

The Benefits and Costs of DyC's Run-Time Optimizations

BRIAN GRANT, MARKUS MOCK, MATTHAI PHILIPOSE,
CRAIG CHAMBERS, and SUSAN J. EGGERS

University of Washington

DyC selectively dynamically compiles programs during their execution, utilizing the run-time-computed values of variables and data structures to apply optimizations that are based on partial evaluation. The dynamic optimizations are preplanned at static compile time in order to reduce their run-time cost; we call this *staging*. DyC's staged optimizations include (1) an advanced binding-time analysis that supports polyvariant specialization (enabling both single-way and multiway complete loop unrolling), polyvariant division, static loads, and static calls, (2) low-cost, dynamic versions of traditional global optimizations, such as zero and copy propagation and dead-assignment elimination, and (3) dynamic peephole optimizations, such as strength reduction. Because of this large suite of optimizations and its low dynamic compilation overhead, DyC achieves good performance improvements on programs that are larger and more complex than the kernels previously targeted by other dynamic compilation systems. This paper evaluates the benefits and costs of applying DyC's optimizations. We assess their impact on the performance of a variety of small to medium-sized programs, both for the regions of code that are actually transformed and for the entire application as a whole. Our study includes an analysis of the contribution to performance of individual optimizations, the performance effect of changing the applications' inputs, and a detailed accounting of dynamic compilation costs.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers; optimization*

General Terms: Performance

Additional Key Words and Phrases: Dynamic compilation, specialization

1. INTRODUCTION

*Dynamic compilation*¹ transforms parts of programs at run time, using information available only at run time, to optimize them more fully than strictly statically compiled code. *Value-specific*, dynamic compilers derive their benefits

This work was supported by ONR contract N00014-96-1-0402, ARPA contract N00014-94-1-1136, NSF Grant EIA-9975057, NSF Young Investigator Award CCR-9457767, and an Intel Graduate Fellowship.

Authors' addresses: Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2530; email: {grant; mock; matthai; chambers; eggers}@cs.washington.edu.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

© 2001 ACM 0164-0925/00/0900-0932 \$5.00

¹We are referring to systems that dynamically optimize (or specialize) portions of programs and stage their analysis across static-compile and execution times. Just-in-time compilers (JITs) for Java do all compilation of the entire program at run time.

by optimizing parts of programs for particular run-time-computed values of invariant variables and data structures (called *run-time constants*), in effect performing a kind of dynamic constant propagation and folding. Proposed applications for value-specific, dynamic compilation include specializing architectural simulators for the configuration being simulated, language interpreters for the program being interpreted, rendering engines for scene-specific state variables, numeric programs for dimensions and values of frequently used arrays, and critical paths in operating systems for the type of data being processed and the current state of the system. Trends in software engineering toward dynamic reconfigurability and parameterization for reuse also imply a promising role for value-specific, dynamic compilation.

Recent research efforts have made considerable progress toward proving the viability of value-specific, dynamic compilation. In particular, researchers have demonstrated that dynamic compilation overhead can be quickly amortized by the increased efficiency of the dynamically optimized code. Most experiments, however, were confined to simple kernels and did not demonstrate that the dynamic compilation systems could cope reasonably with the increased size and complexity of applications like the interpreters and simulators mentioned above.

The reason current systems have not made better progress on larger, more complex applications varies, depending on the approach taken. In *imperative* systems, such as `C` [Eggers et al. 1997; Poletto et al. 1997; 1999], a programmer explicitly constructs, composes, and compiles code fragments at run time. Imperative approaches can express a wide range of optimizations, but they impose a large burden on the programmer to manually program the optimizations; the programming burden makes it difficult to apply imperative approaches effectively to larger applications. Other systems, such as Tempo [Consel and Noël 1996; Noël et al. 1998], Fabius [Leone and Lee 1995], and our earlier, prototype system [Auslander et al. 1996], follow a *declarative* approach, where sparse user annotations trigger analyses and transformations of the program (using partial-evaluation-style techniques) to exploit value-specific dynamic compilation. To reduce dynamic compilation costs, these systems preplan the possible effects of dynamic optimizations statically, producing a specialized dynamic compiler that is tuned to the particular part of the program being dynamically optimized; this sort of preplanning we call *staging* the optimization. Declarative approaches are easier for programmers to use, but are only as powerful as the analyses and optimizations they apply. The limitations of previous declarative systems prevented them from coping effectively with the more involved patterns of control and data flow found in some small and most large applications, causing them to miss key optimization opportunities or forcing substantial rewriting of the application to fit the limitations of the system.

DyC (pronounced *dicey*) [Grant et al. 1997; 1999] is a value-specific, dynamic compilation system that has good potential for producing speedups on larger, more complex C programs. To reduce the programming burden, DyC is a declarative, annotation-based system. To support effective optimization, DyC contains an advanced binding-time analysis, including program-point-specific

polyvariant division and specialization,² and dynamic versions of traditional global and peephole optimizations. To keep dynamic compilation costs low, all of DyC's dynamic optimizations are staged, with the bulk of the work of the optimization occurring at static compile time and with no run-time program representation or iterative analyses required. DyC automatically caches the dynamically compiled code and reuses it where possible, relieving another programmer burden and reducing dynamic compilation overhead. Programmers can also declaratively specify *policies* that govern the aggressiveness of specialization and caching, enabling them to get finer control over the dynamic compilation process, while preserving the declarative model. (We are currently developing a tool that automatically decides where to apply value-specific, dynamic optimizations and policies, and generates the annotations that DyC then carries out [Mock et al. 1999].)

This paper assesses the benefits, costs and applicability of DyC's optimizations, both individually and when applied together, and explains why they achieved the performance improvements they did. The optimizations are evaluated on a selection of small to medium-sized, widely used applications that are representative of the application classes mentioned earlier. Our study includes a performance evaluation of both the portions of the programs that are dynamically optimized and the applications as a whole, an analysis of the contribution to performance of individual optimizations, an examination of the sensitivity of the optimizations to the particular inputs used and a detailed accounting of dynamic compilation overhead. The results show that:

- DyC produces good speedups on dynamically optimized code; the range of performance improvement was large, extending from 20% to a factor of six.
- Whole-program performance (the performance of the dynamically optimized code executing within the context of its otherwise statically compiled application) also increased, showing that DyC produces speedups on real applications (two simulators, an interpreter, a graphics program, and a numerical program), not simply kernels.
- All of DyC's optimizations were important to obtaining the speedups. Furthermore, several optimizations that are unique to DyC, such as dynamic dead-assignment elimination, were responsible for substantial speedups (3×–5×) in some codes. Two optimizations, complete loop unrolling and static loads, were essential to speedups in nearly all of our benchmarks.
- Each of DyC's optimizations improved performance not only because of its direct benefits, but also because it created opportunities to apply other dynamic optimizations. Moreover, optimizations were interdependent to the extent that disabling any one optimization often prevented program speedups from the others.
- Many of DyC's run-time optimizations were sensitive to particular program inputs. Consequently, widespread usage of dynamic optimization will benefit

²Polyvariant division allows the same piece of code to be analyzed with different combinations of variables being treated as run-time constants; each combination is called a *division*. Polyvariant specialization allows multiple compiled versions of a division to be produced, each specialized for different values of the run-time-constant variables. Program-point-specific polyvariance enables polyvariance to arise at arbitrary points in programs, not just at function entries.

from run-time and/or profile-driven static evaluation of their costs and benefits, and control over which optimizations to apply, if any.

- To ensure low overhead, all parts of the dynamic compilation process must be fast, because different dynamic optimizations stress different parts of the dynamic compiler.
- DyC's dynamic compilation cost is low enough that the break-even point at which dynamic compilation becomes profitable is well within the normal usage of our applications.

The next section describes the DyC dynamic compilation system and its optimizations, and presents an example similar to one of our benchmarks. Section 3 details our experimental methodology and workload. Section 4, the heart of the paper, contains our analysis of the benefits and costs of DyC's run-time optimizations. Section 5 continues the discussion of performance results with an analysis of the performance effect of different inputs on dynamic optimizations. Finally, Section 6 compares DyC to related research, and Section 7 concludes.

2. DYC AND ITS OPTIMIZATIONS

DyC's approach to dynamic compilation is based on techniques derived from the partial-evaluation field but adapted and extended to the needs of dynamically compiling C programs. DyC's ability to produce efficient dynamic code also depends on special, staged versions of traditional global optimizations, such as zero and copy propagation and dead-assignment elimination, and peephole optimizations, such as strength reduction.

DyC compiles and optimizes selected parts of programs dynamically, during their execution. To trigger dynamic compilation, programmers annotate their source code to identify *static variables* (variables that have a single value or relatively few values during program execution) on which many calculations depend; static variables are run-time constants. DyC then automatically determines which parts of the program downstream of the annotations can be optimized based on the static variables' values (we call these *dynamically compiled regions* or just *dynamic regions*) and arranges for each dynamic region to be compiled at run time, once the values of the static variables are known. To minimize dynamic compilation overhead, DyC stages its dynamic optimizations, performing much of the analysis and planning for dynamic compilation and optimization during static compile time.

This preplanning of dynamic optimization is performed within a traditional (static) optimizing compiler, because doing so allows DyC to use the existing compiler infrastructure to statically manipulate, analyze, and (partially) optimize dynamic regions. Specifically, we chose the Multiflow trace-scheduling compiler [Lowney et al. 1993] to host our static analyses and transformations, because it generated good-quality code for our target architecture (the statically scheduled Alpha 21164). DyC extends the Multiflow compiler with two major components, as illustrated in Figure 1:

- As in offline partial-evaluation systems [Jones et al. 1993], a *binding-time analysis* (BTA) identifies those variables (called *derived static variables*) whose values are computed solely from annotated or other derived static variables; the lifetimes of these static variables determine the extent of the

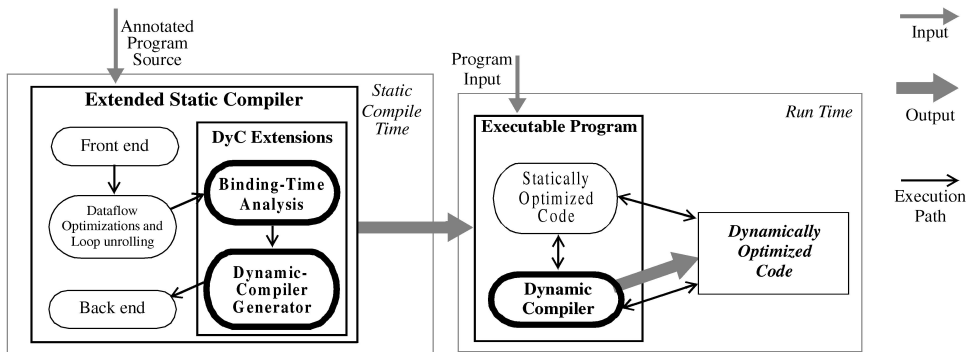


Fig. 1. DyC's static and dynamic components. This figure diagrams DyC's extensions to a static optimizing compiler (DyC's components are circled in **bold**), and illustrates the creation and use of a custom dynamic compiler for an application.

dynamic region. The BTA divides operations within a dynamic region into those that depend solely on static variables and therefore can be executed only once (the *static computations*) and those that depend at least in part on run-time data and must be reexecuted each time the flow of execution reaches them (the *dynamic computations*). The static computations correspond to those computations that will be constant-folded at run time. The BTA is the static component of staged dynamic constant propagation and folding; it is an aggressive analysis, in that it creates multiple versions of the same piece of code (e.g., by splitting control-flow merges) in order to continue propagating constant values, and it folds operations such as loads and calls, in addition to arithmetic operations and branches.

- For each dynamic region, a *dynamic-compiler generator* produces a customized³ dynamic compiler (also called a *generating extension* [Jones et al. 1993]) that will generate code at run time for that region, given the values of its static variables.

In more detail, DyC/Multiflow performs the following steps when compiling each procedure at static compile time:

- First, Multiflow applies many traditional intraprocedural optimizations, stopping just prior to register allocation and code scheduling. The optimizations performed include virtual-register assignment, constant propagation and folding, copy propagation, common-subexpression elimination, dead-code elimination, if conversion (to conditional moves), loop-invariant code motion, induction-variable simplification, strength reduction, height reduction (for n -ary-operand operations and simple recurrences), loop peeling and unrolling, and (directive-controlled) function inlining [Lowney et al. 1993]. DyC's analyses and transformations follow these optimizations, because our transformations would otherwise interfere with them.⁴

³We use the word *customized* rather than *specialized* to reduce confusion between the specialization of the dynamic region at run time (using the values of the annotated variables) and specialization of the dynamic compiler at static compile time (using the identities of the annotated variables).

⁴Multiflow's optimizations also interfere with DyC's analyses, mainly by obscuring the intended meaning of the annotations, so some modifications to them were required to preserve information [Grant et al. 2000].

- Then, for procedures that contain annotations, the BTA identifies derived static variables, the static computations,⁵ and the boundaries of dynamic regions. It also determines which loops have static induction variables and should be completely unrolled at dynamic compile time. The optimizations supported by DyC's BTA are described in Section 2.1.
- Each dynamic region is replaced with two control-flow subgraphs, the *set-up code*, which contains the static computations, and the *template code*, which contains the dynamic computations. The set-up code forms the basis of the customized dynamic compiler, whereas the template code serves as a template for the dynamically specialized code. Where a dynamic instruction in the template code refers to a static operand, a place-holder operand called a *hole* is used. Holes are filled in at run time by the dynamic compiler, using results of the static computations. Both subgraphs are anchored by (essentially) a switch at the program point where the dynamic compiler is invoked.
- Multiflow's combined register allocation and code scheduling are then applied to the procedure's modified control-flow graph. By separating the set-up and template subgraphs, register allocation and scheduling can be applied to each independently, without one interfering with the other. By keeping these two subgraphs in the context of the rest of the procedure's control-flow graph, any variables live both inside and outside a dynamic region can be allocated registers seamlessly across dynamic-region boundaries. DyC relies on this static scheduling of the template code for creating good-quality dynamically generated code and performs no instruction scheduling during dynamic compilation.
- To preplan dynamic zero and copy propagation and dead-assignment elimination, DyC performs abstract versions of these analyses on the template subgraph, identifies potential analysis and transformation actions to be performed at run time, and inserts code sequences that implement these actions into the set-up code. These optimizations are described in more detail in Section 2.2.
- Finally, the custom dynamic compiler for each dynamic region is completed by assembling the instructions from the template code and then inserting *emit* code sequences [Leone and Lee 1995] into corresponding locations in the set-up code. (The emit code sequences construct and store the resulting executable operations into memory at run time.) The template subgraph is then thrown away. The customized dynamic compiler is fast, in large part, because it neither consults an intermediate representation nor performs any general analysis when run. Instead, these functions are in effect "hard-wired" into the custom compiler for that region, represented by the set-up code and its embedded emit code. Section 4.3.1 describes the custom dynamic compilers in more detail.

At run time, a dynamic region's custom dynamic compiler is invoked to generate the region's code. The dynamic compiler first checks an internal cache of previously dynamically generated code for a version that was compiled for

⁵Note that conditional branches and switches that test static variables are static computations and can be folded at dynamic compile time.

the current values of the annotated variables. If one is found, it is reused. Otherwise, the dynamic compiler creates a new version by evaluating the static computations and emitting machine code for the dynamic computations, patching holes and branches, and evaluating conditions under which peephole and staged optimizations can be applied. When it is done, it saves the newly generated machine code in the dynamic-code cache. Invoking the dynamic compiler and dispatching to dynamically generated code (that is found in DyC's cache) are the principal sources of run-time overhead.

2.1 Optimizations Adapted from Partial Evaluation

DyC's binding-time analysis, and those of other declarative dynamic compilers, identifies which variables are static over which paths of the procedure's control-flow graph. By distinguishing static computations from dynamic computations at static compile time, the BTA enables run-time constant propagation without incurring any run-time cost from analyzing an intermediate representation. The BTA starts its analysis with the annotations that identify static variables and ends after the last use of any static value. To effectively analyze C programs, DyC's BTA is program-point-specific and flow-sensitive: a dynamic region can start and stop at any program point (a dynamic region may have multiple exits), and a variable may be static at some program points but dynamic at others. DyC's BTA supports polyvariant specialization (which enables complete loop unrolling), polyvariant division, static loads, and static calls.

2.1.1 Polyvariant Specialization. Dynamic compilation generates code that is specialized to particular values of static variables. Where dynamic specialization on the run-time computed values of some variables should begin, such as at the entry to a dynamic region, the variables are said to be *promoted* from dynamic to static. If these promoted variables take on different values at different entry times, DyC produces multiple versions of the code after the promotion point, each specialized for a different combination of promoted values; this is called *polyvariant specialization*. A dynamic-code cache lookup implements the promotion. In DyC, the programmer can control the cost of the dynamic-code cache lookups using a policy annotation, described in Section 4.3.2.

In DyC, dynamic-to-static promotions can also occur at arbitrary program points in the middle of a dynamic region. These *internal promotions* enable a kind of *multistage specialization* [Leone and Lee 1998]. For example, an internal promotion can occur at the point where an annotated variable is assigned a dynamic value, which allows specialization to continue using the newly computed value (at the cost of a cache check for code that uses the promoted variable). Internal promotions also allow code to be increasingly specialized on a growing set of static variables as execution proceeds through a dynamic region.

Polyvariant specialization can result in *complete loop unrolling* by creating a specialized copy of a loop body for each distinct combination of values of the loop induction variables. Complete loop unrolling is unlike unrolling done by traditional static compilers in that the unrolled loop is eliminated rather than enlarged. For simple loops, such as those that merely increment a counter until an exit condition is reached, a linear chain of unrolled loop bodies results (we call this *single-way loop unrolling*). For more complex loops, however, one iteration may lead to several alternative loop iterations (e.g., if the loop contains branch

paths that update the loop induction variables differently), or even return to a previously generated loop iteration, producing, in general, a directed graph of unrolled loop bodies (which we call *multiway loop unrolling*).

2.1.2 Polyvariant Division. Polyvariant division allows the same program point to be analyzed multiple times, each time with a different set of variables assumed to be static. After binding-time analysis, each division gives rise to a separate version of the code, since each has its own partitioning into static and dynamic computations. Without polyvariant division, programmers would have to duplicate code by hand for the different divisions or adopt some least-common-denominator set of annotations with fewer optimization opportunities.

Polyvariant division can be used to implement *conditional specialization*, an optimization that enables DyC to decide at run time whether to dynamically optimize a dynamic region. Rather than unconditionally executing an annotation, the programmer guards it with a test of whether specialization is desirable. Polyvariant division then automatically duplicates the code following the test statement, one copy being specialized and the other not. Conditional specialization can be used, for example, to limit specialization to those values of the static variables that are particularly amenable to optimization (e.g., values that enable strength reduction or copy propagation), to those values that occur frequently enough to merit the effort of dynamic compilation, or to those loops that, when completely unrolled, will fit in the L1 instruction cache. While we did not study the use of conditional specialization for this paper, several types of cost-benefit checks, including the types mentioned, would be appropriate to the benchmarks we used.

2.1.3 Static Loads and Static Calls. By default, DyC assumes that the contents of data structures, even if referenced through a run-time-constant or compile-time-constant address, are dynamic. In many programs, however, at least some of these contents are invariant. In these cases, we wish to treat loads of invariant parts of static structures as static computations, done once as part of dynamic compilation. DyC allows programmers to annotate such loads as static, enabling them to be optimized in this way. (An alternative scheme would annotate *declarations* of array, structure, or pointer values or types as having static components, implying that all loads of those components were static.)

Similarly, users can annotate pure functions as static. Invocations of static functions with all static arguments are treated as static computations and hence are executed once as part of dynamic compilation. Calls to unannotated functions, even with all static arguments, are conservatively treated as dynamic computations, since they may have side effects.⁶

These annotations are potentially unsafe programmer assertions. The wide use of these annotations in our benchmarks (see Table IV) suggests that a safe, automatic implementation should be included in value-specific dynamic compilation systems. One possible such implementation would have the static compiler perform an automatic alias and side-effect analysis to identify static portions of data structures and pure functions. Tempo does this analysis within modules, but still relies on potentially unsafe user annotations to discover

⁶Tempo includes an additional feature where a function can be classified as being dynamic (i.e., having side effects), but still return a static value if all its arguments are static [Hornof et al. 1997].

the alias and side-effect properties of external data structures and procedures [Consel and Noël 1996; Noël et al. 1998].

2.2 Staged Versions of Traditional Optimizations

DyC also includes novel staged versions of dynamic zero and copy propagation and dead-assignment elimination that depend on the values of static variables. For example, if the single static operand to a multiply turns out to be 1 at dynamic compile time, then the multiply can be replaced by a simple move. Moreover, eligible downstream references to the target of the move can be replaced with the operand of the move (performing copy propagation), and if all references are so replaced, the move instruction can be eliminated (performing dead-assignment elimination). On some architectures, such as the DEC Alpha 21164 on which we performed our experiments, a floating-point move takes the same time as a floating-point multiply, so strength reduction of the multiply into a move alone yields no benefit; copy propagation and dead-assignment elimination are necessary to see performance improvements. Similarly, if the static operand to the multiply is 0, then the multiply can be replaced with a clear instruction, the 0 can be propagated to eligible downstream uses of the result of the clear instruction, and, if all uses are replaced, the clear can be eliminated. Moreover, replacing a multiply with a clear causes the use of the dynamic operand to be eliminated, potentially causing its computation to become dead as well. But copy propagation and dead-assignment elimination cannot be performed entirely statically (unlike the constant propagation embodied by binding-time analysis), since, if the run-time check of the operand of the multiply does not yield a 0 or 1, neither can be applied.

To perform data-dependent zero and copy propagation and dead-assignment elimination with low run-time cost, DyC divides their analyses into a planning stage done at static compile time and a completion stage done during dynamic compilation. The static planning stage computes whether an operation *potentially* may be replaced with a move or clear instruction. For each such instruction, all downstream uses of the result within a single *specialization unit* (a generalization of a basic block that has a single entry but multiple exits [Grant et al. 2000]) are identified statically. The emit code sequences for potentially optimizable instructions check for the special run-time-constant operand values that enable optimization; if one occurs, the instruction is simplified or deleted. For zero and copy propagation, replacements of registers by zeroes, ones, or other registers are recorded in a table that is maintained during dynamic compilation. This table contains one entry for every assignment within a specialization unit that might potentially yield a propagatable result and is reused across units. Emit code sequences for uses of the potentially optimized instruction check this table to see how they should generate code for their operand. Dynamic compilation time for run-time zero and copy propagation and dead-assignment elimination is kept low by forgoing any run-time intermediate representation or analysis other than the table that records the optimizations' effect on registers. However, the current implementation restriction on optimization within a specialization unit does prevent exploitation of some optimization opportunities, for example, those that must cross iterations of completely unrolled loops.

```

/* Perform the dot product of two floating-point
   vectors u and v */
double fpdotproduct(
    double u[], double v[], int length
)
{
    double x, y, product, sum;
    int i;
    make_static(u, length, i);
    sum = 0.0;
    for (i = 0; i < length; ++i) {
        x = u@[i];
        y = v[i];
        product = x * y;
        sum = sum + product;
    }
    return sum;
}

```

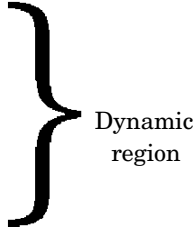


Fig. 2. Annotated dot-product procedure. This dot-product routine is annotated so that DyC will dynamically optimize it using the values of one of its input vectors (u), assuming that the routine will be invoked enough times with the same vector to recoup the dynamic compilation cost. DyC's `make_static` and `@` annotations are shown in **bold**. The resulting static operations and variables are shown in *italics*.

The emit code sequences also perform strength reduction of integer multiply, divide, and modulus operations with a single static operand. The operations that could be potentially transformed are identified at static compile time, so these peephole optimizations only incur cost at dynamic compilation time for operations to which they could be applied. Multiplies are replaced with shifts and adds using Booth's algorithm. We currently emulate dynamic strength reduction of divide and modulus operations by inserting special-case code in the program source.

2.3 Example

The example in Figure 2 illustrates some of DyC's capabilities and shows how the annotation interface is used. It is a floating-point version of the `dotproduct` kernel from our benchmark suite and is also very similar to (but simpler than) the dynamic region of the image-transformation benchmark program `pnmconvol`. The `fpdotproduct` routine takes as input vectors u and v of length `length`, and computes and returns the dot product of the two vectors (i.e., the sum of the pair-wise products of their elements). In the common case that the dot product is invoked repeatedly with one of the vectors the same, for example as a subroutine of a matrix multiplication or convolution routine, it can be profitable to specialize the dot product for the values of the invariant vector, say u .

Two DyC annotations, `make_static` and `@`, were inserted to accomplish this dynamic specialization. A `make_static` annotation on a variable specifies to DyC that the code that follows should be specialized (polyvariantly) for all distinct values of that variable. The `make_static` annotation in our example indicates that code that uses the pointer u and its length `length` should be

```

emit ( `sum = 0.0;` );
for (i = 0; i < length; ++i) {
    x = u[i];
    emit( `y = v[hole0];`, i);
    emit( `product = hole0 * y;`, x);
    emit( `sum = sum + product;` );
}

```

Fig. 3. Basic dynamic compiler for dot product. This is a sketch of the customized dynamic compiler for `fpdotproduct`, ignoring helper functions such as dynamic-code cache lookups and memory allocation, and without dynamic zero and copy propagation and dead assignment elimination. The operations identified as static by DyC’s binding-time analysis are executed directly, and the dynamic operations are emitted as executable code, with run-time static values patched into `hole` symbols. This figure presents all code at the source-code level for readability, but DyC’s customized dynamic compilers are generated as assembly code and the emitted instructions are preassembled at static compile time.

specialized. Additionally, the `make_static` on the loop index variable `i` results in the complete unrolling of the loop, because this annotation directs DyC to produce one specialized instance of the loop body for each value of `i`. An `@` sign on an array, pointer, or structure dereference identifies a static load. In this example, the `@` sign ensures that the result of dereferencing the static pointer `u` at the static offset `i` is also static.⁷ The dynamic region extends just to the end of the loop over the vectors, since no static variables are used beyond this point.

These dynamic optimizations are performed at run time by a customized dynamic compiler for the dynamic region that is produced by DyC at static compile time. The customized dynamic compiler interleaves execution of static operations, dynamic optimizations, and dynamic code generation. Figure 3 shows a sketch of a basic customized dynamic compiler that applies the optimizations mentioned above, but not DyC’s additional staged optimizations, zero and copy propagation, and dead-assignment elimination. Static operations are essentially executed directly, whereas dynamic operations are generated as dynamically specialized instructions, parameterized by any run-time static values they reference.

Figure 4 shows an example of the dynamically compiled code produced for the dynamic region on a particular combination of values of the static variables when DyC’s partial-evaluation-style optimizations have been applied, but dynamic zero and copy propagation and dead-assignment elimination have not been (i.e., corresponding to the dynamic compiler in Figure 3). All the static computations in Figure 2 have been folded away by specialization, static uses in dynamic computations (e.g., use of `i` to index `v`) have been instantiated with their run-time-constant values, and the loop has been completely unrolled. Completely unrolling the loop has eliminated the direct costs of branching and induction-variable updating, and, by making the loop induction variable `i` static, it has also indirectly enabled the address calculations and loads from `u` to be eliminated.

⁷In order for this use of `@` to be safe, the contents of a particular vector `u` (i.e., vector with a particular base address) must remain the same for all invocations of `fpdotproduct`. Vectors with different base addresses are permitted by default.

```

sum = 0.0;

  y = v[0];
  product = 0.0 * y;
  sum = sum + product;
} Iteration i = 0

  y = v[1];
  product = 1.0 * y;
  sum = sum + product;
} Iteration i = 1

  y = v[2];
  product = 5.5 * y;
  sum = sum + product;
} Iteration i = 2

```

Fig. 4. Basic dynamically optimized dot-product code. This code shows the dynamically compiled code produced for the dynamic region when `fpdotproduct` is invoked with a vector `u` of length 3 that contains the values [0.0, 1.0, 5.5], but without staged zero and copy propagation and dead-assignment elimination applied. Again, static values are shown in *italics*. Operations that can be further optimized by DyC's staged optimizations are underlined. (The optimized code produced by DyC is actually in machine-code format. We use source code here for readability.)

DyC's dynamic zero and copy propagation and dead-assignment elimination make further improvements to the code for the dynamic region, as shown in Figure 5. The static compiler plans for the possibility of the multiplications and additions being dynamically optimizable by zero or copy propagation. In addition, zero and copy propagation create opportunities for dead-assignment elimination, once again anticipated and planned for statically. In each case, the customized dynamic compiler shown in Figure 3 is updated to optimize affected operations as they are generated. For example, emits of potentially dead assignments are guarded to eliminate dead code at run time. (DyC's customized dynamic compilers make only one pass with no backtracking except for patching forward branches.) In this example, in the first iteration the multiplication by 0.0 is folded away, the following increment of `sum` is removed by zero propagation, and the previous load from the `v` array is deleted as dead. In the second iteration, the multiplication by 1.0 is folded away, with the `y` copy-propagated to the following increment of `sum`. Note that the copy of `y` to `sum` is not propagated and eliminated because it is blocked by an intervening definition of `y` and DyC's current implementation of copy propagation does not rename variables.

This example demonstrates how DyC produces speedup. DyC's dynamic optimizations use the run-time values of variables to produce highly specialized code with fewer and/or cheaper operations than the original, general-purpose code. In this case, the dynamic optimizations reduced the number of operations required by nearly a factor of four.

A number of dynamic optimizations were needed to achieve this reduction. Using complete loop unrolling and folding of static loads eliminated half of the operations, and zero and copy propagation and dead-assignment elimination eliminated half of those remaining. However, the latter optimizations could only be applied because of opportunities exposed by the former.

Static optimizations such as loop-invariant code motion, induction-variable simplification, and loop unrolling cannot achieve the same, dramatic reduction

```

y = v[1];
sum = y;

y = v[2];
product = 5.5 * y;
sum = sum + product;

```

Fig. 5. Fully dynamically optimized dot product. This fully dynamically optimized code shows that many additional operations can be eliminated by DyC's staged zero and copy propagation and dead-assignment elimination optimizations.

in numbers of instructions executed, because they cannot eliminate loads from *u*, all increments of the loop induction variable, all branch-termination tests, and assignments that can only be determined to be dead dynamically.

3. METHODOLOGY

This paper experimentally assesses the benefits, costs and applicability of DyC's optimizations, both individually and when applied together, and analyzes why they achieved the performance improvements they did. In this section, we describe the workload we used for our experiments, explain how we annotated the programs, and describe our experimental methodology.

3.1 Workload

Our workload, shown in Table I, consists of applications that are representative in function, size, and complexity of the different types of programs that researchers are targeting for dynamic compilation. All are used in practice in research or production environments, *dinero* (version III) is a cache simulator that can simulate caches of widely varying configurations and has been the linchpin of numerous memory subsystem studies since it was developed in 1984 [Hill and Smith 1984], *m88ksim* simulates the Motorola 88000 and was taken from the SPEC95 integer suite [SPEC CPU 1995]. *mipsi* [Simer 1993] is a simulation framework that has been used for evaluating processor designs that range in complexity from simultaneous multithreaded [Eggers et al. 1997] to embedded processors. *pnmconvol* is an application from the *netpbm* image processing tool kit (release 7, Dec. 1993) that performs convolutions on images of various formats.⁸ *Viewperf* is the driver for the SPEC *Viewperf* benchmarks; the two routines we dynamically compile in *viewperf* (*project_and_clip_test* (henceforth called *project&clip*), a matrix transformer, and *gl_color_shade_vertices* (henceforth called *shade*), a shader, are from Mesa (version 2.5), a freely available implementation of the Open GL run-time library.⁹ The original Mesa library included additional versions of its general-purpose routines that were hand-specialized for particular combinations of argument values. We deleted these extra hand-specialized versions, letting dynamic compilation automatically generate any needed specializations from the general-purpose code. The general-purpose routines were used as the baseline in our experiments.

⁸Netpbm Web page: <ftp://wv.archive.wustl.edu/graphics/graphics/package/NetPBM/>.

⁹Mesa Web page: <http://www.ssec.wisc.edu/brianp/Mesa.html>.

Table I. Program Characteristics of Our Workload

Program	Description	Annotated Static Variables	Values of Static Variables	Total Size (Lines)	Number and Size of Dynamically Compiled Functions		
					#	Lines	Instructions
Applications							
dinero	cache simulator	cache configuration parameters	8 KB I/D, direct-mapped, 32 B blocks	3,317	8	389	1,624
m88ksim	Motorola 88000 simulator	an array of breakpoints	no breakpoints	12,531	1	14	145
mipsi	MIPS R3000 interpreter	its input program	bubble sort	3,417	1	400	2,884
pmmconvol	image convolution	a convolution matrix	11×11 with 9% ones, 83% zeroes	1,054	1	76	1,226
viewerperf	renderer	a 3D projection matrix, lighting variables	a perspective matrix, one light source	15,006	2	168	1,155
Kernels							
binary	binary search over an array	the input array and its contents	16 integers	147	1	19	134
chebyshev	polynomial function approximation	the degree of the polynomial	10	145	1	19	146
dotproduct	dot product of two vectors	the contents of one of the vectors	a 100-integer array with 50% zeroes	134	1	11	84
query	database entry tested for exact match	a query	7 comparisons	149	1	24	272
rumberg	function integration by iteration	the iteration bound	6	158	1	24	301

We have also included in our workload a set of kernel applications that have comprised the benchmark suites for other dynamic compilation systems for C (binary, chebyshev, dotproduct, query, romberg). The kernels are one or two orders of magnitude smaller than the applications in our workload and contain dynamic regions that are, excluding `m88ksim`, two to eight times smaller. We include them to provide continuity to previous studies [Noël et al. 1998; Poletto et al. 1999] and to contrast their characteristics with the larger programs.

Our workload is currently limited to these programs for a number of reasons. First, our manual annotation process (described below) was time-consuming. Second, for dynamic compilation to be profitable, some programs appear to need techniques or optimizations we have not yet implemented. For example, the standard `perl` interpreter executes its basic operations through indirect function calls. DyC would need to be able to dynamically inline these calls to achieve the same level of optimization as `mipsi`, whose instruction execution is implemented using `switch` statements. Other programs perform cache lookups over a small range of values (e.g., integers between 0 and 255). For such programs, DyC's cache lookup could be implemented as simple array indexing, rather than its current general-purpose (and therefore expensive) hash-table lookup [Grant 2001]. Finally, we found several programs that were not conducive to dynamic optimization. Their potential dynamic regions were executed too infrequently to recoup the dynamic compilation cost, contained too few optimizable operations (usually programs that were already heavily hand-optimized), or contained loops that were too large to be completely unrolled (a number of dense-matrix operations we examined suffered from this latter problem).

3.2 Selection of Optimization Targets

Our annotation methodology depended on the type of program. We annotated the kernels to have the same static variables as previous studies of other dynamic compilation systems. To annotate the applications, we first profiled them with `gprof`. We then examined the functions that comprised the most execution time, searching for invariant function parameters. In cases when invariance was too difficult to infer by inspection, we logged the values of the functions' parameters and searched the log. Optimization opportunities were determined by trial and error. For example, to determine whether complete loop unrolling was beneficial, we generally first performed the unrolling, but then disabled it (by removing an annotation) if it did not improve performance.

By using this unsophisticated methodology, we have undoubtedly missed opportunities to apply dynamic compilation. In particular, a number of additional procedures in `m88ksim` or `viewperf` could potentially benefit from dynamic compilation. One of our future research goals is to automate program annotation, using techniques such as value profiling [Calder et al. 1997] to identify static variable candidates and a cost-benefit model to select appropriate optimizations [Mock et al. 1999].

3.3 Experimental Methodology

The binding-time analysis and the dynamic-compiler generator are implemented in the Multiflow compiler [Lowney et al. 1993], which is roughly comparable to today's standard optimizing compilers. (As a point of reference, dynamic

regions in the applications executed on average 8% more slowly when statically compiled with Multiflow than with `gcc -O2`; kernels were 7% faster.) Because our version of Multiflow has an incomplete implementation of the Compaq Alpha calling convention, most of the nondynamically compiled procedures in the applications were compiled with Compaq's Alpha C compiler or `gcc`.

Each application in our workload has a statically compiled and several dynamically compiled versions, depending on which of DyC's optimizations are turned on. The statically compiled version is compiled by ignoring the annotations in the application source. We used the same options to Multiflow for both versions; this meant, for example, that loops unrolled (by some constant factor) in the statically compiled version were also statically unrolled in the dynamically compiled versions, in addition to any run-time complete loop unrolling.

We produced the majority of our results using a single input for each program (described in Table I). Mid-sized inputs for the kernels were chosen from the sets of inputs used in the studies in which the benchmarks originally appeared. Application inputs that exercised our optimizations were usually chosen from among those provided with their packages. We also explored the sensitivity of our optimizations to particular inputs; the results of this study can be found in Section 5.

All programs were executed on a lightly loaded Compaq Alpha 21164-based workstation with 1.5GB of physical memory. Execution times for both the whole programs and their dynamic regions were measured using `getrusage` (for user time). Whole programs were executed 51 times, with the first run discarded and the rest averaged. When timing dynamic regions, we invoked the specialized functions of most benchmarks many times (tens of thousands of times for the kernels) to overcome the coarseness of the `getrusage` timer and to minimize cache effects. We obtained the time for one invocation by dividing the average of the measurements by the number of invocations timed. The hardware cycle counters were used to gather CPU (user + system) times for dynamic compilation and dispatching overheads, because the granularity of `getrusage` was also too coarse for these measurements.

Dynamic counts of numbers of instructions executed (both in total and categorized by type of instruction or type of dynamic compilation cost) and cache miss rates were measured using an emulation-based, instruction-level simulator for the Alpha instruction set [Tullsen et al. 1996].

4. ANALYSIS OF RESULTS

We begin the analysis of our results with a discussion of the performance bottom line for dynamic optimization, whole-program speedup. We then shift to an analysis of the components of whole-program speedup, the benefits and costs, and examine each in turn. This latter discussion focuses on the portion of the programs that dynamic optimization alters, the dynamic regions. For the applications, this means those sections of code affected by our annotations; for the kernels it is the entire program.

The benefits of dynamic optimization depend on how fast the dynamic regions execute relative to their statically compiled versions (known as their *asymptotic speedup*) and the proportion of time a program executes in the dynamic regions. We quantify both components and discuss two factors that contribute to

Table II. Whole-Program Speedup with All Optimizations. This table presents the execution times for the statically and dynamically compiled applications, the observed speedups of the dynamically compiled versions, and the proportion of execution time contributed by the dynamic regions. Speedup was calculated as the ratio of the execution time of a statically compiled version to that of its dynamically compiled version.

Program	Execution Time (seconds)		Average Whole-Program Speedup	Execution Time of the Dynamic Regions (% of total static execution)
	Statically Compiled	Dynamically Compiled		
dinero	1.3	0.9	1.5	49.9
m88ksim	81.0	76.8	1.1	9.8
mipsi	20.8	4.5	4.6	~100
pnmconvol	10.8	3.6	3.0	83.8
viewperf	7.4	7.3	1.02	41.4

asymptotic speedup: which optimizations could be profitably applied and their relative contributions to performance.

We next examine the principal sources of dynamic compilation overhead, the dynamic optimization and code generation that is performed by the dynamic compiler and dispatching to dynamically generated code. The dynamic compilation overhead depends primarily on the frequency of respecialization (the number of invocations of the dynamic compiler), what optimizations are applied, the control complexity of the dynamically generated code, and the number of instructions that are generated. The dispatch overhead depends primarily on the number of variables whose values are used to specialize the dynamic region and the frequency with which these variables change.

The results analysis concludes in Section 5 with one additional study, an analysis of dynamic optimization performance over a range of inputs, to demonstrate the sensitivity of DyC’s optimizations to the particular inputs used.

4.1 Overall Program Performance

Table II shows observed speedups of the dynamically compiled applications¹⁰ over their statically compiled counterparts. Whole-program speedup, which ranged from 1.02 to 4.6, depended on how much DyC’s optimizations sped up the dynamic region, the cost of the dynamic compile (both discussed below), and the proportion of total run time that was spent executing the dynamic region (column 5). In our programs, the percentage of execution time spent in the dynamic regions ranged from a low of 9.8% for `m88ksim` to almost the entire execution for `mipsi`. Dynamic regions accounted for smaller proportions of execution time in the larger applications (which resulted in lower speedups), at least in part because our manual approach to annotating forced us to concentrate only on those few routines that were executed most frequently, and some potential dynamic regions relied on optimizations we have not yet implemented.

4.2 The Benefits of Dynamic Optimization

Table III contains asymptotic speedups for the dynamic regions of both applications and kernels. Asymptotic dynamic-region speedups for the applications

¹⁰The kernels were called by simple drivers, (i.e., they were measured outside the context of an application); therefore they are not shown in this table of whole-program results.

Table III. Dynamic Region Performance with All Optimizations for Both Applications and Kernels. Asymptotic speedup (Column 2) represents the optimal improvement of dynamically compiled code over statically compiled code (excluding dynamic compilation cost) and is calculated as the ratio of statically compiled execution time of a dynamic region (column 3) over dynamically compiled execution time of the same region (column 4). Note that columns 3 and 4 present times for only a single execution of each dynamic region, whereas Table II's times covered potentially many executions of the programs' dynamic regions, dynamic compilation cost, and the statically compiled portions of the programs as well.

Program [Its Dynamic Region]	Asymptotic Speedup	Execution Time (μ s)	
		Statically Compiled	Dynamically Compiled
dinero [mainloop]	1.6	1,400,000	860,000
m88ksim [ckbrkpts]	3.7	0.22	0.058
mipsi [run]	5.0	22,000,000	4,400,000
pnmconvol [do_convolve]	3.1	52,000,000	17,000,000
viewperf [project&clip]	1.3	58	44
viewperf [shade]	1.2	240	190
binary	1.8	0.12	0.066
chebyshev	6.3	26	4.0
dotproduct	1.5	1.6	1.1
query	1.4	0.11	0.078
romberg	1.2	8.8	7.6

ranged as widely as their whole-program speedups, and, because they include neither dynamic compilation overhead nor execution of the statically compiled code, were higher. The magnitude of the asymptotic speedups was program-dependent. For example, most of the code in *mipsi* and *m88ksim*'s dynamic regions was optimized away as static computations. The gain in *pnmconvol* was primarily due to the benefits of applying a single optimization, dynamic dead-assignment elimination, which was enabled by complete loop unrolling and static loads. *chebyshev* was dominated by static calls to the cosine function, most of which were memorized through dynamic compilation.

DyC achieved speedups on the kernels that were comparable to other dynamic-compilation systems [Noël et al. 1998; Poletto et al. 1999]. That being said, one should be wary when comparing reported results of these systems. All execute on different underlying architectures, and consequently their results could be affected by differences in a number of microarchitectural features. For example, we expect speedups to be greater on an architecture that supports dynamic scheduling (because DyC and similar systems, such as Tempo, currently do no run-time instruction scheduling) or that has a much larger instruction cache (because complete loop unrolling increases code size).

4.2.1 Analysis of the Optimizations. Asymptotic speedup depends in part on how many optimizations can be used and the contribution of each to performance. Table IV indicates which dynamic optimizations were applied to each of the dynamic regions. All optimizations were beneficial to at least one of the applications, and a few were beneficial to nearly all. The diversity of the applications, which include simulators, interpreters, and image-processing and graphics programs, meant that, as a group, they were able to use all of DyC's optimizations. Being smaller, more homogeneous, and less complex than the

Table IV. Optimizations Applied to Each Dynamic Region. A check mark (✓) indicates that the dynamic optimization benefited the dynamic region. No check mark means that the optimization could not be applied (e.g., loop unrolling cannot be used if there are no loops in the dynamic region) or could not be *profitably* applied (e.g., loop unrolling may increase code size to such an extent that additional conflict misses swamp any benefits of applying the optimization). Section numbers in which the optimizations are described appear in parentheses below the optimization name.

Program [Its Dynamic Region]	Optimization									
	Complete Loop Unrolling ^a (2.1.1)	Static Loads (2.1.3)	Static Calls (2.1.3)	Dynamic Zero and Copy Propagation (2.2)	Dynamic Dead- Assignment Elimination (2.2)	Dynamic Strength Reduction (2.2)	Internal Dynamic- to-Static Promotions (2.1.1)	Polyvariant Division (2.1.2)		
dinero	✓					✓				
[mainloop]										
m88ksim	✓									
[ckbrkpts]										
mispi [run]	✓	✓					✓			
pnmconvol	✓			✓						
[do_convol]										
viewperf	✓			✓						
[project&clip]										
viewperf	✓		✓	✓						✓
[shade]										
binary	✓									
chebyshev	SW		✓							
dotproduct	SW				✓					✓
query	SW	✓								
romberg	SW	✓								

^a SW = single-way, MW = multiway

applications, the kernels could take advantage of fewer optimizations. Usually they could apply only the optimizations that were used by most applications (complete loop unrolling and static loads); rarely could they take advantage of the optimizations that are unique to DyC (multiway loop unrolling, dynamic zero and copy propagation, dynamic dead-assignment elimination, dynamic strength reduction, internal dynamic-to-static promotion, and polyvariant division).

To study the effectiveness of individual optimizations, we compared dynamic compilation with all optimizations enabled against runs with one optimization disabled. The results appear in Table V. Table V contains asymptotic speedups; its second column repeats the original values from Table III (which had all optimizations turned on), and later columns show the reduced speedup or even slowdown (worse than not dynamically compiling!) with a selected optimization disabled. Note that in some cases (those with speedups of less than 1.0), omitting an optimization caused a slowdown relative to statically compiled code.

The results in Table V support quantitatively the previous qualitative observation (Table IV) that all of DyC's optimizations are important for dynamic optimization speedups. In particular, some optimizations, such as complete loop unrolling and static loads, contributed significantly to speedups of nearly all of the dynamic regions, and each of the others was critical to speeding up at least one application. Table V shows that for most programs and most optimizations, disabling an optimization produced either slowdowns or marginal speedups relative to the statically compiled code. In a few cases, such as loop unrolling in binary, a single optimization was pivotal to dynamic optimization performance. However, for most dynamic regions, benefits were only seen when all applicable optimizations were used together—eliminating any one of them denied the program speedup. For example, *mipsi* required all four of (multiway) complete loop unrolling, static loads, static calls, and internal promotions to achieve its 5-fold speedup; without any one of these optimizations, *mipsi* slowed down. Without complete loop unrolling, DyC needs to perform a dynamic-to-static promotion (entailing an expensive dynamic-code cache lookup) upon every iteration of the interpretation loop in order to specialize its body. Without either static loads or static calls, DyC is unable to specialize the loop body for the type of instruction executed; this dramatically reduces optimization opportunities, increases the amount of code generated, and requires dynamic-to-static promotions for all interpreted branches. Without internal dynamic-to-static promotions, DyC essentially cannot specialize the interpreter loop at all due to a dependence cycle in *mipsi*'s main loop that includes dynamic control, static loads, a static call, and dynamic loads (whose values are promoted to static). On the other hand, when applied together, these optimizations greatly reduced most types of instructions executed—branches for the interpretation loop, loads for fetching the interpreted instructions, calls and returns from eliminated calls to an address-translation routine, integer operations and indirect jumps for decoding instructions, and stores for register spills.

The kernels, in particular, had their speedups turned into slowdowns from disabling an optimization. Usually, the cause was dynamic optimization overhead, such as dispatching, and the lack of certain capabilities in our current

Table V. Dynamic Region Asymptotic Speedups without a Particular Optimization. This table compares asymptotic speedups with all optimizations enabled to that with a particular optimization disabled. Only those entries that correspond to optimizations that were applied (those with a check mark in Table IV) are shown. A number greater than 1.0 indicates that a dynamic region with a particular dynamic optimization disabled was still faster than its statically compiled version; a number less than 1.0 indicates that it was slower.

Program [Its Dynamic Region]	With All Opts	Complete Loop Unrolling	Static Loads	Static Calls	Dynamic Zero and Copy Propagation	Dynamic Dead- Assignment Elimination	Dynamic Strength Reduction	Internal Dynamic- to-Static Promotions	Polyvariant Division
dinero	1.6		0.9				1.03		
[mainloop]									
m88ksim	3.7	0.4	0.6						
[ckbrkpts]								0.9	
mipsi [run]	5.0	0.5	0.4	0.4					
pnmconvol	3.1	0.8	0.8		2.1	0.9			
[do.convol]									
viewperf	1.3		1.1		1.1				
[project&clip]									
viewperf	1.2	1.0	1.1	1.02	1.1				1.1
[shade]									
binary	1.8	0.6	1.3						
chebyshev	6.3	0.9		1.2					
dotproduct	1.5	0.2	0.5			0.7	0.6		
query	1.4	0.5	0.5						
romberg	1.2	0.8							

implementation, such as dynamic instruction scheduling. In the programs where few opportunities for run-time optimization existed, these negative effects dominated any small gains.

In summary, removing an optimization lost most of the asymptotic speedup or caused a slowdown. This was the case for all optimizations, in all programs. The following sections discuss the contributions of the optimizations in more detail.

Complete Loop Unrolling. Despite its expansionary effect on code size and the consequence for instruction bandwidth requirements and cache footprints, complete loop unrolling (single-way and multiway) was the single most important optimization (column 3 in Table V). Complete loop unrolling was key in speeding up all but two of the dynamic regions, and without it, most applications and all kernels experienced slowdowns relative to their statically compiled counterparts. Some of complete loop unrolling's benefits stemmed from the elimination of all loop overhead. However, additional benefit was also realized because complete loop unrolling enabled other dynamic optimizations. For example, static loads and dynamic strength reduction in `dotproduct` could only be applied when its loop induction variable was a static variable, which only occurred when the loop was completely unrolled. A similar dependence existed between single-way loop unrolling and static loads in `m88ksim`, and multiway loop unrolling and static calls, static loads, and internal dynamic-to-static promotions in `mipsi`. `m88ksim` unrolled over a static table of breakpoints, which eliminated loads of the table entries. `mipsi` multiway unrolled over a static instruction array, eliminating loads of instructions and the instruction decoding logic following the loads, and dynamically memoizing calls to the address translation routine. Sometimes an interdependence existed between complete loop unrolling and another optimization. For example, in `pnmconvol` complete loop unrolling opened opportunities to apply dynamic dead-assignment elimination; eliminating the dead assignments then enabled larger loops to be unrolled without overflowing the instruction cache.

Static Loads. Static loads played a similar role to that of complete loop unrolling. The optimization was very important in all applications and most kernels, both for its direct benefits from eliminating loads and as an enabling optimization. Although just a modest number of loads were eliminated from `dinero`, static loads enabled another optimization, strength reduction, from which `dinero` derived most of its speedup.

Infrequently Used but Pivotal Optimizations. Some optimizations (static calls, dead-assignment elimination, zero and copy propagation, strength reduction, internal dynamic-to-static promotions, and polyvariant division) were used infrequently but, when used, were extremely profitable. For example, treating calls to `cosine` as static in `chebyshev` turned a marginal 20% advantage over the statically compiled configuration into a 6-fold speedup.

The dynamically compiled region of `pnmconvol` executed 3.1 times faster than its statically compiled counterpart, mainly from the contribution of dynamic dead-assignment elimination. Without it, the amount of generated code exceeded the size of the L1 instruction cache by a factor of 2.7, causing slowdowns relative to the static code. Both of `viewperf`'s dynamic regions benefited from dynamic zero and copy propagation, and got only half or a third of the full

improvement without it. `shade` additionally required intraprocedural polyvariant division in order to specialize for the values of variables that were derived as static only on some paths through the procedure, but not on others. Without polyvariant division, many of the other optimizations could not have been performed.

As previously mentioned, `dinero`'s speedup can largely be attributed to the strength reduction of divide and modulus operations used in address calculations.

4.3 Costs of Dynamic Compilation

The previous subsection showed that DyC's dynamic optimizations can significantly improve dynamic region performance. However, these asymptotic speedups are reduced in practice by the costs of generating code at run time. Consequently, the cost of invoking the dynamic compiler limits the frequency with which new specializations can be created for different values.

To eliminate redundant invocations of the dynamic compiler and unnecessary code duplication, DyC maintains an internal cache of previously generated dynamic code, which consists of the code itself and indices that map run-time values to specialized-code addresses. At a dynamic-to-static promotion point (such as the entry to a dynamic region), DyC checks the cache for a version of the region that was compiled for the current values of the annotated variables. If one is found, DyC transfers control directly to it. Only when there is no previously generated version does DyC invoke the dynamic compiler. The newly created code is then saved in the dynamic-code cache.

While the code cache greatly reduces invocations of the dynamic compiler, it introduces the per-promotion-point overhead of dispatching. Because the dispatch incurs overhead on every execution of the dynamic region (or, more generally, at every promotion point), its cost determines the minimum benefit that dynamic compilation of a region of code must yield before any asymptotic speedup can be achieved.

In addition to the indexing that occurs at promotion points, dynamically generated code is indexed at the beginning of other specialization units where code reuse may be possible, such as at the heads of loops (e.g., interpreter loops) that are completely unrolled but whose induction variables do not monotonically change. The cache check avoids unnecessary code duplication and, in some cases, guarantees termination of specialization. When the dynamic compiler reaches a specialization unit and finds (by performing a lookup in that unit's index) that code downstream of that point has already been generated with the current static values, it simply produces a branch to the previously generated code (creating a control-flow merge at that point).

DyC allows users to control the cost of the dynamic-code cache lookups at promotion points and at other specialization units through two policies specified as part of the `make_static` annotation [Grant et al. 2000]. DyC's default for both policies, called *cache-all*, maintains an index at each of these points, which is accessed using double hashing [Cormen et al. 1990]. The index maps the values of the static variables at that point to code downstream of that point that is specialized for those values. The index is checked each time the point is reached in order to reuse specializations should the values of the static variables recur.

This default caching policy does a fairly costly, but safe, cache lookup every time a promotion point or other specialization unit is encountered. Both types of points have alternative policies, which we used in our benchmarks to reduce caching costs.

Invoking the dynamic compiler and dispatching to dynamically generated code are the principal sources of run-time overhead. The following sections analyze these overheads in more detail. Section 4.3.1 examines where the dynamic compiler spends its time, and Section 4.3.2 presents a promotion-point caching policy that reduces dispatching overhead when the annotated variables are invariant. Section 4.3.3 focuses on an additional overhead of dynamic compilation, patching template code holes with run-time-constant values.

4.3.1 Dynamic Optimization and Code Generation Costs. Table VI shows the costs of dynamic code optimization and generation. DyC's dynamic compilation cost was quite low, ranging from 14 to 660 cycles per instruction generated. In contrast, full compilers take hundreds of thousands of cycles per instruction they generate, and even fast, just-in-time compilers for Java, which only perform a handful of local optimizations, take a few thousand cycles per instruction generated. DyC's dynamic compilation costs were low because, as outlined in Section 2, DyC constructs a customized dynamic compiler for each dynamic region. The custom dynamic compiler emits code as it executes, in one pass, with very little buffering of either static values or dynamic instructions, and without consulting an intermediate representation. In addition, most of its duties, such as memory allocation, instruction emission, and hole patching, were implemented as inline code; only cache lookups, cache insertions, and branch patching were implemented by calls to library routines. Experiments with the kernels showed that this inlining improved performance of their custom dynamic compilers by up to a factor of 7. However, excessive inlining negatively affected the performance of the applications' custom dynamic compilers; column 3 in Table VI reveals the cycles per instruction (CPI) of the applications' custom dynamic compilers to be much greater than that of the kernels'. This observation indicates that a strategy that simultaneously reduces function calls but limits code growth would be more effective in decreasing dynamic compilation cost.

As expected, the dynamic compilation cost (columns 2 and 4) generally increased as more instructions were generated (column 6), though there was substantial variance in the cost per instruction generated (columns 3 and 5). `mipsi`, which generated by far the most instructions (36,614), also had the greatest total dynamic compilation cost (7.6M cycles). The number of instructions generated was particularly high in `mipsi` because it unrolled the interpretation of its entire input program. At the other extreme, `m88ksim`, which had a fairly high cost per instruction generated (270 cycles), still had the lowest total dynamic compilation cost (only 2,200 cycles) because of the extremely small number of instructions it produced. The number of instructions `m88ksim` generated was so small, because its dynamic region was a routine that checked breakpoints, and the SPEC input contained none. With other inputs, the number of generated instructions rose and the dynamic compilation cost per instruction fell. For example, our experiments with four breakpoints yielded 71 generated instructions at a cost of only 86 cycles per instruction.

Table VI. Dynamic Compilation Overhead. This table presents dynamic compilation cost measured in both cycles and instructions. Columns 2 and 4 contain the number of instructions and cycles executed by the dynamic compiler, and columns 3 and 5 are these numbers normalized by the number of instructions generated. We also include the number of instructions generated (column 6), to place the total and instruction-specific costs in context, and the number of instructions in the statically compiled configuration of the dynamic region (column 7), to show how much the region grows or shrinks with dynamic compilation.

Program [Its Dynamic Region]	Dynamic Compilation Cost				Number of Dynamically Generated Instructions	Number of Instructions When Statically Compiled
	Total Cycles	Cycles/ Instruction Generated	Total Instructions Executed	Instructions Executed/ Generated		
dinero [mainloop]	210K	330	17K	27	634	1,624
m88ksim [ckbrkpts]	2.2K	370	1.6K	270	6	145
mipsi [run]	7.6M	210	1.9M	52	36,614	2,884
pnmconvol [do convol]	260K	110	150K	63	2,394	1,226
viewperf [project&clip]	75K	660	4.3K	38	114	190
viewperf [shade]	220K	370	19K	32	589	965
Application Average		340		80		
binary	22K	72	16K	53	304	134
chebyshev	25K	31	32K	40	807	146
dotproduct	26K	54	26K	55	474	84
query	3.8K	53	2.7K	38	71	272
romberg	17K	14	17K	14	1,205	301
Kernel Average		45		40		
Overall Average		210		62		

Although fewer instructions were executed by the dynamically compiled programs, their dynamic regions sometimes contained many more instructions than the comparable statically compiled code (compare columns 6 and 7 of Table VI). In particular, complete loop unrolling generated more instructions than the other optimizations and accounted for most of the instructions generated.

Figure 6 shows the relative costs (measured in instructions executed) of the dynamic compiler's principal activities for each dynamic region. The figure's caption explains these categories in detail. As Figure 6 shows, no single source of overhead dominated all dynamic regions. Instead the largest dynamic compilation cost for an individual program depended on what optimizations were performed, on the control complexity of the dynamically generated code, and on how many instructions were generated versus eliminated.

Branch-patching (BP) was the single most costly task for five dynamic regions (mainloop in *dinero*, *ckbrkpts* in *m88ksim*, *project&clip* in *viewperf*, *binary*, and *query*), averaging 21% of their total dynamic compilation cost. The primary reason for its high cost was the call to the branch-patching

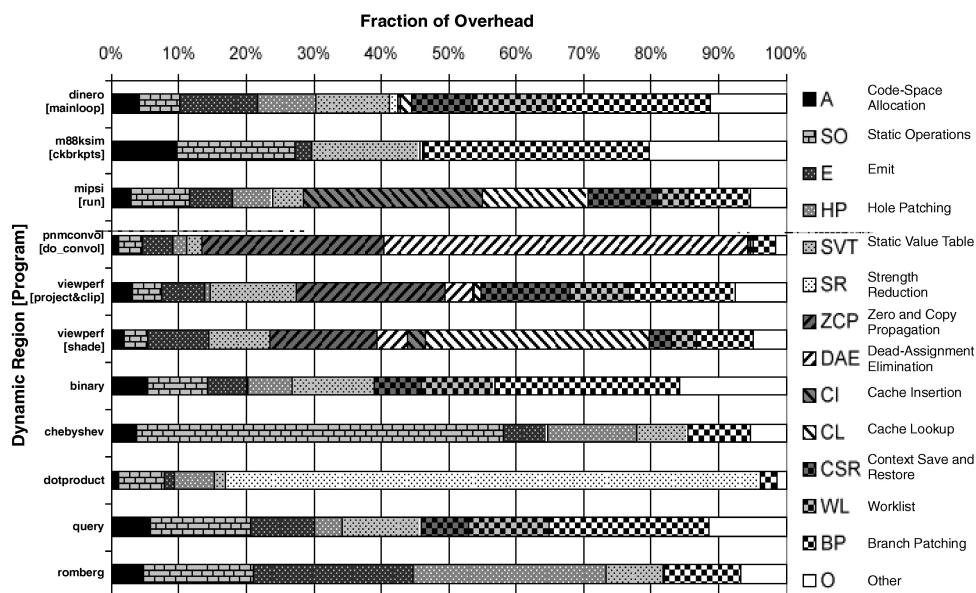


Fig. 6. A breakdown of dynamic compilation cost by subtask. This graph shows the cost (in instructions executed) of dynamic compilation for each dynamic region partitioned into categories derived from the dynamic compiler's principal activities. Related categories are assigned the same pattern but different background shades (e.g., code-cache insertions and lookups both use downward sloped lines). Before dynamically compiling a portion of a procedure, a custom dynamic compiler must construct the cache key and invoke the lookup routine to check whether the code has already been produced using the same static values (category CL). If the appropriate version of code is not found, the dynamic compiler then allocates memory for new code (A), which in some cases must be aligned to a cache-line boundary, and inserts the new location into the code cache (CI). Subsequently, it executes the static instructions (SO), checks conditions under which staged peephole or dataflow optimizations apply (SR, ZCP, DAE), constructs code for dynamic instructions, patches holes with static values (HP), and emits the patched instructions (E). Static values used in dynamically generated code that are too large to fit in instruction immediate fields are inserted into a table (SVT). A worklist is used to suspend and later resume dynamic compilation of dynamic branch successors. The values of live static variables (CSR) and information about the source and destination of the branch or jump must be stored in the worklist (WL). The same information must be stored when generating a reentry stub where dynamic compilation is suspended and then eventually resumed, such as at internal dynamic-to-static promotion points. Finally, the dynamic compiler patches branches whose destinations are within the code just generated (BP) and executes an instruction memory barrier to flush the Alpha's store buffer and to ensure that the instruction cache is up-to-date (IC). The number of instructions required to ensure instruction-cache coherency was too small (usually 1, at most 27) to appear in this graph for all dynamic regions. The catch-all category, 'Other' (O), includes some control-flow, bookkeeping, register-spill, register-restore, and other operations that were scheduled outside our task markers.

routine (which was made once per specialization unit). These five dynamic regions also had the greatest context (CSR) and worklist (WL) overheads (a combined average of 18%) because they produced relatively large numbers of dynamic branches. (DyC's current implementation saves the context and pushes it onto the worklist at every dynamic branch—see the caption of Figure 6 for further explanation.) On the other hand, applications and kernels that had few or no dynamic branches had negligible context and worklist overhead. These three costs (BP + CSR + WL), which together averaged 23% of the dynamic

compilation cost across all dynamic regions, could be reduced by inlining the branch-patching calls and by implementing an optimization we previously described as *linearization* [Grant et al. 2000].¹¹

Code generation tasks, which include constructing and emitting instructions (E), hole patching (HP), maintaining the table of large static values (SVT), and strength reduction (SR), had the greatest combined cost across all dynamic regions (31%), and were the greatest source of dynamic compilation overhead for dotproduct (88%) and romberg (61%). Strength reduction was so costly for dotproduct because it was applied to a large fraction of its generated instructions. In contrast, strength reduction in dinero was applicable to relatively few of its generated instructions and consequently had a lower cost. Code generation was the dominant cost in romberg because it had few static operations, performed few optimizations, and the code it generated contained many holes to patch.

Dynamic-code caching, which incurs costs for both insertions (CI) and lookups (CL), was the largest source of overhead for shade because it had complex dynamic control flow and for mipsi because it required at least one cache lookup for each specialized iteration of its interpretation loop. Caching costs in m88ksim, pnmconvol, and the kernels were negligible because we were able to use the *cache-all-unchecked* caching policy at control-flow merges within their dynamic regions. Cache-all-unchecked eliminates cache lookups at specialization units where the programmer knows the values of the static variables (in this case, the induction variables of completely unrolled loops) do not recur, and therefore checking for code reuse would yield no benefit. (However, the code that is generated for unchecked specialization units is still stored in DyC's dynamic-code cache, since it is reachable from other dynamically specialized code; it is simply not directly indexed.)

Cache insertions were even more costly than cache lookups for mipsi because the large number of insertions triggered a rapid expansion of the dynamic-code cache's hash tables (by repeated reallocations and rehashings of the entries). The hash tables had been designed assuming that many lookups would be performed relative to insertions, and that a single hash table would not have a large number of entries. This was the case for indices at dynamic-to-static promotion points (e.g., at entries to dynamic regions), where most cache lookups were performed to dispatch to previously generated code and relatively few values for the promoted variables occurred, but not for dynamic-code indices at other specialization units (e.g., at the heads of completely unrolled loops). Each specialized iteration of mipsi's interpretation loop (one for each program-counter value) was inserted into the same hash table; however, relatively few of these iterations actually corresponded to control-flow merges. Consequently, comparable numbers of insertions and lookups were performed, and the hash table became very large. This situation is likely to occur for any loop that is unrolled a large number of times and when the cache-all-unchecked annotation cannot be applied. Therefore, a different code-cache implementation should be used for this kind of lookup.

¹¹Linearization eliminates some worklist operations and reduces the cost of saving and restoring values of live static variables at dynamic branches by performing a renaming similar to SSA form [Cytron et al. 1989].

The staged dataflow optimizations (ZCP+DAE) were the top cost for `pnmconvol`. Together, dynamic zero and copy propagation and dead-assignment elimination approximately doubled its total dynamic compilation cost and were thus substantially more expensive than all other cost categories. However, they still appear to be much less costly than applying full dataflow analysis at run time. Like strength reduction, the actions required to perform the optimizations were themselves relatively inexpensive (mainly consisting of testing conditions under which the optimizations apply, clearing and setting analysis table entries, checking the table entries, patching register operand fields, and emitting instructions). However, the frequency with which these actions needed to be performed drove up the cost of the optimizations.

Static operations (SO) are not truly overhead, since they also must be performed by the statically compiled program. However, they are included here as part of dynamic compilation cost because they are performed by the dynamic compiler. Static operations were `chebyshev`'s greatest single source of dynamic compilation cost, comprising 54% of the total. The bulk of this cost emanated from one (relatively expensive) call to the `cosine` function for each (relatively small) unrolled loop body emitted. Overall, static operations accounted for an average of 20% of dynamic compilation cost for the kernels, but only 7% for the applications. The proportion was higher for the kernels, because their simpler structure decreased caching, context, and worklist overhead, and because fewer optimizations could be applied to them.

The cost of code-space allocation, which averaged only 4%, was not a significant source of overhead for any dynamic region. The current implementation statically (and conservatively) estimates the maximum possible size of each specialization unit and dynamically checks for available space before generating code for each unit, but assumes that the code space is never deallocated. Its cost would increase due to additional bookkeeping if we were to allow code space to be manually or automatically deallocated, which would become necessary if dynamic optimization were more liberally applied.

The catch-all category (O), which averaged 8% of total dynamic compilation cost, was smaller than the primary contributors to dynamic compilation overhead.

4.3.2 Dispatching Overhead. The dynamic-code cache reduces dynamic compilation overhead by eliminating redundant invocations of the dynamic compiler, but introduces the (much smaller) costs of the cache lookup and control transfer to the dynamically generated code. These costs comprise dynamic-code dispatching. The dispatch should be fast, because its overhead determines the granularity at which dynamic compilation is profitable. DyC's default cache-all policy (which requires a cache lookup) is safe, but fairly slow. Alternatively, if the programmer knows that a static variable will have the same value for all executions of the promotion point, then the cache lookup can be simplified to a single load and an indirect jump; this policy we call *cache-one-unchecked*.

All of our benchmarks contain some static variables whose values remain invariant throughout program execution. To avoid costly execution-time cache lookups, we annotated the variables with the *cache-one-unchecked* policy. Table VII contains the dispatch costs with checked (*cache-all*) and unchecked (*cache-one-unchecked*) dispatching, and the consequent asymptotic speedups for

Table VII. Performance with Checked and Unchecked Dispatching. `dinero` and `pnmconvol` are not shown because they did not execute any dispatches, since they entered their dynamic regions only once, after code was already produced by their dynamic compilers. `mipsi` could not apply unchecked dispatching because the variable it promoted at its internal promotions (the program counter) was not invariant.

Program [Its Dynamic Region]	m88ksim [ckbrkpts]	viewperf [project&clip]	viewperf [shade]	binary	chebyshev	dotproduct	query	romberg
Average Dispatch Cost (cycles)								
With Checked Dispatching	86	89	90	73	86	88	91	79
With Unchecked Dispatching	15	10	11	10	12	10	10	10
Asymptotic Speedup								
With Checked Dispatching	1.6	1.3	1.2	0.6	6.0	1.3	0.6	1.2
With Unchecked Dispatching	3.7	1.3	1.2	1.8	6.3	1.5	1.4	1.2

all dynamic regions. According to our measurements, an unchecked dispatch has one sixth to one ninth the cost of a general-purpose, hash-table-based dispatch for small numbers of promoted variables (as was the case for our benchmarks, which had at most three).

Although *cache-one-unchecked* has lower cost, it is potentially unsafe, because an annotator may mistakenly use it for static variables whose values do in fact change. Our results indicate that for some real applications, the safer *cache-all* policy can be used without sacrificing much performance (row 5 in Table VII). For *viewperf*, speedups with *cache-all* were identical or very close to speedups with *cache-one-unchecked* because the cache lookup cost was dwarfed by the execution time of the dynamically generated code. The *cache-one-unchecked* policy was important to *m88ksim*, however, because its dynamic region was small and was reentered very frequently (at least once for every simulated instruction). Most kernels were sensitive to *cache-all*'s overhead (in fact, *binary* and *query* suffered slowdowns relative to their statically compiled configurations) because there were too few instructions executed in their small dynamic regions to amortize the cache-lookup cost.

In addition to the dispatch at the entry to the dynamic region, *mipsi* had dispatches at internal promotion points within its fetch-decode-execute loop. These points corresponded to indirect jumps and function returns in the interpreted program. The cost of these dispatches, about 200 cycles each (more expensive than the average due to hash-table collisions), could have been significant if *mipsi*'s input program had contained jumps or returns in an inner loop. That was not the case for the *sort* program input, which was used to produce all results presented for *mipsi* thus far. But it was the case when *dinero* was used as *mipsi*'s input, where dispatch overhead comprised nearly 6% of *mipsi*'s execution time. These interior dispatches could not be optimized using the *cache-one-unchecked* policy because the static variables that comprised their cache keys were not invariant.

These results demonstrate that the performance of some programs could benefit from careful engineering of the dispatch. Our implementation of *cache-all* is not highly optimized. It stores the static variables that comprise the cache's hash key into a structure, performs a function call that computes a hash function based on these values, and then does the lookup. A faster implementation would inline the hash function, only hash on the subset of live static variables being promoted at that point, and use cheaper hash functions when possible. Other techniques, such as simple table lookups for single promoted variables with a small range (e.g., 0 to 255), inline caching for quasi-invariant promoted variables [Deutsch and Schiffman 1984; Hölzle 1993], and efficient dispatching algorithms for multimethods [Dussud 1989; Chambers and Chen 1999], could further reduce the cost of the lookups.

4.3.3 Alternative Hole-Patching Strategies. The efficiency of the dynamically generated code depends on the ability of the dynamic compiler to patch holes with computed values without producing too many additional instructions. The cost section concludes with a discussion of which strategies we found to be effective.

For a dynamic instruction with an integer run-time-constant operand, the instruction's emit code sequence attempts to put the run-time-constant value

into the instruction's immediate field. However, if the integer is too large, a minimal sequence of instructions is emitted to construct the integer constant into a temporary register. Specifically, small values are stored in integer arithmetic immediate fields (this field is 8 bits on the Alpha), and 16-bit values are patched into build-literal and memory operations (e.g., `lda`, `ldah` on the Alpha). Two build-literal operations construct a signed 32-bit integer, with larger integers loaded from a table.

Since Alpha pointer values are almost always greater than 32 bits and no Alpha floating-point operations have immediate fields, pointer and floating-point values are loaded from a run-time-allocated table. The set-up code stores each computed run-time constant into the table, and the template code loads it from the table into a temporary register. By statically inserting the load instructions in the template code rather than when dynamically emitting instructions, we enable the load instructions to be scheduled by the static compiler.

The same load can be repeated in the template code, if there are multiple dynamic instructions that use its value or if the load occurs inside a loop. DyC applies two heuristics to minimize the number of these loads that are executed. The simplest strategy, *use-point loading*, inserts the loads right before each dynamic use. Alternatively, as a simple approximation to loop-invariant code motion and partial redundancy elimination [Briggs and Cooper 1994], DyC inserts a single load instruction immediately after the original definition point of the static variable; we call this *def-point loading*. Although it can eliminate loads in the scenarios mentioned above, def-point loading also has disadvantages. If the definition is far from its uses, the resulting long live range can cause spilling in the dynamically generated code. Additionally, the definition point may be executed more often than the sum of the use points (e.g., if all uses are inside a conditional branch). Consequently, def-point loads may perform worse than use-point loads. A real implementation of partial redundancy elimination of loads inserted in the template code could make more educated decisions regarding load placement.

We experimented with a few alternative strategies for building the values of static variables into dynamically generated code. We compared checking for and patching 8-bit integer values in the immediate field versus producing an additional instruction to build values of all sizes, building 32-bit integers versus loading them from the table, and def-point loading versus use-point loading. All alternatives yielded roughly similar performance. Although theoretically very attractive, patching the immediate fields of integer operations was not important to any of our benchmarks. Constructing 32-bit values was also not needed in most benchmarks, and slightly degraded performance in the only program where it could be used, `mipsi`. (Only `mipsi` had integer hole values that fit in 32 bits but not 16 bits. Using two instructions to construct these values rather than a single load (which completes in 2 cycles on the 21164 if an L1 cache hit) produced nearly 2% more instructions and yielded an asymptotic speedup of only 4.7, compared to the 5.0 speedup obtained with loading.) Similarly, for floating-point and pointer values, we found def-point loading to be comparable to use-point loading for all dynamic regions. Def-point loading did not significantly improve performance because it usually did not reduce the number of loads DyC executed, it increased register pressure, sometimes causing more register spills

and restores, and the number of loads affected accounted for fewer than 1% of executed instructions.

4.4 When Benefits Overcome Costs: Break-Even Points

Dynamic compilation can realize overall performance improvements only if the execution-time savings from the dynamically generated code are greater than the cost of producing it. The overall speedup results presented in Section 4.1 shows that the greater efficiency of the dynamically generated code (Section 4.2) more than compensated for the dynamic compilation costs (Section 4.3) in all applications.

The break-even points at which dynamic region profitability first occurs are shown in Table VIII, column 5. The break-even point is defined as the number of executions of a dynamic region at which its statically and dynamically compiled versions (including dynamic compilation cost) have the same execution times. This point is expressed as the number of instances of a particular input (e.g., memory references) that caused break-even to be reached. Table VIII also includes, in column 6, the quantity of these inputs that were actually executed (i.e., the number of executions of the dynamic regions).

The break-even points for the applications were well within normal usage. For example, dynamically compiling *dinero* paid off after simulating only 3500 memory references—today's cache simulation results are obtained by simulating millions or billions of references. *mipsi*'s break-even point depends on the number of *reinterpreted* input instructions (i.e., the number, sizes, and numbers of iterations of the loops in *mipsi*'s input program, relative to the total size of the input program). For our input, *mipsi* broke even after less than 0.5% of the way through its whole execution.

The break-even points of the kernels were comparable to those reported for other dynamic compilation systems [Noël et al. 1998; Poletto et al. 1999].

5. INPUT-SENSITIVITY ANALYSIS

The results presented so far were generated using a single input for each program. To better assess the range of circumstances under which DyC's optimizations are effective, we compared the performance of statically and dynamically compiled programs for several inputs. We present a subset of those results here. In general, they show that dynamic optimizations are sensitive to program inputs, but that for most inputs used in practice good speedups are achievable for our applications.

Some program inputs determine the values of the annotated (static) variables and, consequently, affect what value-specific optimizations are performed and what code is produced. Other inputs, those to dynamic variables, determine dynamic execution paths and, thus, with what frequency different portions of the run-time optimized code is executed. Because of this difference, we found that in most of our benchmarks the static inputs usually affected performance more than the dynamic inputs. For example, the values of *dinero*'s (static) cache configuration parameters determined whether or not dynamic strength reduction could be effectively applied and, therefore, whether any speedup was possible; however, speedup only varied about 5% across different (dynamic) memory-reference traces. *viewperf* was the exception, being most affected by its

Table VIII. Break-Even Points for All Dynamic Regions. The break-even point is the number of executions of a dynamic region at which its statically and dynamically compiled versions (including dynamic compilation cost) have the same execution times. It is calculated as $o / (s - d)$, where s represents statically compiled execution cycles of a dynamic region (duplicated from Table III, column 3), d is dynamically compiled execution cycles of the same region (duplicated from Table III, column 4), and o is dynamic compilation cost (duplicated from Table VI, column 2). The break-even point is presented in terms of the quantity of input that enabled the program to reach profitability. The last column shows the numbers of dynamic region executions in terms of the quantity of input that occurred in the program's entire execution. These were determined by the whole-program timings (Table II), so none are presented for the kernels.

Program [Dynamic Region]	Execution Time of Statically Compiled Dynamic Region (cycles)	Execution Time of Dynamically Compiled Dynamic Region (cycles)	Dynamic Compilation Cost (cycles)	Break-Even Point	Executions of Dynamic Regions
dinero [mainloop]	71M	43M	210K	3.5K memory references	1M memory references
m8ksim [ckbrkpts]	110	29	2.2K	28 breakpoint checks	36M breakpoint checks
mipsi [run]	11G	2.2M	7.6M	490K instructions	120M instructions
pnmconvol [do_convol]	26G	8.5G	270K	59 pixels	790K pixels
viewperf [project&clip]	29K	22K	75K	11 projections	5.9K projections
viewperf [shade]	120K	97K	220K	12 shadings	5.9K shadings
binary	60	33	22K	840 searches	
chebyshev	13K	2.0K	25K	2 interpolations	
dotproduct	820	570	26K	100 dot products	
query	54	39	3.8K	260 database entry comparisons	
romberg	4.4K	3.8K	17K	31 integrations	

Table IX. Applications’ Dynamic-Region Performance for Different Inputs. This table shows the applications’ dynamic regions’ asymptotic speedups and break-even points for a few static inputs. For reference, the results from Section 4 are included in **BOLD**.

Program [Its Dynamic Region]	Static Input	Asymptotic Speedup	Break-Even Point (computed)
dinero [mainloop]	associativity = 1	1.6	35K memory references
	4	1.4	59K
	16	1.3	59K
	64	1.2	65K
	256	1.03	19K
m88ksim [ckbrkpts]	0 breakpoints	3.7	28 breakpoint checks
	4	3.3	64
	8	3.6	59
	12	2.8	79
	16	2.5	101
mipsi [run]	sort	5.0	485K instructions
	dinero	2.3	2.8M
pnmconvol [do_convolve]	11 × 11, 95% 0s	5.8	41 pixels
	83%	3.1	59
	50%	1.5	100
	25%	1.08	330
	0%	0.9	never
viewperf [project&clip]	perspective matrix	1.3	11 projections
	orthographic matrix	1.2	17K
viewperf [shade]	1 light source	1.2	11 shadings
	2	1.3	6
	4	1.4	3
	8	1.4	2

dynamic inputs. Different dynamic inputs caused different paths in its dynamic regions to be executed that had significantly diverse levels of optimization, or even bypassed its dynamic regions altogether.

Tables IX and X contain asymptotic speedups and break-even points for a few additional static inputs for the applications and kernels, respectively. The inputs included in the table are intended to demonstrate some performance trends we observed. We also varied inputs in other ways in our experiments and report those findings in the text. In all cases, as speedups improved or worsened, the break-even points correspondingly declined or rose.

For some inputs, dynamic compilation caused slowdowns. Dynamic compilation may not always be profitable for a particular dynamic region, primarily for two reasons. First, for some program inputs the dynamic region may not be executed enough to recoup the overhead incurred by dynamic compilation. Second, DyC’s code-expanding optimizations (especially complete loop unrolling) may degrade performance if they create an excessive amount of code with poor spatial and temporal locality.

5.1 Dinero

Good speedups were achieved in *dinero* across different cache and block sizes, since their values are predominantly powers of two and consequently can be straightforwardly strength-reduced. Varying the cache associativity did affect speedup, however, but not because it significantly affected what code was

Table X. Kernel Performance for Different Inputs. This table shows the kernels' asymptotic speedups and break-even points for a few static inputs. For reference, the results from Section 4 are included in **BOLD**.

Dynamic Region	Static Input	Asymptotic Speedup	Break-Even Point (computed)
binary	array size = 16 integers	1.8	836 searches
	1,023	2.1	13,410
	4,095	2.7	30,375
chebyshev	degree = 5	4.1	3 interpolations
	10	6.3	2
	15	6.7	2
dotproduct	20	7.6	2
	length 100, 90% 0s	5.8	13 dot products
	50%	1.5	102
	25%	0.9	never
query	0%	0.7	never
	7 comparisons	1.4	259 entry comparisons
	13	3.1	77
	21	3.5	65
romberg	iteration bound = 2	1.1	64 integrations
	4	1.2	43
	6	1.2	31
	8	1.2	24

generated. Unlike the cache lookup routine presented in one of our earlier papers [Auslander et al. 1996], which searched a particular row in a cache array with a run-time-constant row length, *dinero*'s cache-lookup loops search linked lists. Because of this difference, completely unrolling *dinero*'s lookup loops only reduced loop overhead, which yielded no benefit, and did not expose any further opportunities for eliminating or strength-reducing instructions.¹² Therefore, the transformation was not performed. Consequently, the value of *dinero*'s associativity parameter only marginally affected what code was dynamically generated and the decrease in performance (and increase in the break-even point) with an increase in associativity was simply due to Amdahl's law (i.e., more time was spent in the unoptimized lookup loops and relatively less time was spent on the strength-reduced computations). This effect has little impact in practice, however, because L1 and L2 caches have low associativity and cache simulators are often implemented with cache arrays.

5.2 M88ksim

The routine we optimized in *m88ksim* was *ckbrkpts*, which scans a fixed-length array of breakpoint information to determine whether a breakpoint has been encountered. The same number of entries is scanned regardless of the number of breakpoints set, but if a breakpoint is set, an additional load and comparison are required to test its entry. Our initial input, taken from the SPEC95 integer suite [SPEC 1995], had no breakpoints set, so the entire scanning loop could be eliminated, resulting in a speedup of 3.7. Table IX shows that asymptotic

¹²Keppel [1996] found that dynamic compilation of a lookup using the array implementation improved speedups of a 4-way set-associative cache by 2% on the Alpha.

speedups declined slightly as the number of breakpoints increased, but that a speedup of 2.5 was still achieved with 16 breakpoints, the maximum number possible in the application. Speedups decreased with increasing numbers of breakpoints, because the breakpoint-specific instructions could not be eliminated.

5.3 Mipsi

In addition to the `sort` program input, we used `dinero` as an input to `mipsi`. `dinero`'s main loop was much larger than `sort`'s and contained more expensive operations, such as floating-point operations and function calls.

The asymptotic speedup with `dinero` was less than half of that with the `sort` program. Although there were somewhat fewer instructions eliminated with the `dinero` input than with `sort` (74% versus 81%), there was little difference in the overall instruction mix between the two dynamically compiled executions (i.e., there was approximately the same mix of expensive and cheap instructions). Consequently, it is unlikely that the number and type of generated instructions contributed much to the disparity in speedup.

Instead, increased misses in the L1 instruction cache was the primary factor contributing to the lower speedup with `dinero`. Our simulations showed that dynamically compiled `mipsi` interpreting `dinero` had only an 88% L1 instruction-cache hit ratio, compared to over 99% for `sort`. The increase in misses stemmed from the much larger number of dynamically generated instructions (162K versus 37K) produced with `dinero`. Sustaining good instruction-cache performance will be a challenge for any dynamically compiled program that, like `mipsi`, generates a very large number of instructions.

5.4 Pnmconvol

Most of `pnmconvol`'s speedup stems from dynamic zero and copy propagation and dead-assignment elimination, and the effectiveness of these optimizations is proportionally related to the number of zeroes and ones (particularly zeroes) in the convolution matrix. (The results in Table IX confirm that speedup in `pnmconvol` increases with the proportion of zeroes in the convolution matrix.) However, we were unable to find matrices for performing common image transformations, such as blurring, smoothing, or sharpening, that contained zeroes or ones. For these applications, the entire contents of the convolution matrix sums to a number between zero and one, so that resulting pixel values are roughly in the same range as for the original image. More ones could be created by scaling the matrix by the reciprocal of the most commonly occurring value and then rescaling each pixel value with one additional multiplication, but the number of zeroes cannot be increased. Because of this lack of zeroes, dynamic zero and copy propagation and dead-assignment elimination are probably of use only for niche applications of convolution.

We also varied the size of `pnmconvol`'s convolution matrix and found that the asymptotic speedup of `pnmconvol`'s dynamic region increased with the size of the convolution matrix until the size of the generated code was significantly larger than the L1 instruction cache (data not shown). However, since most convolution matrices used in practice are small, `pnmconvol`'s dynamically generated code should easily fit in today's instruction caches.

5.5 Viewperf

Unlike convolution matrices, all common types of projection matrices contained several zeroes and ones, which made `project&clip` a good application for dynamic zero and copy propagation and dead-assignment elimination. In fact, the numbers of zero and nonzero entries were the same with the orthographic matrix as with our normal input, a perspective matrix. Consequently, the code that was generated for the two inputs was quite similar. The small difference in `project&clip`'s speedups was related to differences in the values of a dynamic parameter that controlled how many times the optimized code was executed for each entry to the dynamic region, and thus the amount of benefit relative to the dispatching cost.

`shade`'s dynamic region iterates over the number of light sources in a scene. Therefore increasing the number of light sources improves `shade`'s performance, because it increases the proportion of execution time spent in the dynamically optimized code.

5.6 The Kernels

The kernels were all small enough that the dispatch overhead had a noticeable impact on their performance when we used small input values. Therefore, the kernels' asymptotic speedups increased with input size until the dispatch overhead was insignificant, or until the portion of generated code that was actually executed exceeded the size of the L1 instruction cache. (Recall that dynamic code generation overhead is not included in asymptotic speedup calculations.) This was precisely the reason for `binary`'s increase in speedup with increasing array length up to 4K integers.

`chebyshev` is dominated by calls to a cosine routine that vary with the degree of the polynomial; its speedup continued to increase up through degree 20, because $O(n^2)$ of the cosine calls are eliminated and only $O(n)$ calls remain.

Speedups in `dotproduct` increase both with the percentage of zeroes in the matrix and the size of the matrix (the latter is not shown). Like `pnmconvol`, `dotproduct` was sensitive to zeroes in its static input vector, and dynamic optimization produced slowdowns when the matrix was too dense. Whereas `pnmconvol` was profitable below 25% zeroes, the point of profitability for `dotproduct` was somewhere between 25% and 50% zeroes. As an example of the effect of matrix size, a static input vector of length 10, with 9 zeroes, generated only a 2.4 speedup, whereas a 100-element vector with 90 zeroes (still a 10% density) yielded a speedup of 5.8.

`query` was small enough that the dispatch had a fairly large impact on its performance. Larger queries yielded greater speedups, but the structure of the queries was also important. The query used in Table III was organized such that its most restrictive comparisons were performed first,¹³ but the query almost always completed after the first or second of the seven comparisons. Larger queries would have similar performance if they did the same, so we organized the larger queries with the most restrictive comparisons last. To illustrate the difference between the comparison organizations, we compared

¹³Key comparison are done until there is a mismatch between key values; putting the most restrictive comparison first will insure that this occurs early.

the 13-comparison query organized most-restrictive last with one ordered most-restrictive first; the former sped up by 3.1, whereas the latter sped up only by 2.0.

Asymptotic speedup for romberg leveled off more rapidly than for the other kernels because the size of the generated code grew fairly quickly with increases in its iteration bound (and all of the code was executed linearly) and relatively few instructions were eliminated.

6. RELATED WORK

As mentioned in the introduction, several previous systems performed selective dynamic compilation, including Tempo [Consel and Noël 1996; Noël et al. 1998], Fabius [Leone and Lee 1995], `^C` [Engler et al. 1996; Poletto et al. 1997; 1999], and our prototype system [Auslander et al. 1996]. Two of our previous publications have compared DyC's features to these other systems in detail [Grant et al. 1997], but in general, DyC supports more flexible treatments of polyvariant specialization and division than the earlier declarative systems, including the important idioms of multiway loop unrolling and conditional specialization. DyC is unique in supporting automatic caching of dynamically compiled code, internal dynamic-to-static promotions, policy annotations that control cache policies, and staged versions of dynamic zero and copy propagation and dead-assignment elimination. Tempo supports an automatic side-effect and alias analysis to eliminate some of the need for static loads and calls, and it also supports interprocedural dynamic regions. Fabius addresses only purely functional ML programs, and because of its limited context of applicability can perform all dynamic compilation automatically and safely, given only hints through a function currying syntax. `^C`'s imperative approach offers programmers direct control over dynamic compilation and optimization, but its high cost in programming complexity may hinder the use of sophisticated optimizations. Register allocation is the only automatic run-time optimization performed by `^C`. Our prototype system included only a limited form of polyvariant specialization that was tailored for single-way loop unrolling, lacked polyvariant division, dynamic zero and copy propagation, and dead-assignment elimination, and did not specialize the dynamic compilers for particular dynamic regions (which led to much greater dynamic compilation cost). Our initial performance study of DyC contained an analysis of whole-program speedups and the contribution of individual optimizations. This paper augments that work with a study of the sensitivity of dynamic optimizations to varying program inputs and a detailed accounting of dynamic compilation overhead.

An alternative to DyC's selective dynamic compilation is dynamically compiling the whole program, perhaps from some intermediate bytecode representation. Current just-in-time compilers for Java follow this approach, as did earlier systems, such as the dynamic optimizing compilers for Self [Chambers and Ungar 1989; 1991; Hölzle and Ungar 1994; 1996] and a dynamic compiler for Smalltalk [Deutsch and Schiffman 1984]. These systems use dynamic compilation to provide better performance for their portable intermediate representation than simple interpretation, or to exploit knowledge of the program available at run time that would be difficult to determine statically. Another key difference between these systems and DyC is that DyC follows a staged

approach to dynamic compilation, reducing the cost of aggressive dynamic optimizations through static preplanning and selectivity; whole-program dynamic compilers tend to curtail their optimization aggressiveness because they do all analysis at run time.

Finally, other researchers have applied various techniques to optimize `m88ksim`, a SPEC benchmark. The most relevant was the value profiling work by Calder et al. [1999], who found different run-time invariants than the one we exploited. By using this information to hand-optimize two routines, `killtime` and `alignd`, which together accounted for 34% of dynamic instructions, they sped up `m88ksim` by 1.2 (compared to the 1.1 speedup we reported in Table II). However, it is not clear how their `killtime` optimization, which changed the procedure’s algorithm and primary data structures, could be automated. In addition, to automatically perform the same optimization they performed on `alignd`, DyC’s staged dataflow optimizations would need to be extended to dynamically eliminate loops whose contents had been eliminated. This would be difficult in a “zero-pass” system¹⁴ such as DyC, that performs dynamic analysis and code generation together in a single pass; at least one additional pass appears to be necessary prior to generating any code, for the loop test and induction-variable update that determines whether the loop’s contents could be eliminated by zero propagation and dead-assignment elimination.

7. CONCLUSION

DyC is capable of producing good speedups on real programs, such as interpreters, simulators, and graphics programs. The range of performance improvement in our benchmarks was large: speedups extended from 20% to a factor of six on the code that was dynamically compiled, which translated into a few percent to a 4.6 improvement for entire applications. Our analysis of DyC’s benefits showed that a few basic techniques were critical to achieving good speedups across all benchmarks, including single- and multiway loop unrolling (conferred by DyC’s general technique of program-point-specific polyvariant specialization) and static loads. As with classical optimizations, other techniques were not universally applicable, but still made a major impact on particular subsets of the benchmarks; such optimizations include dynamic strength reduction, dynamic zero and copy propagation, dynamic dead-assignment elimination, static calls, and internal dynamic-to-static promotions. Each of DyC’s optimizations improved performance not only because of its direct benefits, but also because it created opportunities to apply other dynamic optimizations. Moreover, optimizations were interdependent to the extent that disabling any one optimization often prevented program speedups from the others.

DyC’s dynamic compilation cost is low enough that the break-even point at which dynamic compilation becomes profitable is well within the normal usage of our applications. However, our analysis of DyC’s costs showed that different dynamic optimizations stress different parts of the dynamic compiler. Consequently, to ensure low cost, all parts of the dynamic compilation process must be fast. Although DyC’s dynamic compilation overhead is already very low,

¹⁴This terminology is consistent with general usage, in which many systems that perform analysis and code generation in separate passes are referred to as “one-pass” code generators.

the efficiency of some parts, such as its code-caching routines, may need to be improved to prevent these parts from becoming bottlenecks in applications with small dynamic regions and/or frequent internal dynamic-to-static promotions.

The results of this study suggest several areas for further work, which we have already initiated. Our experience with annotating programs and tuning their performance strongly suggests that more automatic dynamic optimization is desirable to further reduce the burden to the programmer. Moreover, due to the sensitivity to the particular input values used, run-time control over which optimizations to apply is probably necessary. Because every new application we study appears to benefit from extensions of our existing optimizations or new optimizations altogether, we are working on a general framework for staging optimizations. Finally, we intend to expand our study of the interaction of dynamic compilation with the underlying architecture in order to seek opportunities for hardware assistance for software-based dynamic optimizers.

ACKNOWLEDGMENTS

We owe thanks to Trygve Fossum and John O'Donnell for the source for the Alpha version of the Multiflow compiler.

REFERENCES

- AUSLANDER, J., PHILIPOSE, M., CHAMBERS, C., EGGERS, S. J., AND BERSHAD, B. 1996. Fast, effective dynamic compilation. In *Conference on Programming Language Design and Implementation*, May, 149–159.
- BRIGGS, P., AND COOPER, K. D. 1994. Effective partial dead code elimination. In *Conference on Programming Language Design and Implementation*, June, 159–170.
- CALDER, B., FELLER, P., AND EUSTACE, A. 1997. Value profiling. In *International Symposium on Microarchitecture*, Dec., 259–269.
- CALDER, B., FELLER, P., AND EUSTACE, A. 1999. Value profiling and optimization. *Journal of Instruction Level Parallelism 1*, March, 1–37.
- CHAMBERS, C., AND CHEN, W. 1999. Efficient multiple and predicate dispatching. In *OOPSLA '99*, Nov., 238–255.
- CHAMBERS, C., AND UNGAR, D. 1989. Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *Conference on Programming Language Design and Implementation*, July, 146–160.
- CHAMBERS, C., AND UNGAR, D. 1991. Making pure object-oriented languages practical. In *OOPSLA '91*, Nov., 1–15.
- CONSEL, C., AND NOËL, F. 1996. A general approach for run-time specialization and its application to C. In *Symposium on Principles of Programming Languages*, Jan., 145–156.
- CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms*. MIT Press/McGraw-Hill.
- CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1989. An efficient method of computing static single assignment form. In *Symposium on Principles of Programming Languages*, Jan., 25–35.
- DEUTSCH, L. P., AND SCHIFFMAN, A. M. 1984. Efficient implementation of the Smalltalk-80 system. In *Symposium on Principles of Programming Languages*, Jan., 297–302.
- DUSSUD, P. H. 1989. TICLOS: An implementation of CLOS for the explorer family. In *OOPSLA '89*, Oct., 215–220.
- EGGERS, S. J., EMER, J. S., LEVY, H. M., LO, J. L., STAMM, R. L., AND TULLSEN, D. M. 1997. Simultaneous multithreading: a foundation for next-generation processors. *IEEE Micro 17*, (5), 12–19.
- ENGLER, D. R., HSIEH, W. C., AND KAASHOEK, M. F. 1996. C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Symposium on Principles of Programming Languages*, Jan., 131–144.

- GRANT, B. 2001. Benefits and Costs of Staged, Run-Time Specialization. Ph.D. Thesis, University of Washington.
- GRANT, B., MOCK, M., PHILIPSE, M., CHAMBERS, C., AND EGGERS, S. J. 1997. Annotation-directed run-time specialization in C. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, June, 163–178.
- GRANT, B., PHILIPSE, M., MOCK, M., CHAMBERS, C., AND EGGERS, S. J. 1999. An evaluation of staged, run-time optimizations in DyC. In *Conference on Programming Language Design and Implementation*, May, 293–304.
- GRANT, B., MOCK, M., PHILIPSE, M., CHAMBERS, C., AND EGGERS, S. J. 2000. DyC: An expressive annotation-directed dynamic compiler for C. *Theoret. Comput. Sci.* 248, 2, 147–199.
- HILL, M. D., AND SMITH, A. J. 1984. Experimental evaluation of on-chip microprocessor cache memories. In *International Symposium of Computer Architecture*, June, 158–166.
- HÖLZLE, U. 1993. Integrating independently-developed components in object-oriented languages. In *ECOOP '93, Lecture Notes in Computer Science 707*, July, 36–56. Springer-Verlag.
- HÖLZLE, U., AND UNGAR, D. 1994. Optimizing dynamically-dispatched calls with run-time type feedback. In *Conference on Programming Language Design and Implementation*, June, 326–336.
- HÖLZLE, U., AND UNGAR, D. 1996. Reconciling responsiveness with performance in pure object-oriented languages. *Transactions on Programming Languages and Systems* 18, (4), 355–400.
- HORNOF, L., NOYÉ, J., AND CONSEL, C. 1997. Effective specialization of realistic programs via use sensitivity. In *International Symposium on Static Analysis, Lecture Notes in Computer Science 1302*, Sep., 293–314. Springer-Verlag.
- JONES, N. D., GOMARDE, C. K., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall.
- KEPPEL, D. 1996. Runtime Code Generation. Ph.D. Thesis, University of Washington.
- LEONE, M., AND LEE, P. 1995. Optimizing ML with Run-Time Code Generation. Technical report CMU-CS-95-205, Carnegie Mellon University.
- LEONE, M., AND LEE, P. 1998. Dynamic specialization in the Fabius system. *Computing Surveys* 30, (3es) 23-es.
- LOWNEY, P. G., FREUDENBERGER, S. M., KARZES, T. J., LICHTENSTEIN, W. D., NIX, R. P., O'DONNELL, J. S., AND RUTTENBERG, J. C. 1993. The Multiflow trace scheduling compiler. *Journal of Supercomputing*, 1–2, 51–142.
- MOCK, M., BERRYMAN, M., CHAMBERS, C., AND EGGERS, S. J. 1999. Calpa: a tool for automating dynamic compilation. In *Workshop on Feedback-Directed Optimization*, Nov.
- NOËL, F., HORNOF, L., CONSEL, C., AND LAWALL, J. L. 1998. Automatic, template-based run-time specialization: Implementation and experimental study. In *International Conference on Computer Languages*, May, 132–142.
- POLETTI, M., ENGLER, D. R., AND KAASHOEK, M. F. 1997. cc: A system for fast, flexible, and high-level dynamic code generation. In *Conference on Programming Language Design and Implementation*, June, 109–121.
- POLETTI, M., ENGLER, D. R., HSIEH, W. C., AND KAASHOEK, M. E. 1999. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems* 21, (2), 324–369.
- SIRER, E. G. 1993. Measuring Limits of Fine-Grain Parallelism. Princeton University Senior Project, June.
- SPEC CPU. 1995. <http://www.specbench.org>.
- TULLSEN, D. M., EGGERS, S. J., EMER, J. S., LEVY, H. M., LO, J. L., AND STAMM, R. L. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, May.

Received January 2000; accepted May 2000