Fast, Effective Dynamic Compilation

Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad

Department of Computer Science and Engineering
University of Washington

Abstract

Dynamic compilation enables optimizations based on the values of invariant data computed at run-time. Using the values of these run-time constants, a dynamic compiler can eliminate their memory loads, perform constant propagation and folding, remove branches they determine, and fully unroll loops they bound. However, the performance benefits of the more efficient, dynamically-compiled code are offset by the run-time cost of the dynamic compile. Our approach to dynamic compilation strives for both fast dynamic compilation *and* high-quality dynamically-compiled code: the programmer annotates regions of the programs that should be compiled dynamically; a static, optimizing compiler automatically produces pre-optimized machine-code templates, using a pair of dataflow analyses that identify which variables will be constant at run-time; and a simple, dynamic compiler copies the templates, patching in the computed values of the run-time constants, to produce optimized, executable code. Our work targets general-purpose, imperative programming languages, initially C. Initial experiments applying dynamic compilation to C programs have produced speedups ranging from 1.2 to 1.8.

"One man's variable is another man's constant."
— adapted from Alan J. Perlis [Per90]

1 Introduction

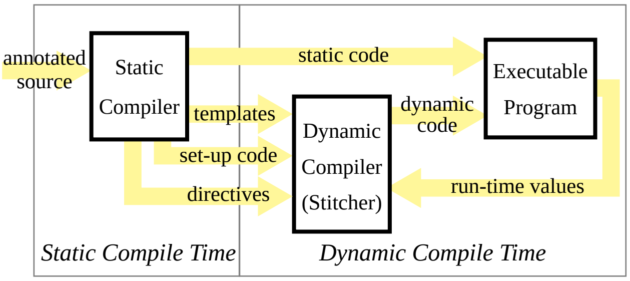
Traditional compilation performs optimizations that either are independent of the actual values of program variables or depend on the values of compile-time constants. It is unable to optimize around variables whose values are invariant during program execution, but are unknown until then. Consequently, these variables must be reinterpreted on each run-time use and cannot trigger value-based optimizations.

Our work applies *dynamic compilation* to enlarge the scope of optimization to include those that depend on the values of variables that are computed at run-time, and once computed, remain fixed for some significant period of time. By compiling performance-critical parts of the program at run-time, more efficient code can be produced. For example, run-time constants can become instruction immediates rather than memory loads, constant propagation and folding can be applied to them, conditional branches based on them can be eliminated, and loops they control can be fully unrolled. Data structures can be considered run-time constants whenever they are accessed through run-time constant pointers. Applications with such run-time constants include interpreters (where the data structure that represents the program being interpreted is the run-time constant), simulators (where the circuit or architecture description is a run-time constant), graphics renderers (where the scene or viewing parameters are constant), numerical codes (where scalars, vectors, matrices, or the patterns of sparsity can be run-time constant), and extensible operating system kernels (where the current set of extensions to the kernel is run-time constant [BSP+95,CEA+96]).

Dynamic compilation can realize overall performance improvements, however, only if the execution time savings from the dynamically-generated code is greater than the time to compile at run-time. Different trade-offs between dynamic compilation time and dynamically-compiled code quality are possible, and previous systems have explored several alternatives [CN96,EHK96,LL96]. Our approach strives to achieve the best of both options: fast dynamic compilation and high-quality dynamically-generated code. We do this by planning out most of the actions of dynamic compilation in advance (during static compilation), based on the static knowledge of which variables and data structures will be invariant at run-time, but without knowing their exact run-time values.

Our dynamic compilation system targets general-purpose, imperative programming languages, initially C. Because of the difficulty in this general setting of automatically identifying which data structures will be invariant over which portions of the program, and where this invariance will be profitable to exploit through dynamic compilation, our current implementation relies on simple, programmer-inserted annotations. The annotations indicate which parts of the program should be compiled dynamically (called *dynamic regions*) and which source variables will be constant during execution of the dynamic regions. Through a kind of constant propagation and constant folding, our system automatically identifies other derived run-time constants in the region.

Our dynamic compilation system is composed of both a static and a dynamic compiler. To achieve fast dynamic compile times, the static compiler produces pre-compiled *machine-code templates,* whose instructions contain *holes* that will be filled in with run-time constant values. The static compiler also generates *set-up code* to calculate the values of derived run-time constants, and *directives* that instruct the dynamic compiler how to produce executable code from the templates and the set-up code's computed constants.



Given this advance preparation, the dynamic compiler (called the *stitcher*) simply follows the directives to copy the machine-code templates and fill in the holes with the appropriate constants. The run-time overhead of dynamic compilation (the stitcher, set-up code, and directives) is executed at most once per dynamic region; the dynamically-compiled templates become part of the application and usually are executed many times.

In PLDI'96

To generate high-quality dynamically-compiled code, the static compiler applies standard global optimizations to the machine-code templates, optimizing them in the context of their enclosing procedure. It also plans out the effect of run-time constant-based optimizations, so that the final, optimized templates contain only the calculations that remain after these optimizations have been performed.

Our dynamic compilation work is novel in several respects. First, it is capable of handling the full functionality of C, without restricting its normal programming style. Second, automatic run-time constant derivation is accomplished via two interconnected dataflow analyses, one that identifies run-time constants and another that determines their reachability conditions downstream of run-time constant branches. When executed in parallel, they provide an analysis that is general enough to handle unstructured control flow. Finally, by integrating our analyses into an optimizing compiler, dynamically-compiled code can be heavily optimized with its surrounding code, with few limitations on the kinds of optimizations that can be applied.

This paper presents our dynamic compilation framework and the algorithms that optimize dynamic regions with respect to their run-time constants. The next section outlines the programmer annotations that drive the analysis. Sections 3 and 4 describe the static and dynamic compilers, respectively. Section 5 reports on empirical studies of the effectiveness of dynamically compiling several programs. Section 6 provides a detailed comparison to related work, and section 7 concludes with a summary of our main contributions and directions for future work.

2 Programmer Annotations

Our current implementation relies on programmer annotations to specify which regions of code to dynamically compile, which variables to treat as run-time constants, and which loops to completely unroll. To illustrate the annotations (as well as the rest of the dynamic compilation framework), we will use the following cache lookup routine of a hypothetical cache simulator (the bold keywords are the programmer-supplied annotations):[*]

```
cacheResult cacheLookup(void *addr, Cache *cache) {
  dynamicRegion(cache) { /* cache is run-time constant */
    unsigned blockSize = cache->blockSize;
    unsigned numLines = cache->numLines;
    unsigned tag =
      (unsigned) addr / (blockSize * numLines);
    unsigned line =
      ((unsigned) addr / blockSize) % numLines;
    setStructure **setArray =
      cache->lines[line]->sets;
    int assoc = cache->associativity;
    int set;
    unrolled for (set = 0; set < assoc; set++) {
      if (setArray[set]dynamic->tag == tag)
        return CacheHit;
    }
    return CacheMiss;
  } /* end of dynamicRegion */
}
```

`dynamicRegion` delineates the section of code that will be dynamically compiled (in the example, the body of the `cacheLookup` function). The arguments to **`dynamicRegion`** indicate which source variables are constant at the entry of the dynamic region and remain unchanged throughout this and all future executions. The static compiler automatically computes all run-time constants that are derived from this initial set, as described in section 3.1. There is no restriction on the kind of data that we can treat as a run-time constant; in particular, the contents of arrays and pointer-based data structures are assumed to be run-time constants whenever accessed through run-time constant pointers. For partially-constant data structures, we use an additional annotation on memory dereference operators to indicate that the result of the dereference is a variable, even if its argument is constant, e.g., `x :=` **`dynamic*`** `p`, `x := p` **`dynamic->`**`f`, and `x := a` **`dynamic[`**`i`**`]`**. (In the above example, the tags stored in the cache are not constant.[†])

`unrolled` directs the dynamic compiler to completely unroll a loop. The loop termination condition must be governed by a run-time constant. Complete unrolling is a critical dynamic optimization, because it allows loop induction variables and data derived from them to be run-time constants (the value of an induction variable in each unrolled copy of the loop is a distinct, fixed value). Since not all loops governed by run-time constants are profitable or practical to unroll, we only unroll annotated loops. We can automatically check whether an annotated loop is legal to unroll, using the analyses described in section 3.1.

For some applications, it is important to produce several compiled versions for a single dynamic region, each optimized for a different set of run-time constants. For example, if the cache simulator were simulating multiple cache configurations simultaneously, each configuration would have its own `cache` values and need cache lookup code specialized to each of them. Accordingly, we allow a dynamic region to be *keyed* by a list of run-time constants. Separate code is generated dynamically for each distinct combination of values of the key variables; the generated code is cached and reused for later invocations of the region with the same key values. The dynamic region of the multi-cache simulator example would be annotated as follows:

```
dynamicRegion key(cache) (/* no other constants */) { ... }
```

Given these annotations, our system manages all other aspects of dynamic compilation automatically. Programmers are insulated from the dynamically-compiled code, and the interface to procedures containing dynamic regions is unchanged. This is in contrast to some other systems that require more programmer involvement in the management of dynamically-compiled code [EP92,EHK96,CN96].

We chose this set of annotations as a balance between a completely manual approach and a completely automatic approach. Given a few select annotations, our compiler can automatically identify the derived run-time constants and perform several important optimizations, saving the programmer much of the tedious and error-prone effort of a fully-manual approach. The drawback is that errors in the annotations can lead to incorrect optimizations being performed dynamically. Unfortunately, automatic, safe, and effective dynamic compilation is quite challenging: it requires whole-program side-effect and pointer analysis to reason about invariance of variables and (parts of) data structures, analysis of loops to judge termination conditions, profile information to choose both dynamic regions that are the most profitable and loops that make sense to fully unroll, and so on. Data structures that are invariant for only part of a program's execution or routines that should be replicated for different invariant clients are even harder to handle automatically. Our long-term goal is to try to automate most of the dynamic compilation process, but for now our simple annotations are both practical to use and facilitate early experimentation with different choices for run-time constants and dynamic regions. Annotations may also be useful as a human-readable intermediate representation for more automatic implementations.

[*] Our current implementation uses a collection of lower-level annotations that provide the same information but do not require modifications to the C parser.

[†] For this example, it turns out that this annotation is unnecessary, since the dereferenced pointer is not run-time constant.

3 The Static Compiler

The static compiler compiles procedures that do not contain dynamic regions normally. For procedures with dynamic regions, it performs the following four steps:

- It identifies which variables and expressions within the dynamic region will be constant at run-time, based on the set of variables annotated at the start of the region. This step plans the constant propagation, constant folding, dead code elimination, and loop unrolling that will be performed by the stitcher at run-time.
- It splits each dynamic region subgraph into set-up and template code subgraphs, replacing the region's original subgraph with the corresponding pair of subgraphs.
- It optimizes the control flow graph for each procedure, applying all standard optimizations with few restrictions.
- It generates machine code and stitcher directives.

We have chosen to embed our support for dynamic compilation into a standard, optimizing compiler framework for two reasons. First, we wished to generate high-quality dynamically-compiled code; we therefore integrated our specialized analyses into an infrastructure that already performed sophisticated optimizations. Second, we wished to support a variety of general-purpose programming languages, including C, without restricting their normal programming style. Accordingly, our analyses and transformations operate at the lower but more general level of control flow graphs connecting three-address code [ASU86], rather than the higher, language-specific level of abstract syntax trees (as does some other work in this area [CN96,KR96,LL96]). Our analyses go to some length to support partially unstructured[*] control flow graphs well, since these graphs occur frequently in C programs. We consider the increased generality of our analyses to be an important contribution of our work.

The rest of this section discusses the four steps executed by the static compiler when compiling code that contains dynamic regions.

3.1 Computing Derived Run-Time Constants

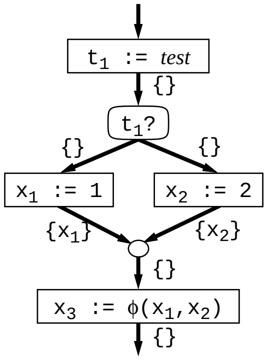
As the first step in compiling a dynamic region, the static compiler computes the set of variables and expressions that are constant[†] at each point in the dynamic region, given the set of constants annotated by the programmer. This analysis is similar to binding time analysis in off-line partial evaluators [SZ88,JGS93] (except that our analysis is at the level of control flow graphs rather than abstract syntax trees) and to traditional constant propagation and folding (except that our analysis must cope with knowing only that a variable will be constant, not what the constant value is). We have developed a pair of interconnected analyses, one that computes the set of run-time constant variables at each program point, and another that refines that solution by computing reachability information downstream of run-time constant branches. We first describe the run-time constants analysis alone, and then augment it with the reachability analysis. Appendix A contains a more precise specification of our algorithms.

The run-time constants analysis is a forward dataflow analysis that computes the set of variables that are run-time constants for each program point. (To simplify the exposition, we assume that the dynamic region is in static single assignment (SSA) form [AWZ88,CFR$^+$89].) At the start of a region, the set of constants is the set of variables specified by the programmer. Analysis proceeds by propagating this initial set of constants through the dynamic region's control flow graph, updating the set after each instruction as follows:

- `x := y`: `x` is a constant iff `y` is a constant.
- `x := y op z`: `x` is a constant iff `y` and `z` are constants and `op` is an idempotent, side-effect-free, non-trapping operator; for example, `/` is excluded, since it might trap. Unary operations are handled similarly.
- `x := f(`y_1`, ..., `y_n`)`: `x` is a constant iff the y_i are constants and `f` is an idempotent, side-effect-free, non-trapping function, such as `max` or `cos`. `malloc` is excluded, since it is not idempotent.
- `x := *p`: `x` is a constant iff `p` is a constant.
- `x :=` **`dynamic*`** `p`: `x` is not a constant.
- `*p := x`: Stores have no effect on the set of constants. A load through a constant pointer whose target has been modified to hold a non-constant value during execution of the dynamic region should use **`dynamic*`**.

After a control flow merge, if a variable has the same run-time-constant reaching definition along all predecessors, it is considered a constant after the merge. However, if a variable has different reaching definitions along different predecessors, the value of the variable may not be a run-time constant after the merge, even if all reaching definitions before the merge are constant. For example, in the following control flow graph, if *test* is not a constant, after the merge `x` could be `1` on some executions and `2` on others, and hence it cannot be treated as a run-time constant. (In the figure, the sets labeling each arc represent the computed constant sets at those program points.)

```
/* assume test is not constant */
if (test) {
  x1 = 1;
} else {
  x2 = 2;
}
/* x1 and x2 are constants */
x3 = φ(x1,x2);
/* x3 is not constant */
```



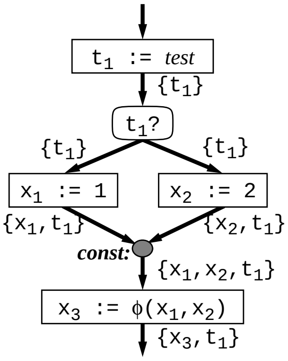
On the other hand, if *test* is a constant, for any execution of the dynamic region in a given program run, either *test* is always true and `x` is always `1` after the merge, or *test* is always false and `x` is always `2` after the merge. In the first case, the φ function after the merge is not an idempotent operator (and so its result is not constant

[*] By "unstructured" we mean graphs that are not composed solely of nested single-entry/single-exit regions corresponding to syntactic nesting, but rather have some control flow transfers that do not respect the syntactic nesting structure. By this definition, commonly-occurring unstructured constructs in C include `switch` statements with fall-through `case`'s, `break` and `continue` statements, and `goto` statements (for instance, implementing multi-level loop exits and hand-eliminated tail recursion).

[†] For brevity we use the term "constant" to refer to run-time constants, which include compile-time constants as a special case.

irrespective of whether its arguments are constant), while in the second case it is.

```
/* test is a constant */
if (test) {
  x1 = 1;
} else {
  x2 = 2;
}
/* x1 and x2 are constants */
x3 = ϕ(x1,x2);
/* x3 is constant */
```



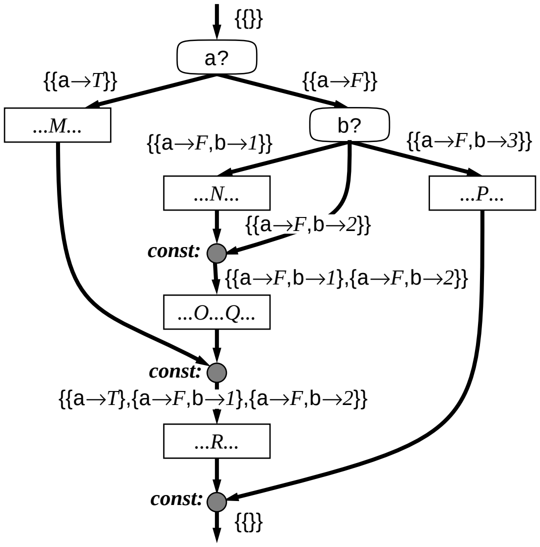
Identifying *constant merges* whose corresponding branches have constant predicates can be done for structured, nested control flow graphs by identifying diamond-shaped, single-entry/single-exit subgraphs.[*] However, to handle general C programs well, we need to identify constant merges even in unstructured control flow graphs. Accordingly, we supplement our run-time constants analysis with a reachability analysis that computes the conditions (in terms of branch outcomes for run-time constant branches) under which each program point can be reached during execution. Then, if the reachability conditions for each merge predecessor are mutually exclusive, the merge is labeled as a constant merge and can use the better idempotent-ϕ rule; otherwise, the merge must use the more conservative non-idempotent-ϕ rule.

Our reachability analysis is a forward dataflow analysis that is performed in parallel with the run-time constants analysis.[†] The reachability analysis computes conjunctions and disjunctions of branch conditions at each program point, where each branch condition has the form B→*S* for a constant branch B (either 2-way or *n*-way) that goes to successor arc *S*. We use sets of sets to represent disjunctions of conjunctions, in conjunctive normal form (CNF). For example, the set {{A→*T*}, {A→*F*, B→*1*}}, computed for program point p, can only be executed if A's constant predicate is true or if A's constant predicate is false and B's constant switch value takes case 1.

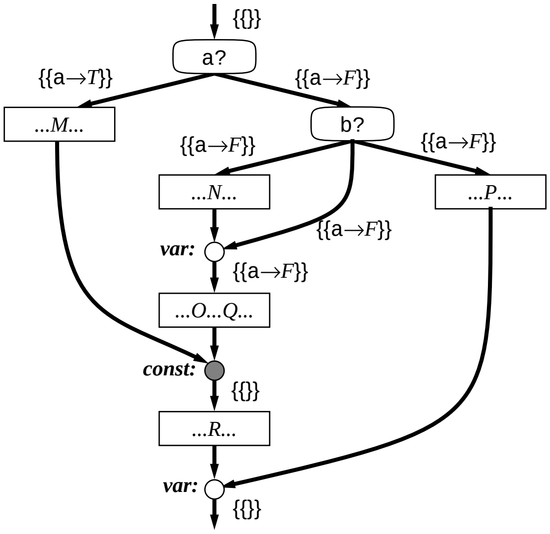
At the start of the dynamic region, the reachability condition is true (represented by the set {{}}), since the entry is always reachable. Straight-line code and branches whose predicates are not run-time constants have no effect on the reachability analysis. A branch B whose predicate is a run-time constant updates the reachability set along successor *S* by and'ing in the condition B→*S* (in the CNF set representation, B→*S* is added to each of the element sets). At merges, the incoming conditions are or'd together. (At the representation level, the incoming sets are combined with set union, followed by simplifications which reduce sets of the form {{A→*T*,*Cs*},{A→*F*,*Cs*},*Ds*} to {{*Cs*},*Ds*}.) The following example illustrates the results of reachability analysis on an unstructured control flow graph for two different situations (the labels on the arcs in this figure are reachability conditions in the CNF set representation, not sets of run-time constants):

```
if (a) {
  ...M...;
} else {
  switch (b) {
    case 1: ...N...; /* fall through */
    case 2: ...O...; break;
    case 3: ...P...; goto L;
  }
  ...Q...;
}
...R...;
L:
```

If a and b are constant branches:



If only a is a constant branch:



In the upper graph, where both a and b are constant, the reachability analysis determines that all three merges are constant merges, since the reachability conditions of the predecessors of each of the merges are mutually exclusive. As a result, any variables identified as run-time constant along all predecessor branches of a merge will be considered constant after the merge. In the lower graph, only one of the merges is constant.

Conjunctions of branch conditions support sequences of branches, while disjunctions of branch conditions are crucial for coping with unstructured graphs.[‡] An analysis based solely on abstract syntax

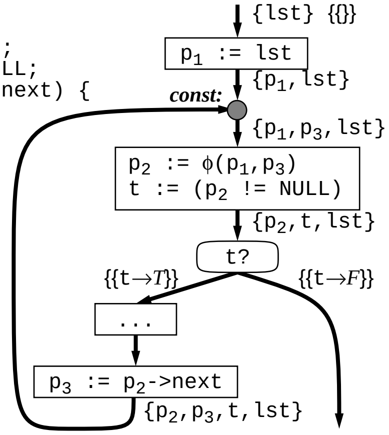
[*] Alpern *et al.* extended ϕ functions to include an argument representing the corresponding branch predicate, for structured if and loop constructs [AWZ88]. This would allow ϕ to be treated as idempotent for all merges: if all the reaching definitions *and* the branch predicate were constant, then the result would be constant. Unfortunately, this technique does not extend easily to unstructured control flow graphs.

[†] The reachability analysis uses the results of run-time constants analysis to identify run-time constant branches, and the run-time constants analysis uses the results of reachability analysis to choose between ϕ merge rules [CC95].

trees would have a difficult time identifying as many run-time constants on unstructured programs.

A loop head is a merge node. Since the reachability conditions of the loop entry arc and the loop back edge arc will not normally be mutually-exclusive, our analysis as described so far will treat the loop head as a non-constant merge. This is safe, and it is appropriate for loops that are not unrolled. For unrolled loops, however, only one predecessor arc will ever enter an unrolled copy of the loop head merge at run-time: either the loop entry predecessor (for the first iteration) or the loop back edge predecessor (for subsequent iterations). Accordingly, we mark all loop heads for unrolled loops as constant merges. As a consequence, loop induction variables can be identified as constants within the loop body. The following example illustrates how labeling an unrolled loop head as constant enables the main loop induction variable `p` to be marked constant (arcs are labeled by constant sets and/or reachability conditions, depending on what information changes at that arc):

```
/* lst is constant */
unrolled for(p = lst;
             p != NULL;
             p = p->next) {
  /* p is constant */
  ...
}
```



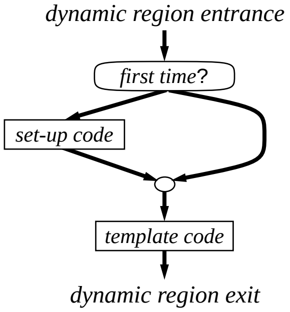
The following code shows (using underlining) the expressions identified as run-time constants in the dynamic region of the cache lookup example from section 2:

```
dynamicRegion(cache) {
  unsigned blockSize = cache->blockSize;
  unsigned numLines = cache->numLines;
  unsigned tag =
    (unsigned) addr / (blockSize * numLines);
  unsigned line =
    ((unsigned) addr / blockSize) % numLines;
  setStructure **setArray =
    cache->lines[line]->sets;
  int assoc = cache->associativity;
  int set;
  unrolled for (set = 0; set < assoc; set++) {
    if (setArray[set]dynamic->tag == tag)
      return CacheHit;
  }
  return CacheMiss;
}
```

3.2 Extracting Set-Up Code and Templates

After identifying run-time constant calculations, the static compiler divides each dynamic region into two separate subgraphs: set-up code and template code. Set-up code includes all the calculations that define run-time constants. Templates contain all the remaining code within the region, with "holes" embedded in some instructions for run-time constant operands. Additionally, templates contain marker pseudo-instructions identifying the entry, exit, and back edge arcs of unrolled loops to help generate stitcher directives (described in section 3.4). The control flow connections of the two subgraphs are the same as in the original dynamic region.

Once constructed, these two subgraphs replace the original subgraph of the dynamic region, roughly as follows:



The set-up code is executed only the first time the dynamic region is entered, and it calculates all the run-time constants needed in the region. The set-up code stores all the constants referenced by template code in a table data structure, which it passes to the dynamic compiler for use in instantiating templates into executable code (as described in section 4). For most code, run-time constant table space can be pre-allocated, enabling the set-up code to store computed constants quickly into the table. However, for fully-unrolled loops, an unbounded amount of space may be needed. We solve this problem by allocating a fresh record for each iteration of an unrolled loop. Within each iteration, we can statically allocate the run-time constants computed within that iteration.

The set-up and template code for the cache lookup routine (expressed in C rather than as a flow graph, for readability) is shown in Figure 1. The set-up code calculates all constants and stores those needed by the template code into the table `t`. The last element of `t` acts as the head of a linked-list of table records for the run-time constants within the unrolled loop. The template code contains hole markers for the necessary run-time constants, plus markers that delimit the unrolled loop's iterations.

3.3 Optimization

A major goal of our approach is to allow full optimization to be performed on procedures that contain dynamic regions. In particular, we wish optimizations such as global common subexpression elimination and global register allocation to be performed across dynamic region boundaries. Optimizations can be performed both before and after the body of the dynamic region is divided into set-up and template code. We place no restriction on optimizations performed before this division. Optimizations performed afterwards must be modified slightly to deal with the special semantics of "hole" operands in templates.

For the most part, the compiler's analyses can treat each hole marker as a compile-time constant of unknown value. However, in a few circumstances hole markers must be treated differently:

- Instructions in a template subgraph that contain hole markers cannot be moved (e.g., by code motion or instruction scheduling) outside the template subgraph.
- Hole marker values should not be treated as legal values outside the dynamic region. In particular, copy propagation should not propagate references to holes outside the dynamic region.
- Holes that correspond to induction variables defined in run-time unrolled loops cannot be treated as loop-invariant with respect to the unrolled loop; each iteration of the unrolled loop will get its own distinct version of the value.

In our current implementation, we conservatively satisfy these requirements by placing barriers to code motion and other

[‡] The extra flexibility of being able to represent disjunctions does, however, lead to a worst-case size of a reachability condition for a program point that is exponential in the number of constant branches in the dynamic region. In practice, the size of reachability conditions has been small.

optimizations at the start and end of the template code and at the head of unrolled loops.

Original dynamic region, after run-time constants identification:

```
dynamicRegion(cache) {
  unsigned blockSize = cache->blockSize;
  unsigned numLines = cache->numLines;
  unsigned tag =
    (unsigned) addr / (blockSize * numLines);
  unsigned line =
    ((unsigned) addr / blockSize) % numLines;
  setStructure **setArray =
    cache->lines[line]->sets;
  int assoc = cache->associativity;
  int set;
  unrolled for (set = 0; set < assoc; set++) {
    if (setArray[set]dynamic->tag == tag)
      return CacheHit;
  }
  return CacheMiss;
}
```

Set-up code:

```
t = allocateTable(5); /* allocate space for constant table */
t[0] = t0 = cache->blockSize;
t[1] = t1 = cache->numLines;
t[2] = t0 * t1;
t[3] = cache->lines;
assoc = cache->associativity; /* not used in templates */
loopTable = &t[4]; /* head of unrolled loop's list of tables */
for (set = 0; ; set++) {
  lt = *loopTable = allocateTable(3);
  lt[0] = lt0 = (set < assoc);
  if (!lt0) break;
  lt[1] = set;
  loopTable = &lt[2]; /* next pointer for loop's linked list */
}
```

Template code (where $hole_{4:x}$ references the x^{th} entry of the appropriate iteration of the loop headed by t[4]):

```
L0: enter region marker;
L1: unsigned tag = (unsigned)addr / $hole_2$;
L2: unsigned t1 = (unsigned)addr / $hole_0$;
L3: unsigned line = t1 % $hole_1$;
L4: set_structure **setArray = $hole_3$[line]->sets;
L5: unrolled loop entry marker;
L6: constant branch marker ($hole_{4:0}$)
L7: t2 = setArray[$hole_{4:1}$]->tag;
L8: if (t2 == tag) {
L9:   unrolled loop exit marker;
      return CacheHit; }
L10: unrolled loop back edge marker;
L11: unrolled loop exit marker;
     return CacheMiss;
L12: exit region marker;
```

Stitcher directives, ignoring labels:

```
START(L0)
HOLE(L1,2,2) HOLE(L2,2,0)
HOLE(L3,2,1) HOLE(L4,1,3)
ENTER_LOOP(L5,4)
CONST_BRANCH(L6,4:0)
HOLE(L7,2,4:1)
BRANCH(L8)
EXIT_LOOP(L9)
RESTART_LOOP(L10,4:2)
EXIT_LOOP(L11)
END(L12)
```

Shape of constants table computed by set-up code:

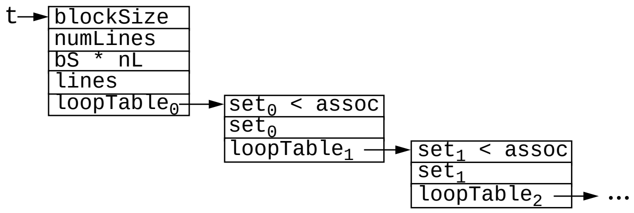


Figure 1: Set-Up Code, Templates, and Directives

3.4 Code Generation

The final step of static compilation is to generate machine code from the optimized control flow graph. For dynamic regions, code generation also produces stitcher directives. The stitcher directives form a simple instruction set, described in Table 1. Directives are generated as a side-effect of generating code for instructions containing holes (to inform the stitcher to patch in the hole's run-time value), for the markers that identify the entry, exit, and back-edge branches of unrolled loops (to inform the stitcher how to break up the pieces of the loop for loop unrolling), and for any pc-relative instructions, such as branches (which need to be adjusted when the stitcher copies the templates). Section 4 describes the actions performed by the stitcher for these directives.

Table 1: Stitcher Directives

Directive	When Generated
START(*inst*)	beginning of template code
END(*inst*)	end of template code
HOLE(*inst*, *operand #*, *const's table index*)	hole marker operand in instr
CONST_BRANCH(*inst*, *test's table index*)	hole marker as branch test
ENTER_LOOP(*inst*, *table header index*)	unrolled loop entry marker
EXIT_LOOP(*inst*)	unrolled loop exit marker
RESTART_LOOP(*inst*, *next table index*)	unrolled back edge marker
BRANCH(*inst*)	pc-relative instruction
LABEL(*inst*)	target of pc-relative instr.

The stitcher directives for the cache lookup example appear in Figure 1.

4 The Stitcher

Given the preparation by the static compiler, the stitcher has only to follow the directives to instantiate the machine-code templates. Most of these tasks are straightforward, such as copying blocks of machine code between directive labels and updating offsets in pc-relative branch and call instructions. For run-time constant branches, the stitcher uses the corresponding value in the run-time constants table (computed by the set-up code) to select the appropriate successor path, implicitly performing dead code elimination of the other path(s). For unrolled loops, the stitcher traverses links in the run-time constants table to access the appropriate subtable for each iteration; the directives at loop entries, exits, and back edges instruct the stitcher when to switch between subtables.

The stitcher also patches the values of run-time constants into the machine-code template holes. For an integer operand, the static compiler has selected an instruction that admits the hole as an immediate operand, and the stitcher first tries to fit the run-time constant integer value into it. If the value is too large, the stitcher either generates code to load the value from a table of large run-time constants or constructs it from smaller constants. For floating-point and pointer constant operands, which typically will not fit into the immediate field, the static compiler inserts the load instruction as part of producing template code, so that the load can be better scheduled during regular static optimizations.

As initially constructed, the nested structure of the run-time constants table requires a fair amount of run-time bookkeeping to track which loop iteration is being handled. In addition, accesses to the table are likely to be sparse, since only the large or non-integer

Table 2: Speedup and Breakeven Point Results

Benchmark	Run-time Constant Configurations	**Asymptotic Speedup** (static/ dynamic region times)	Breakeven Point	**Dynamic Compilation Overhead**: **set-up & stitcher** (1000s cycles)	**Cycles/Instruction Stitched** (number of instructions stitched)
Reverse-polish stack-based desk calculator	$2xy - 3y^2 - x^2 + (x+5) * (y-x) + x + y - 1$	**1.7** (1690/997)	**916** interpretations with different x, y values	452 183	**734** (865)
Scalar-matrix multiply (adapted from [EHK96])	800×800 matrix, multiplied by all scalars 1..100	**1.6** (16000/10000)	**31,392** individual multiplications	260 34.3	**4032** (73)
Sparse matrix-vector multiply	200×200 matrix, 10 elements/row, 5% density	**1.8** (76200/43200)	**2645** matrix multiplications	83,700 3,580	**7390** (11810)
	96×96 matrix, 5 elements/ row, 5% density	**1.5** (13200/8840)	**1858** matrix multiplications	7,070 1,030	**2478** (3269)
Event dispatcher in an extensible OS [BSP$^+$95,CEA$^+$96]	7 predicate types; 10 different event guards	**1.4** (4606/3228)	**722** event dispatches	638 357	**597** (1667)
Quicksort record sorter (extended from [KEH93])	4 keys, each of a different type	**1.2** (1140/960)	**3050** records	444 105	**8446** (65)
	12 keys, each of a different type	**1.2** (1310/1060)	**4760** records	790 400	**6869** (173)

run-time constants are accessed indirectly. To avoid these problems, the stitcher constructs a second linearized, compressed array to hold the large and non-integer constants. Since loads from the linearized table are fast, requiring only a dedicated base pointer register and a fixed index for each reference, the stitcher fills holes for large and non-integer constants with references to the linearized table. The structured table is deallocated after stitching completes.

The stitcher performs simple peephole optimizations that exploit the actual values of the constant operands. For example, integer multiplications by constants are rewritten as shifts, adds, and subtracts, and unsigned divisions and modulus's by powers of two become shifts and bit-wise and's, respectively.

The final code generated by the stitcher for the cache lookup example, invoked for a cache configuration of 512 lines, 32-byte blocks, and 4-way set associativity, is the following (where *cacheLines* is an address loaded from the linearized run-time constants table):

```
  unsigned tag = (unsigned)addr >> 14;
  unsigned line = ((unsigned)addr >> 5) & 511;
  setStructure **setArray = cacheLines[line]->sets;
  if (setArray[0]->tag == tag) goto L1;
  if (setArray[1]->tag == tag) goto L1;
  if (setArray[2]->tag == tag) goto L1;
  if (setArray[3]->tag == tag) goto L1;
  return CacheMiss;
L1: return CacheHit;
```

In our design, the static compiler is separated from the stitcher through a simple interface language comprised of directives and the run-time constants table. An alternative would be to fold together the set-up code with the stitcher, with the set-up code directly invoking stitcher primitives at appropriate places or even generating instantiated machine code directly without copying templates, as is done in some other systems [CN96,LL96]. This approach would eliminate the need for directives and for the intermediate constants table computed by the current set-up code, and consequently would most likely produce significantly quicker dynamic compiles. Our current approach is a convenient intermediate point, since it is simple, flexible, and reasonably fast, and it side-steps difficult problems of planning out final stitcher activities for arbitrary control flow graphs prior to optimization.

5 Experimental Assessment

We embedded our static analysis in the Multiflow optimizing compiler [LFK$^+$93,FRN84] and dynamically compiled kernels from several application domains (Table 2). All programs, both the static and dynamic versions, were executed on a DEC Alpha 21064. Each program was executed with a variety of run-time constant configurations; we report results for two configurations for those programs whose speedups were sensitive to multiple factors. For example, execution times for sparse matrix multiplication depend on both the size of the matrix and the number of non-sparse elements per row. Speedups on scalar-matrix multiply, on the other hand, are relatively independent of the size of the matrix.

Our preliminary results show good asymptotic speedups over statically-compiled code, but, as yet, high dynamic compilation overhead, leading to high breakeven points.[*] As mentioned in the previous section, the overhead of dynamic compilation is due to our separation of set-up code from the stitcher, leading to extra intermediate data structures and stitcher directive interpreter costs. Merging these components into a single pass should drastically reduce our dynamic compilation costs without affecting our asymptotic speedups.

In some applications, most of the template code corresponds to array loads and stores, which limits speedup. If all references to an array are through run-time constant offsets, then some array elements can be allocated to registers by the stitcher. We have begun experimenting with a variation of Wall's register actions used in his link-time register allocator [Wal86]: the static compiler produces directives that indicate how to remove or modify instructions if a particular array element is stored in a register; the stitcher then executes these directives to eliminate loads, stores, and address arithmetic, after choosing registers for some number of array elements. We have obtained a speedup of 4.1 (as opposed to the current 1.7) on the calculator program using this technique.

[*] Asymptotic speedups were determined by comparing hardware cycle counter values for statically and dynamically compiled versions of each program's dynamic region. The breakeven point is the lowest number of executions of the dynamically-compiled code (including the overhead of executing set-up and stitcher code) at which the dynamic version is profitable.

Table 3: Optimizations Applied Dynamically

Benchmark	Constant Folding	Static Branch Elimination	Load Elimination	Dead Code Elimination	Complete Loop Unrolling	Strength Reduction
Calculator	√	√	√	√	√	
Scalar-matrix multiply			√			√
Sparse matrix-vector multiply	√		√		√	
Event dispatcher	√	√	√	√	√	
Record sorter	√	√	√	√	√	

Several optimizations, all applied dynamically, were responsible for the asymptotic speedups (Table 3). Although constant folding, load elimination, and complete loop unrolling were used most often, each optimization was important for some application.

6 Related Work

6.1 Partial-Evaluation-Based Dynamic Compilers

Most closely related to our work are other dynamic compilation systems that incorporate ideas from partial evaluation [SZ88,JGS93]. Partial evaluators enable a phased compilation strategy, where a program is compiled, given partial knowledge of its input, to produce a new faster program that takes the remaining input. Analogously, our static compiler compiles a dynamic region, given the partial knowledge that some of the variables at entry to the region will be invariant; the output of the static compiler is a subprogram whose compilation is completed by the stitcher. Off-line partial evaluators incorporate a binding time analysis that determines which variables depend only on the known inputs, much like our run-time constants identification analysis. Sophisticated partial evaluators handle partially-known data structures, as does our compiler. On the other hand, partial evaluators are usually source-to-source transformers for purely functional languages, whose analyses are expressed as abstract interpretations over the abstract syntax of the program; our system is an intermediate representation-to-optimized machine-code translator for general-purpose imperative languages, whose analyses operate over low-level control flow graphs. Partial evaluators operate interprocedurally (but over relatively small programs), often handle higher-order functions, and can produce multiple, specialized versions of procedures to maximize the flow of known information. Our compiler currently is only intraprocedural, but it can produce multiple compiled versions of a single dynamic region.

Both Leone and Lee [LL96] and Consel and Noël [CN96] use a partial evaluation-based framework to build dynamic compilers. Leone and Lee's system, called Fabius, applies dynamic compilation to a first-order, purely-functional subset of ML. The programmer uses explicit currying to indicate where dynamic compilation is to be applied. As each argument to a curried function is supplied, a new function that takes the remaining arguments is dynamically compiled, specialized to the run-time value of the first argument. An intraprocedural binding time analysis on the original function body identifies the calculations that depend only on the early argument values. The dynamic compilation step is fast, because the statically-generated code for a function contains the calculations that are based only on the first argument, interspersed with `emit` pseudo-instructions that generate the remaining code. However, the dynamically-generated code is not optimized across instructions. (Leone and Lee suggest extending their run-time code generator to perform register assignment at dynamic compile-time; however, this will slow dynamic compilation.) In contrast, our compiler targets a more general programming model and strives for both fast dynamic compilation and fast dynamic execution. Finally, Fabius is safe, n that the compileroptimizations do not affect program correctness; however, Fabius achieves safety by disallowing side-effects. The correctness of our transformations depends on the correctness of the programmer annotations.

Consel and Noël's system, developed concurrently with ours, follows a very similar structure. It too is a compiler for C programs that produces machine-code templates with holes that are instantiated at run-time. Their system pre-plans run-time constant propagation and folding, dead branch elimination, and loop unrolling, like ours. Some key differences in our approaches are the following:

- Their system follows more closely the traditional partial evaluation approach. Programmers annotate arguments of the top-level procedure to be dynamically compiled, global variables, and components of data structures as run-time constant. Their binding time analysis then interprocedurally identifies derived run-time constants. Our annotations currently apply only intraprocedurally, but our annotations offer more flexibility in treating a variable or data structure as constant in one context but variable in another.
- They do not describe their binding time analysis, other than to show that its output annotates syntax trees, and their remaining transformations are expressed as tree transformations. They do not analyze reachability conditions for constant branches. This suggests that they would have difficulty coping with the unstructured C programs that we handle.
- To produce machine-code templates, they generate C code containing special marker code sequences, compile it with a regular C compiler, and then post-process the assembly code to rediscover the markers and identify templates and holes. The post-processing tool is specific to a particular target machine and compiler, and relies on the compiler's optimizations not interfering with the marker structure. Our approach directly modifies an optimizing compiler to avoid such limitations, at some cost in implementation effort.
- They do not perform peephole optimizations at dynamic compile time, nor do they maintain a table of large constants for faster run-time access.
- In their system, the programmer is responsible for managing the code pointers that are returned from invoking a dynamically-compiled function. Our system takes care of this automatically, including managing a keyed collection of code pointers for different invocation contexts.
- To handle C pointers and support partially-constant data structures, they include an automatic pointer/alias analysis (which currently is not sound in the presence of C casts and pointer arithmetic), while we rely on programmer annotations. Although more susceptible to programmer errors, annotations can identify constants that are beyond the ability of current alias analyses. In addition, they do not do alias analysis of callers of the dynamically-compiled function, so they cannot automatically identify which formal parameters and global variables really are constant. They rely on the programmer to use the generated code pointer appropriately, analogous to our reliance on the correctness of the programmer assertions.

Guenter, Knoblock, and Ruf have developed a specialized compiler that applies partial evaluation-like techniques to a graphics rendering application [GKR95,KR96]. While not producing machine code at run-time, their system does analyze the rendering procedures to produce multiple, specialized versions for different combinations of constant arguments, and dynamically computes and caches the results of constant calculations in a data structure much like our run-time constants table. They observed speedups of up to 100 for their particular application.

6.2 Other General-Purpose Dynamic Compilers

Keppel, Eggers, and Henry [KEH93,Kep96] developed a library for manually constructing expression trees and then compiling them into callable machine code from within a program, in a portable fashion. They also developed a template-based approach. Their experiments demonstrated that these techniques outperformed the best statically-compiled, hand-tuned code in several applications. In a similar vein, Engler and Proebsting developed DCG [EP94], a library for constructing and manipulating expression trees that exploits the IBURG portable code generator library [Pro92]. The code generator infrastructure performed no optimizations other than instruction selection. Engler, Hsieh, and Kaashoek developed `C [EHK96], an extension of the C language that makes constructing and manipulating expression trees look syntactically like fragments of C code, greatly easing the programming burden. DCG is used as the back-end infrastructure. More recently, Poletto, Engler, and Kaashoek have retargeted `C to use a template-based back-end [PEK96].

Compared to our approach, these manual approaches offer more flexibility of optimization (since the programmer is responsible for performing all global optimizations by hand), but at the cost of longer dynamic compilation times (with the exception of template-based `C) and more tedious and error-prone programming work.

6.3 Other Dynamic Compilation Systems

A number of previous systems have exploited dynamic compilation for run-time performance or flexibility gains, for example, in graphics displaying [PLR85], operating system operations [PAAB$^+$95,PMI88], and object-oriented systems [DS84,CU89,HU94]. However, these systems did not make dynamic compilation available to the programmer in more general scenarios.

7 Conclusions

We have designed and built a dynamic compilation framework for general-purpose imperative languages like C whose twin goals are high-quality dynamically-compiled code and low run-time compilation overhead. Several factors contribute to the quality of the dynamically-compiled code: optimizing dynamic regions within the context of their enclosing procedure, planning out optimizations that depend on run-time constants (including the capability to analyze unstructured control flow), segregating the set-up code that applies these optimizations at run-time from the repeatedly-executed templates, and embedding this entire analysis within an optimizing static compiler. Dynamic compilation overhead is reduced by presenting the dynamic compiler with almost completely constructed machine code. Initial speedups over a set of statically-compiled C programs range from 1.2 to 1.8.

We plan to extend our framework in several dimensions: to provide run-time constants and reachability analyses on the interprocedural level, to more fully automate the selection of run-time constants and dynamic regions, to merge set-up code with stitching for faster dynamic compilation, to provide dynamic compilation support for other input languages, and to extend our benchmark suite to other application areas and larger programs.

Acknowledgments

We would like to thank John O’Donnell (Equator Technologies, Inc.) and Tryggve Fossum (Digital Equipment Corp.) for the source for the Alpha AXP version of the Multiflow compiler; Ben Cutler (Equator Technologies, Inc.), Michael Adler, and Geoff Lowney (Digital Equipment Corp.) for technical advice in altering it; Marc Friedman for work on the design and implementation of an early version of our framework; Charles Consel, Wilson Hsieh, Jack Lo, and the PLDI referees for their helpful comments on the submitted version of the paper; Jeffrey Dean and David Grove for discussions on the run-time constant identification and reachability analyses; and David Keppel, Markus Mock, Scott van Woudenberg, and in particular Brian Grant for their help with the applications. The research was supported by ARPA contract N00014-94-1-1136, NSF PYI Award MIP-9058439, NSF NYI Award CCR94-57767, and ONR grant N00014-92-J-1395.

References

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[AWZ88] B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting equality of variables in programs. In *Symposium on Principles of Programming Languages*, January 1988.

[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.

[CC95] C. Click and K.D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2), March 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.

[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*, January 1989.

[CN96] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Symposium on Principles of Programming Languages*, January 1996.

[CU89] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *Conference on Programming Language Design and Implementation*, July 1989.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Symposium on Principles of Programming Languages*, January 1984.

[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.

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

[HU94] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Conference on Programming Language Design and Implementation*, June 1994.

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

[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.

[Per90] A.J. Perlis. Epigrams on programming. In *Communications of the ACM*, 1990.

[PLR85] R. Pike, B.N. Locanthi, and J.F. Reiser. Hardware/software trade-offs for bitmap graphics on the Blit. *Software - Practice and Experience*, 15(2), 1985.

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

[Pro92] T.A. Proebsting. Simple and efficient BURS table generation. In *Conference on Programming Language Design and Implementation*, July 1992.

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

[Wal86] D.W. Wall. Global register allocation at link time. In *Symposium on Compiler Construction*, June 1986.

Appendix A Specifications of Analyses

We use a kind of abstract interpretation-style, lattice-theoretic specification for our dataflow analyses.

A.1 Run-Time Constant Identification Analysis

The domains and operations for instructions, variables, and constant variables:

$\mathtt{Inst}$ = *set of straight-line instructions in dynamic region*

$\mathtt{BranchInst}$ = *set of branch instructions (both* `if` *and* `switch` *branches) in region*

$\mathtt{Point}$ = *set of program points (flow arcs between instructions) in dynamic region*

$\mathrm{successors}\colon \mathtt{Inst} + \mathtt{BranchInst} \rightarrow 2^{\mathtt{Point}}$

$\mathrm{successors}(\mathrm{i})$ = *set of points after* i

$\mathtt{Var}$ = *set of variables in dynamic region*

$\mathtt{Constants} = 2^{\mathtt{Var}}$*; an element of this domain is the set of variables known to be constant at a program point;* $\subseteq$ *is the lattice* $\le$ *ordering operator for this domain*

C denotes the solution to the run-time constant analysis:

$C\colon \mathtt{Point} \rightarrow \mathtt{Constants}$

$C(\mathrm{p})$ = *greatest fixed point solution (largest set of run-time constants) to following dataflow equations at point* p

The interpretation of our analysis is $\forall \mathrm{p} \in \mathtt{Point}$, if $\mathrm{v} \in C(\mathrm{p})$, then v is defined to the same value each time it is executed at run-time, assuming the programmer annotations are correct.

The initial set of constants at the start of the dynamic region:

$C(\mathtt{p}_0)$ = *set of variables labeled as constants by programmer annotation at start of region*

C_{flow} is the flow function for straight-line instructions, computing the set of constants at the point after an instruction from the set of constants at the point before the instruction:

$C_{flow}\colon \mathtt{Inst} \rightarrow \mathtt{Constants} \rightarrow \mathtt{Constants}$

$C_{flow}\ [\![\mathtt{x := k}]\!]\ \mathtt{cs} = \mathtt{cs} \cup \{\mathtt{x}\}$, *where* k *is a compile-time constant*

$C_{flow}\ [\![\mathtt{x := y\ op\ z}]\!]\ \mathtt{cs} =$
$\mathtt{cs} \cup \{\mathtt{x}\}$, *if* $\{\mathtt{y,z}\} \subseteq \mathtt{cs}$ *and* $\mathtt{op}$ *is idempotent and side-effect-free,*
$\mathtt{cs} - \{\mathtt{x}\}$, *otherwise*

$C_{flow}\ [\![\mathtt{x := f(y_1,...,y_n)}]\!]\ \mathtt{cs} =$
$\mathtt{cs} \cup \{\mathtt{x}\}$, *if* $\{\mathtt{y_1,...,y_n}\} \subseteq \mathtt{cs}$ *and*
$\mathtt{f}$ *is idempotent and side-effect-free,*
$\mathtt{cs} - \{\mathtt{x}\}$, *otherwise*

$C_{flow}\ [\![\mathtt{x := *y}]\!]\ \mathtt{cs} =$
$\mathtt{cs} \cup \{\mathtt{x}\}$, *if* $\{\mathtt{y}\} \subseteq \mathtt{cs}$,
$\mathtt{cs} - \{\mathtt{x}\}$, *otherwise*

$C_{flow}\ [\![\mathtt{x :=}\ \mathbf{dynamic*}\ \mathtt{y}]\!]\ \mathtt{cs} = \mathtt{cs} - \{\mathtt{x}\}$

$C_{flow}\ [\![\mathtt{*x := y}]\!]\ \mathtt{cs= cs}$

$C_{flow}\ [\![\mathtt{x :=}\ \phi\mathtt{(x_1,...,x_n)}]\!]\ \mathtt{cs} =$
$\mathtt{cs} - \{\mathtt{x_1,...,x_n}\} \cup \{\mathtt{x}\}$, *if* $\{\mathtt{x_1,...,x_n}\} \subseteq \mathtt{cs}$,
$\mathtt{cs} - \{\mathtt{x,x_1,...,x_n}\}$, *otherwise*

C_{branch} is the flow function for branch nodes. Given a set of constants before the branch, it computes a mapping from successor points to the constant sets at those points, i.e., branches have no effect on the set of computed constants:

$C_{branch}\colon \mathtt{BranchInst} \rightarrow \mathtt{Constants} \rightarrow (\mathtt{Point} \rightarrow \mathtt{Constants})$

$C_{branch}\ \mathtt{b\ cs} = \{(\mathtt{s,cs}) \mid \mathrm{s} \in \mathrm{successors(b)}\}$

The lattice meet function computes the set of constants after a merge (this depends on the solution R to the reachability analysis at the merge):

$\sqcap_{\mathtt{Constants}}\colon \mathtt{Constants} \times \mathtt{Constants} \rightarrow \mathtt{Constants}$

$\sqcap_{\mathtt{Constants}}\ (\mathtt{cs_1, cs_2}) =$
$\mathtt{cs_1} \cup \mathtt{cs_2}$, *if* $\mathrm{exclusive}(\mathtt{cn_1, cn_2})$,
where $\mathtt{cn}_i = R(\mathrm{p}_i)$ *for merge predecessor* p_i,
$\mathtt{cs_1} \cap \mathtt{cs_2}$, *otherwise*

(The meet function and the ϕ functions should be coordinated. The ϕ functions should be part of the merge itself rather than located after the merge, so that none of the values defined only along a subset of the predecessor branches survive to the point after the merge.)

A.2 Reachability Conditions Analysis

The following additional domain supports the reachability analysis. We define the values of this domain using a grammar rule (which we assume is always simplified to conjunctive normal form):

$\mathtt{Condition ::=}$
$\quad \mathrm{B} \rightarrow \mathrm{S}$, *where* $\mathrm{B} \in \mathtt{BranchInst}$ *and* $\mathrm{S} \in \mathrm{successors(B)}$
$\quad \mid \mathtt{Condition} \wedge \mathtt{Condition}$
$\quad \mid \mathtt{Condition} \vee \mathtt{Condition}$
$\quad \mid \mathrm{true}$

Two conditions can be mutually exclusive:

$\mathrm{exclusive}(\mathtt{cn_1, cn_2}) = (\mathtt{cn_1} \Rightarrow \neg\, \mathtt{cn_2}) \wedge (\mathtt{cn_2} \Rightarrow \neg\, \mathtt{cn_1})$,
where $\neg\, (\mathrm{B} \rightarrow \mathrm{S_1}) = \bigvee_{\mathrm{S} \in \mathrm{successors(B)},\ \mathrm{S} \neq \mathrm{S1}} (\mathrm{B} \rightarrow \mathrm{S})$.

This rule implies that $\mathrm{B} \rightarrow \mathrm{S_1}$ and $\mathrm{B} \rightarrow \mathrm{S_2}$ are mutually exclusive iff $\mathrm{S_1} \neq \mathrm{S_2}$.

Reverse logical implication is the lattice $\le$ ordering operator for this domain, i.e., if $\mathtt{cn_1} \Rightarrow \mathtt{cn_2}$, then $\mathtt{cn_2} \le \mathtt{cn_1}$.

$\mathrm{B} \rightarrow \mathrm{S_1}$ and $\mathrm{B} \rightarrow \mathrm{S_2}$ are mutually exclusive if $\mathrm{S_1} \neq \mathrm{S_2}$.

R denotes the solution to the reachability analysis equations:

$R\colon \mathtt{Point} \rightarrow \mathtt{Condition}$

$R(\mathrm{p})$ = *greatest fixed point solution (most constrained set of branch outcomes) to following dataflow equations at point* p

The interpretation of our analysis is $\forall \mathrm{p} \in \mathtt{Point}$, $R(\mathrm{p}) \Rightarrow R_{actual}(\mathrm{p})$, where $R_{actual}(\mathrm{p})$ represents the actual branch outcomes at run-time: $R_{actual}(\mathrm{p}) = \bigwedge_{\mathrm{B} \in \mathtt{BranchInst}} (\bigvee_{\mathrm{S} \in \text{taken-successors(B)}} (\mathrm{B} \rightarrow \mathrm{S}))$, where taken-successors(B) are those successors of B that are taken at run-time.

The initial set of reachability conditions at start of dynamic region:

$R\ \mathtt{p_0} = \top_{\mathtt{Condition}} = \mathrm{true}$

R_{flow} is the flow function for straight-line instructions, which have no effect on the reachability conditions:

$R_{flow}\colon \mathtt{Inst} \rightarrow \mathtt{Condition} \rightarrow \mathtt{Condition}$

$R_{flow}\ \mathtt{i\ cn = cn}$

R_{branch} is the flow function for branch nodes. Branches with run-time constant predicates restrict the conditions along successor branches, while non-constant branches have no effect on the reachability analysis:

$R_{branch}\colon \mathtt{BranchInst} \rightarrow \mathtt{Condition} \rightarrow (\mathtt{Point} \rightarrow \mathtt{Condition})$

$R_{branch}\ \mathtt{b\ cn} =$
$\{(\mathtt{s,cn} \wedge \mathrm{b} \rightarrow \mathtt{s}) \mid \mathrm{s} \in \mathrm{successors(b)}\}$,
if $\mathrm{b} = [\![\mathrm{q?}]\!]$ *and* $\mathrm{q} \in C(\mathrm{before(b)})$
$\{(\mathtt{s,cn}) \mid \mathrm{s} \in \mathrm{successors(b)}\}$, *otherwise*

The meet function, computing reachability conditions after a merge, is simple disjunction:

$\sqcap_{\mathtt{Reach}}\colon \mathtt{Condition} \times \mathtt{Condition} \rightarrow \mathtt{Condition}$

$\sqcap_{\mathtt{Reach}}\ (\mathtt{cn_1, cn_2}) = \mathtt{cn_1} \vee \mathtt{cn_2}$