldots)]\!]cs = cs \cup \{x\}$

$C_{flow}[\![makeDynamic(x)]\!]cs = S(monovariant) \cup S(polyvariant) \cup S(makeStatic) - \{x\}$

All constants that are possibly derived from x have to be killed at a *makeDynamic*. As a conservative estimate we use the set of variables that are used for specialization at the current program point minus the variable being removed from these sets by *makeDynamic*.

Reachability Conditions The reachability conditions are identical to our first generation system and are described in [APC+96]. A *static merge* is defined by the reachability conditions of the merge predecessors. If the reachability conditions of these predecessors are mutually exclusive the merge is static, otherwise it is dynamic [APC+96].

A *discordant merge* is defined by the following conditions: A merge, with run-time constant sets cs_1 and cs_2 of the two predecessors, is *discordant* (*in variable* x) iff:

1. $x \in S(promoteStatic) \cup S(polyvariant)$, and
2. the merge is dynamic, and
3. the reaching definition for x in cs_1 is different from the reaching definition for x in cs_2, i.e., there is no ϕ-function inserted at the merge.

Meet Operation

$$\sqcap(cs_1, cs_2) = \begin{cases} cs_1 \cup cs_2 \cup S(promoteStatic) \cup S(polyvariant) & \text{if the merge is static} \\ (cs_1 \cap cs_2) \cup S(promoteStatic) \cup S(polyvariant) & \text{otherwise} \end{cases}$$

4.1.3 Caching Annotations

Specialized code needs to be cached in order to be reused. Caching is necessary at dynamic-to-static promotions and at discordant merges. As part of our run-time constant analysis we also identify these merge points. The $promotionFlag$ variable indicates at which program points dynamic-to-static promotions take place. Section 4.1.2 showed how it is computed.

[3] Static merges are formally defined in the following paragraph on reachabilty conditions.

Caching Context The *caching context* is the set of program variables that are used as an index into the cache of specialized code. Cache lookup is only done for variables that are explicitly annotated by *promoteStatic*. All other static variables are derived from those root static variables and compile-time constants and are therefore not necessary in the cache check.

At each program point we maintain a set W[4] of variables in the context at this point. A variable that is promoted from a dynamic to a static value — either at a *makeStatic* annotation or when a variable with *promoteStatic* specialization policy is assigned a dynamic value — is added to the cache context W, unless its cache policy is *cachePromotionUnchecked*. Also, at discordant merges, the variables for which the merge is discordant are added to W.

We express these rules as data flow equations, with flow function W_{flow}. The set cs is the set of run-time constant values, and *promotionFlag* the flag indicating if a dynamic-to-static promotion occurs at this point, i.e., after the instruction. M, P, and R are the sets of specialization variables. How these are computed was described in section 4.1.2.

Define $\mu(x) \equiv \nexists s,l : \langle x, cachePromotionUnchecked, s, l\rangle \in (M \cup P \cup R)$.

$$
\begin{array}{lcl}
W_{flow} : Inst \rightarrow Variables \rightarrow Variables & & \\
W_{flow}[\![makeStatic(x : cachePromotionOne)]\!]W & = & \begin{cases} W \cup \{x\} & \text{if } x \in cs, \\ W & \text{otherwise} \end{cases} \\
W_{flow}[\![makeStatic(x : cachePromotionAll)]\!]W & = & \begin{cases} W \cup \{x\} & \text{if } x \in cs, \\ W & \text{otherwise} \end{cases} \\
W_{flow}[\![makeStatic(x : cachePromotionUnchecked)]\!]W & = & W \\
W_{flow}[\![makeDynamic(x)]\!]W & = & W - \{x\} \\
W_{flow}[\![x := y]\!]W & = & \begin{cases} W \cup \{x\} & \text{if } promotionFlag = T \wedge, \\ & \mu(x), \\ W & \text{otherwise} \end{cases} \\
W_{flow}[\![x := y \ op \ z]\!]W & = & \begin{cases} W \cup \{x\} & \text{if } promotionFlag = T \wedge \\ & \mu(x), \\ W & \text{otherwise} \end{cases} \\
W_{flow}[\![x := f(y_1, \ldots, y_n)]\!]W & = & \begin{cases} W \cup \{x\} & \text{if } promotionFlag = T \wedge \\ & \mu(x), \\ W & \text{otherwise} \end{cases} \\
W_{flow}[\![x := *y]\!]W & = & \begin{cases} W \cup \{x\} & \text{if } promotionFlag = T \wedge \\ & \mu(x), \\ W & \text{otherwise} \end{cases} \\
W_{flow}[\![x := static \ *y]\!]W & = & W \\
W_{flow}[\![*x := y]\!]W & = & W \\
W_{flow}[\![x := \&y]\!]W & = & \begin{cases} W \cup \{x\} & \text{if } promotionFlag = T \wedge \\ & \mu(x), \\ W & \text{otherwise} \end{cases}
\end{array}
$$

Meet operation The basic meet operation for the context is set union. Additionally, we need to add the variables that are merged discordantly at this point. At the same time, the merge rule for specialization policies described in section 4.1.1 might remove some variables from the sets of specialization variables (M, P, and R). Those that are removed do no longer have to be

[4]The English noun "context" is derived from Latin texere, to **w**eave.

included in the context. To remove them, we intersect with the set $M \cup P \cup R$. Although we will not do a cache check on variables with cache split policy *cacheSplitReplicate* we will include them in the context because we want to replicate for these variables at the merge. However, a cache check for a variable in the context with this policy will always miss, even if it is in the cache.

$$\sqcap(W_1, W_2) = (W_1 \cup W_2 \cup \{x | \text{the merge is discordant in } x\} \cap (M \cup P \cup R)$$

Cache Size Each program point has a variable *CacheSize* that gives the number of versions of codes that will be cached for this point. Values are taken from the domain $\{1, \infty\}$.

The caching policies at these points are computed as follows: Define $S \equiv S(monovariant) \cup S(polyvariant) \cup S(promoteStatic)$.

1. At discordant merges:

$$CacheSize = \begin{cases} \infty & \text{if } \exists x, p, l : \langle x, p, cacheSplitAll, l \rangle \in S \\ 1 & \text{otherwise} \end{cases}$$

2. At dynamic-to-static promotions, i.e., program points where $promotionFlag == T$.

$$CacheSize = \begin{cases} \infty & \text{if } \exists x, s, l : \langle x, cachePromotionAll, s, l \rangle \in S \\ 1 & \text{otherwise} \end{cases}$$

   The equations above are per program point, i.e., the set S and the computed variable $CacheSize$ are in fact $S(p)$ and $CacheSize(p)$, respectively.

4.1.4 Laziness annotations

Laziness annotations for variables are converted to laziness annotations on the CFG for dynamic branches. An edge is *lazy* or *eager*. If marked *lazy* it will not be followed during specialization until the edge is actually followed during execution. To compute the laziness annotations of the CFG, recall the definition of *postdomination*. A node d in a flow graph *dominates* node n if every path from the initial node of the flow graph to n goes through d. A node d *postdominates* node n if d dominates n in the flow graph with all edges reversed. We extend the notion also to edges of the graph. An edge e dominates a node n of the graph if all paths from the initial node to n follow edge e. Postdomination for edges is defined analogously as for nodes. We call an edge incident from a dynamic branch a *dynamic branch edge*.

Define the following sets:

$$\begin{array}{lcl} \Pi(n) & \equiv & \{e \mid e \text{ is postdominated by } n\} \\ \beta & \equiv & \{e \mid e \text{ is a dynamic branch edge of the CFG}\} \\ PostEdges(n) & \equiv & \{e \mid e \in \beta \cap \Pi(n)\} \\ EarliestPostEdges(n) & \equiv & \{e \mid e \in PostEdges(n) \wedge \nexists e' \in PostEdges(n) : e' \preceq e\} \end{array}$$

where $\preceq$ is a depth first topological order [ASU85] of the CFG.

In the following we write $l \in S$ as a shorthand for: $\exists x, p, s : \langle x, p, s, l \rangle \in S$. $SuccEdges(n)$ denotes the set of successor edges of node n. The laziness annotations at the CFG level are then computed as follows:

1. Initially, mark all dynamic branches *eager*
2. For each discordant merge m do
   if $polyLazy \in S$ then
   $\forall e \in EearliestPostEdges(m)$: lazyFlag = lazy
   if $loopLazy \in S$ and m a loop head merge then
   $\forall e \in EarliestPostEdges(m)$: lazyFlag = lazy
3. For each dynamic branch d do
   if $lazy \in S$ then
   $\forall e \in SuccEdges(d)$: lazyFlag = lazy

For the policies *polyLazy* and *loopLazy* we mark the earliest edges post-dominated by merge nodes to maximize the size of the region of code that is specialized eagerly (and minimize the number of laziness points). The intuition behind this rule is that we expect laziness points to be expensive and assume that specializing larger regions of code at a time will be advantageous.

4.1.5 Function Specialization

For function specialization the following steps are performed:

1. when typechecking declarations record *specialize* and *constant* annotations for functions in a table
2. when compiling a dynamic region compute the static values of the program and identify at function call sites which version of the function to call
3. when compiling a function that has a *specialize* annotation for it produce the specialized versions

Function Selection When we encounter a function call in a dynamic region we need to select the correct version of the function. If there is no *specialize* annotation for it, generate a call to the standard version. If there is a *specialize* annotation:
specialize...() *on* $(l_1, l_2, \ldots, l_n)$
in the table, select the specialization l_i as the list with the maximum number of static actuals at the call site. If there are several with the same number of arguments choose the first in the list. For example, *specialize* $f(a, b, c, d)$ *on* $((a)(a, b), (b, d))$ and the call is $f(s, s, d, s)$ (s static, d dynamic actual) then for both (a, b) and (b, d) there are two static actuals but for (a) only one. Choose (a, b) as the first listed with two.

Function specialization For each identifier list in the *specialize* annotation, a specialized function will be created. The specialized function is implicitly annotated with *makeStatic*() at the top of the function body for all arguments that are used for specialization. The policy is *monovariant* (no polyvariance or dynamic-to-static promotion *inside* the function, unless overridden by the annotations in the function body itself). This allows for polyvariant function specialization without forcing polyvariance at the sub-function level. The list of identifier lists gives polyvariant division.

5 Implementation of the BTA

Currently the policies *monovariant* and *promoteStatic* are implemented. Splitting of the CFG and the *polyvariant* policy are not yet implemented. The $S(promoteStatic)$ is propagated first, then runtime constants and $S(monovariant)$ sets are computed concurrently as an iterative data flow analysis (IDFA). The *promoteStatic* policy can be propagated first because there is no dependency on results from the run-time constant analysis. As the *promoteStatic* annotations only have to be propagated forward, an IDFA is not necessary and simple forward propagation is more efficient. As part of the IDFA discordant merges are identified and the context set W is computed, both of which are needed by later phases for caching and breaking the code into smaller units.

6 Future Work

There are several directions for future work. First of all, the full range of specialization policies needs to be implemented. This will allow us to assess their practical usefulness. For the caching annotations, a fixed cache size n instead of 1 or ∞ might be useful. This will require looking into cache replacement policies, which are not needed in the current design.

Implementing an alias analysis would allow us to be less conservative about taking the address of static variables specialized with *makeStatic*. Currently, variables that have their address taken cannot be used for specialization. We don't know, however, if this restriction presents an actual problem and need to look at real examples where the problem might show up.

In the current framework all derived static values are forgotten at a *makeDynamic* annotation. An additional analysis keeping track of the derivation of static values could provide a less conservative flow function for *makeDynamic*. We could then test if the current conservative rule carries a negative impact on the quality of the generated code. If so, a more complicated rule taking into account the derivation graph could be used to improve code quality.

Derivation analysis together with a live variable analysis might improve caching. As soon as a variable in the context and all its derived static values become dead, the variable can safely be removed from the caching context as none of the downstream code is specialized on its value or any value derived from it.

Identifying good candidate variables for specialization is another open problem. It may be hard for a programmer to find out where to apply dynamic compilation, especially if he tries to dynamically compile third-party code. Profiling support in the dynamic compiler could help in identifying good variables. For instance, knowing how many different values a variable takes on might provide some hints.

While dynamic compilation removes large amounts of overhead from interpreted programs, overheads remain relative to statically compiled programs. Static compilers do register allocation and scheduling on the compiled code. Specializing an interpreter for a particular program will likely require run-time register allocation and scheduling to produce code that performs equally well.

Doing runtime register allocation and scheduling will probably require some form of intermediate representation in the dynamically-produced code to allow for further optimizations. Designing an expressive intermediate language that can be cheaply compiled to native code is another area worth of investigation.

References

[APC+96] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian Bershad. Fast, effective dynamic compilation. In *ACM SIGPLAN '96: Programming Language Design and Implementation. Philadelphia, PA, USA*, pages 149–49, May 1996.

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1985.

[CEA+96] Craig Chambers, Susan J. Eggers, Joel Auslander, Matthai Philipose, Markus Mock, and Przemek Pardyak. Automatic dynamic compilation support for event dispatching in extensible systems. In *Proceedings of the Workshop on Compiler Support for Systems Software, February 1996*, February 1996. Presented at the Workshop on Compiler Support for Systems Software, February 1996.

[CFR+89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Symposium on Principles of Programming Languages*, pages 25–35. ACM, January 1989.

[CN96] Charles Consel and Francois Noel. A general approach for runtime specialization and its application to C. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, January 1996.

[CUL89] Craig Chambers, Dave Ungar, and E. Lee. An efficient implementation of self, a dynamically-typed object-oriented language based on prototypes. In *Sigplan Notices*, volume 24, pages 49–70, October 1989.

[EHK96] Dawson R. Engler, Wilson C. Hsieh, and M. F. Kaashoek. 'C: a language for high-level, efficient, and machine-independent dynamic code generation. In ACM, editor, *Symposium on Principles of Programming Languages*, January 1996.

[GMP+] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan Eggers. Annotation-direction run-time specialization in C. To appear in the Proceedings of the ACM SIGPLAN Conference on Partial Evaluation and Semantics-Based Program Manipulation, (PEPM'97).

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestroft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[LL96] Mark Leone and Peter Lee. Optimizing ML with run-time code generation. In *ACM SIGPLAN '96: Programming Language Design and Implementation. Philadelphia, PA, USA*, pages 137–148. ACM, May 1996.