An Evaluation of Staged Run-Time Optimizations in DyC

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

Department of Computer Science and Engineering
University of Washington
Box 352350, Seattle WA 98195-2350
{grant,matthai,mock,chambers,eggers}@cs.washington.edu

Abstract

Previous selective dynamic compilation systems have demonstrated that dynamic compilation can achieve performance improvements at low cost on small kernels, but they have had difficulty scaling to larger programs. To overcome this limitation, we developed DyC, a selective dynamic compilation system that includes more sophisticated and flexible analyses and transformations. DyC is able to achieve good performance improvements on programs that are much larger and more complex than the kernels. We analyze the individual optimizations of DyC and assess their impact on performance collectively and individually.

1. Introduction

Selective dynamic compilation transforms selected parts of programs at run time, using information available only at run time, to optimize them more fully than strictly statically compiled code. (Selective dynamic compilation is in contrast to *complete dynamic compilation* where all compilation of a program is delayed until run-time; recent "just in time" compilers for Java are examples of complete dynamic compilers.) *Value-specific* selective dynamic compilers derive their benefits 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 selective, 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, such as parameterization for reuse and portability across different hardware architectures, also imply a promising role for dynamic compilation.

Recent research efforts have made considerable progress towards proving the viability of selective 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 reasons current systems have not made better progress on larger, more complex applications vary, depending on the approach taken. In *imperative* systems, such as `C [12, 27, 28], a programmer explicitly constructs, composes, and compiles code fragments at run time. Imperative approaches can express a wide range of optimizations, but 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 [6, 26], Fabius [21], and our previous system [1], 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 keep dynamic compilation costs low, these systems preplan the possible effects of dynamic optimizations statically, producing a specialized dynamic compiler tuned to the particular part of the program being dynamically optimized; this sort of preplanning we call *staging* the optimization. Declarative approaches are relatively easy for programmers to use, but are only as powerful as the 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 optimization opportunities or forcing substantial rewriting to fit the limitations of the system.

DyC (pronounced *dicey*) [13, 14] is a selective, 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 a sophisticated form of partial-evaluation binding-time analysis, including program-point-specific polyvariant division and specialization,[1] and dynamic versions of traditional global and peephole optimizations. To keep dynamic compilation costs low, nearly 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 declaratively specify *policies* that govern the aggressiveness of specialization and caching, enabling programmers to get finer control over the dynamic compilation process while preserving the declarative model. (In the future, we hope to treat DyC as a back-end for a tool that automatically decides where to apply selective, value-specific dynamic optimizations, generating annotations that DyC then carries out.)

This paper assesses the benefits and applicability of DyC's analyses and transformations, both individually and when applied together,

[1] Polyvariant 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.

and analyzes why these optimizations achieved the performance improvements they did.[1] The optimizations are evaluated on a selection of medium-sized, widely used applications that are representative of the application classes mentioned earlier. The results show that:

- Dynamic compilation produces speedups on real applications, not simply kernels. Although there was a subset of optimizations that benefited all programs (both kernels and applications), each of DyC's optimizations was important to obtaining good speedup on some application.
- Several optimizations were important to all our benchmarks. Complete loop unrolling was the single most important optimization, not only because it eliminated all loop overhead but also because it created many opportunities for other dynamic optimizations. Optimizing loads from invariant parts of data structures was similarly critical for most programs.
- Some optimizations unique to DyC, such as dynamic dead-assignment elimination, were responsible for significant speedups (3x-5x) in some applications.
- DyC's dynamic compilation overhead 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. Section 3 details our experimental methodology and workload. Section 4 contains our performance results, including a comparison of whole programs versus their dynamic regions and an analysis of the contribution to performance of individual optimizations. Section 5 compares DyC to related research and section 6 concludes.

2. DyC

2.1 System Overview

DyC compiles and optimizes programs dynamically, during their execution. To trigger run-time 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* 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.

DyC extends a traditional (static) optimizing compiler with two major components, as illustrated in Figure 1:

- As in off-line partial-evaluation systems [20], 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 lifetime of these static variables determines the extent of the 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.
- For each dynamic region, a *dynamic-compiler generator* produces a specialized dynamic compiler that will generate code at run time for that region, given the values of the static variables on entry to the region.

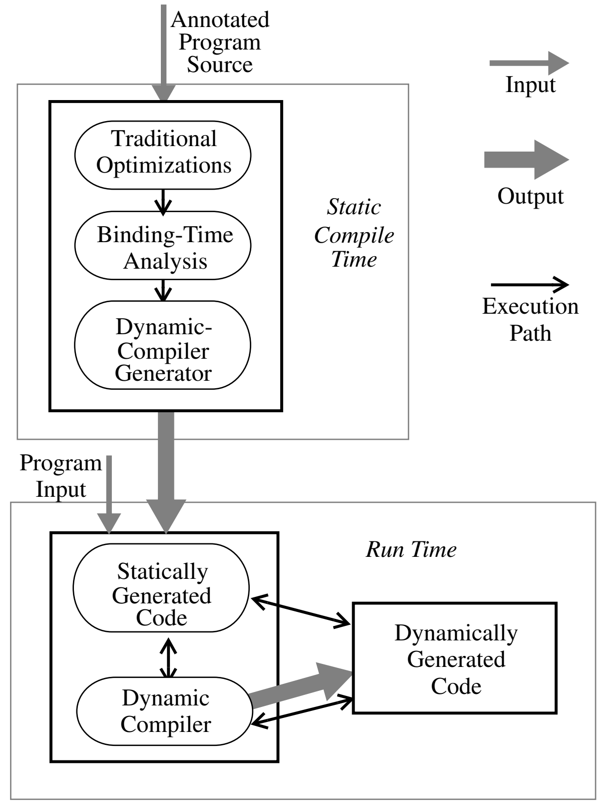

Figure 1. DyC's Static and Dynamic Components

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

- First, DyC applies many traditional intraprocedural optimizations, stopping just prior to register allocation and scheduling.
- Then, for procedures that contain annotations, the binding-time analysis identifies derived static variables and the boundaries of dynamic regions. This analysis also determines which conditional branches and switches test static variables and so can be folded at dynamic compile time. It also determines which loops have static induction variables and can therefore be completely unrolled at dynamic compile time.
- Each dynamic region is replaced with two control-flow subgraphs, one containing the static computations (called *set-up code*) and one containing the dynamic computations (called *template code*). Where a dynamic instruction in the template code refers to a static operand, a place-holder operand (called a *hole*) is used. The hole will be filled in at dynamic compile time, once its value is known.

[1] Our previous paper and journal article on DyC [13, 14] describe its design, but include no empirical assessment. Dynamic, staged zero and copy propagation and dead-assignment elimination are new with this paper as well.

- 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 separately, 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 the dynamic region can be allocated registers seamlessly across dynamic-region boundaries.
- Finally, a custom dynamic compiler for each dynamic region (also called a *generating extension* [20]) is built simply by inserting *emit* code sequences into the set-up code for each instruction in the template code; the template subgraph is then thrown away. This 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.

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 the values of the annotated variables. If one is found, it is reused. Otherwise, the dynamic compiler continues executing, evaluating the static computations and emitting machine code for the dynamic computations (and saving the newly generated machine code in the dynamic-code cache when it is done). Invoking the dynamic compiler and dispatching to dynamically generated code are the principal sources of run-time overhead.

2.2 DyC's Run-Time Optimizations

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, starting with the annotations that identify static variables and ending after the last use of any static value (a dynamic region may have multiple exits). This analysis distinguishes static computations from dynamic computations, enabling run-time constant propagation without incurring any run-time cost from analyzing an intermediate representation. This analysis is program-point-specific and flow-sensitive: a dynamic region can start and stop at any program point, and a variable may be static at some program points and not at others.

DyC's ability to produce efficient dynamic code depends on several extensions to this basic approach. Some, such as polyvariant specialization and division, are derived from the partial-evaluation field but adapted and extended to the needs of dynamically compiling C programs. Others are special staged versions of traditional global and peephole optimizations, such as zero and copy propagation and dead-assignment elimination.

2.2.1 Dynamic-to-Static Promotions and 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 allows multiple versions of the code after the promotion to be generated, each specialized for a different combination of promoted values; this is called *polyvariant specialization*. A compiled-code-cache lookup implements the promotion.

2.2.2 Internal Promotions

In DyC, promotions also can occur at arbitrary program points in the middle of a dynamic region, called *internal promotion points*, enabling a kind of *multi-stage specialization* [22]. For example, an internal promotion can occur at the point a static variable is assigned a dynamic value, to allow specialization on the value of the static variable to resume (at the cost of a cache check for 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.

2.2.3 Unchecked Dispatching

When a promotion point is executed, the point's cache of dynamically compiled code is checked for code that has been specialized to the current values of the static variables. If found, that code is executed; if not, a version is specialized to these values. This *dispatch* to dynamically generated code, comprised of the cache lookup and indirect jump, should be fast because its overhead is incurred on every execution of the dynamic region. Currently, DyC uses a policy annotation to control the cost of the dynamic-code cache lookups found in the template code at promotion points. DyC's default policy, called *cache-all*, maintains a cache at each of these points, implemented using double hashing [7]. The cache maps the values of the static variables at that point to code specialized for those values. The cache is checked each time the point is reached in order to reuse specialization should the values of the static variables recur. 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; this policy we call *cache-one-unchecked*.

2.2.4 Complete Loop Unrolling

Polyvariant specialization can also result in complete loop unrolling by creating a specialized copy of a loop body for each set of values of the loop induction variables. 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 different loop iterations (*e.g.*, if it contains branch paths that update the loop induction variables differently), or even return to a previously executed loop iteration, producing in general a directed graph of unrolled loop bodies (which we call *multi-way loop unrolling*).

Complete loop unrolling is unlike unrolling done by traditional static compilers in that the unrolled loop is eliminated rather than enlarged. The main benefit of complete unrolling is derived from the additional constant- and branch-folding opportunities exposed by making the loop induction variables static, rather than from increased instruction-level parallelism. DyC and similar systems (such as Tempo) currently do no run-time instruction scheduling.

2.2.5 Polyvariant Division

Polyvariant division allows the same program point to be analyzed multiple times, each time with a different set of variables assumed 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.

Although we did not use the capability for this study, polyvariant division also supports *conditional specialization*: rather than unconditionally executing an annotation, the programmer guards the annotation with an arbitrary test of whether specialization is desirable. Polyvariant division will then automatically duplicate 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.

2.2.6 Static Loads and Calls

By default, the contents of memory, even if referenced through a run-time- or compile-time-constant address, is assumed to be dynamic. In many programs, however, at least some of the contents of these data structures are invariant. In such programs, 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 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.[1]

These annotations are potentially unsafe programmer assertions. In contrast, Tempo performs an automatic alias and side-effect analysis within a compilation unit to identify static portions of data structures and pure functions within that unit. However, Tempo still relies on potentially unsafe user annotations to discover the alias and side-effect properties of external data structures and procedures [6].

2.2.7 Strength Reduction, Zero and Copy Propagation, and Dead-Assignment Elimination

Some of DyC's optimizations exploit particular values of static variables. The emit code sequences perform strength reduction of multiplies, divides, and modulus operations with a single static operand and attempt to fit integer static operands into instruction immediate fields. (We currently emulate dynamic strength reduction by inserting special-case code in the program source.)

DyC also includes a novel *staged* version of dynamic zero and copy propagation and dead-assignment elimination that depends 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 operand of the multiply is other than `0` or `1`, no copy propagation or dead-assignment elimination can be performed.

To perform data-dependent zero and copy propagation and dead-assignment elimination with low run-time overhead, DyC divides the 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 potential downstream uses of the result 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 deleted, and a note is made in a table maintained during dynamic compilation. Emit code sequences for uses of the potentially optimized instruction check the 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 to record the results of optimizations.

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 simplified version of the image-convolution routine `pnmconvol` from our benchmark suite. The `do_convol` routine takes an `image` matrix as input, convolves it by the convolution matrix `cmatrix`, and writes the convolved output to the `outbuf` matrix. Since `cmatrix` is unchanged within the (outer) loops over image pixels, we would like to specialize the inner loops over the convolution matrix to the values contained in `cmatrix`.

The three DyC annotations inserted to accomplish this dynamic specialization are highlighted in boldface. 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 the pointer `cmatrix` and its dimensions `crows` and `ccols` should be specialized upon in downstream code. Additionally, the `make_static` on the loop index variables `crow` and `ccol` results in the complete unrolling of the innermost two loops. An `@` sign on an array, pointer, or structure dereference identifies a static load. In our example, the `@` sign ensures that the result of dereferencing the static pointer `cmatrix` at the static offset (`crow`, `ccol`) is also static. Derived static computations and uses, identified by the BTA, are italicized. The dynamic region extends to the end of the loop over the convolution matrix, since no static variables are used beyond this point.

Figure 3 shows a source-code representation[2] of the dynamically compiled code produced for the dynamic region when `do_convol` is invoked with a 3×3 `cmatrix` that contains

[1] Tempo 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 values [19].

```
/* Convolve image with cmatrix into outbuf */
void do_convol(
  float image  [][], int irows, int icols,
  float cmatrix[][], int crows, int ccols,
  float outbuf [][]
)
{
  float x, sum, weighted_x, weight;
  int crow, ccol, irow, icol, rowbase, colbase;
  int crowso2, ccolso2;

  make_static(cmatrix, crows, ccols, crow, ccol);

  crowso2=crows/2; ccolso2=ccols/2;

  /*Apply cmatrix to each pixel of the image*/
  for (irow=0; irow < irows; ++irow){
    rowbase = irow-crowso2;
    for (icol=0; icol < icols; ++icol){
      colbase = icol-ccolso2; sum = 0.0;

      /*Loop over convolution matrix*/
      for (crow=0; crow<crows; ++crow){
        for (ccol=0; ccol<ccols; ++ccol){
          weight = cmatrix @[crow] @[ccol];
          x = image[rowbase+crow][colbase+ccol];
          weighted_x = x * weight;
          sum = sum + weighted_x;
      }}
      /*End loop over convolution matrix*/

      outbuf[irow][icol] = sum;
  }}
  /*End loop over image*/
}
```

Figure 2. Image Convolution Example

alternating zeroes and ones (zeroes in the corners). (For the moment we ignore the effect of the DyC-specific dynamic zero and copy propagation and dead-assignment elimination optimizations described in section 2.2.7.) All the static computations in Figure 2 have been folded away by specialization, static uses in dynamic computations (*e.g.*, that of `ccol` and `crow` to index `image`) have been instantiated with their run-time constant values, and the loop over the convolution matrix has been completely unrolled. Completely unrolling the loop has eliminated the direct cost of branching and induction variable updating, but by making the loop induction variables `crow` and `ccol` static, it also indirectly has enabled the address calculations and load from `cmatrix` to be eliminated.

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 4. 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 creates opportunities for dead-assignment elimination, once again anticipated and planned for statically. In this example, in each even iteration the multiplication by 0.0 is folded away, the following increment of `sum` removed by zero propagation, and the previous load from the `image` array deleted as dead. In each odd iteration, the multiplication by 1.0 is folded away with the `image` value `x` copy-propagated to the following increment of sum.

[2] The optimized code produced by DyC is actually in machine-code format. We use source code here for readability.

```
/*Apply cmatrix to each pixel of the image*/
for (irow=0; irow < irows; ++irow ){
  rowbase = row-1;
  for (icol=0; icol < icols; ++icol){
    colbase = icol-1;

    /*Loop over convolution matrix*/
    /*Iteration 0: crow=0, ccol=0*/
      x = image[rowbase][colbase];
      weighted_x = x * 0.0;
      sum = sum + weighted_x;

    /*Iteration 1: crow=0, ccol=1*/
      x = image[rowbase][colbase+1];
      weighted_x = x * 1.0;
      sum = sum + weighted_x;

    /*Iteration 2: crow=0, ccol=1*/
      x = image[rowbase][colbase+2];
      weighted_x = x * 0.0;
      sum = sum + weighted_x;

    /*Iterations 3-8 follow...*/
      ...

    outbuf[irow][icol] = sum;
}}
/*End loop over image*/
```

Figure 3. Partially Dynamically Optimized Region

```
/*Apply cmatrix to each pixel of the image*/
for (irow=0; irow < irows; ++irow ){
  rowbase = row-1;
  for (icol=0; icol < icols; ++icol){
    colbase = icol-1;

    /*Loop over convolution matrix*/
    /*Iteration 0: crow=0, ccol=0*/
      /*All code eliminated*/

    /*Iteration 1: crow=0, ccol=1*/
      x = image[rowbase][colbase+1];
      sum = x;

    /*Iteration 2: crow=0, ccol=1*/
      /*All code eliminated*/

    /*Iteration 3: crow=1, ccol=0*/
      x = image[rowbase+1][colbase];
      sum = sum + x;

    /*Iterations 4-8*/
      ...

    outbuf[irow][icol] = sum;
}}
/*End loop over image*/
```

Figure 4. Fully Dynamically Optimized Region

Table 1: Application Characteristics

Program	Description	Annotated Static Variables	Values of Static Variables	Total Size (Lines)	Number & Size of Dynamically Compiled Functions		
					#	Lines	Instructions
Applications							
`dinero`	cache simulator	cache configuration parameters	8kB I/D, direct-mapped, 32B blocks	3,317	8	389	1624
`m88ksim`	Motorola 88000 simulator	an array of breakpoints	no breakpoints	12,531	1	14	145
`mipsi`	MIPS R3000 simulator	its input program	bubble sort	3,417	1	400	2884
`pnmconvol`	image convolution	convolution matrix	11x11 with 9% ones, 83% zeroes	1,054	1	76	1226
`viewperf`	renderer	3D projection matrix, lighting vars	perspective matrix, one light source	15,006	2	168	1155
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 90% zeroes	134	1	11	84
`query`	tests database entry for match	a query	7 comparisons	149	1	24	272
`romberg`	function integration by iteration	the iteration bound	6	158	1	24	301

3. Methodology

This paper assesses the benefits and applicability of DyC's analyses and transformations, both individually and when applied together, and analyzes why these optimizations 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 1, 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 [15]. `m88ksim` simulates the Motorola 88000 and was taken from the SPEC95 integer suite [30]. `mipsi` [29] is a simulation framework that has been used for evaluating processor designs that range in complexity from simultaneous multithreaded [11] to embedded processors. `pnmconvol` is an application from the `netpbm` image processing toolkit (release 7-Dec-93) that performs convolutions on images of various formats [25]. `viewperf` is the driver for the SPEC Viewperf benchmarks; the two routines we dynamically compile in `viewperf` (`project_and_clip_test`, a matrix transformer, and `gl_color_shade_vertices`, a shader) are from Mesa (version 2.5), a freely available implementation of the OpenGL run-time library [24]. The original Mesa program included additional versions of its general-purpose shader routine that were hand-specialized for particular combinations of argument values. We deleted these extra hand-specialized versions, letting dynamic compilation automatically generate any needed specialized versions from the general-purpose version.

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 [26, 28] 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, to be profitable, some programs need techniques or optimizations we have not yet implemented. For example, a decompression program and a version of `grep` could become profitable to compile dynamically if DyC supported fast cache lookups over a small range of values (*e.g.*, integers between 0 and 255). For such cases, the lookup could be implemented as a simple array indexing, in place of DyC's current general-purpose hash-table lookup. Finally, we found several programs that were not conducive to dynamic optimization: one type contained dynamic regions that were executed too infrequently or were too small to recoup the dynamic compilation overhead; another type contained loops that were too large to be completely unrolled (a number of dense-matrix operations we examined suffered from this problem).

3.2 Selection of Optimization Targets

Our annotation methodology depended on the type of program. We annotated the kernels to enable optimizations that are comparable to what other dynamic-compilation systems provide. 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 [2] to identify static variable candidates, and a cost-benefit model to select appropriate optimizations.

3.3 Experimental Methodology

The binding-time analysis and the dynamic-compiler generator are implemented in the Multiflow compiler [23], 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 compiled with Multiflow than with `gcc -O2`; kernels were 7% faster.) Because our version of Multiflow had an incomplete implementation of the DEC Alpha calling convention, most of the non-dynamically compiled procedures of the applications were compiled with DEC's C compiler or `gcc`.

Each application in our workload has a statically compiled and several dynamically compiled versions, depending on what 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 the statically and dynamically compiled versions. This means, for example, that loops unrolled (by some constant factor) in the statically compiled version are also statically unrolled in the dynamically compiled versions, in addition to any run-time complete loop unrolling.

All programs were executed on a lightly loaded DEC Alpha 21164-based workstation with 1.5GB of physical memory. A single input was used for each program (described in Table 1). 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 usually were chosen from among those provided with their packages.

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 (to reduce systems effects) and the rest averaged. When timing dynamic regions, most benchmarks invoked their specialized functions 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.

4. Results and Discussion

4.1 Applicability of the Optimizations

Table 2 indicates which dynamic optimizations could be used by each of the programs. All optimizations were needed by at least one of the applications, and several were used by all. Lacking the complexity of the applications, the kernels could take advantage of fewer optimizations. Usually they could apply only the optimizations that are used to all applications (unchecked dispatching, complete loop unrolling, static loads); rarely could they take advantage of the optimizations that are unique to DyC (multi-way loop unrolling, dynamic zero and copy propagation, dynamic dead-assignment elimination, dynamic strength reduction, internal dynamic-to-static promotion, and polyvariant division). This difference suggests that studies of dynamic compilation opportunities should focus on larger, more realistic programs.

4.2 Dynamic Region Performance

Basic performance results for the dynamic regions of both the applications and the kernels appears in Table 3. We report asymptotic speedups, break-even points, and dynamic compilation overhead. Asymptotic speedup represents the optimal improvement of dynamically compiled code over statically compiled code (excluding dynamic compilation overhead), and is calculated as `s/d`, the ratio of statically compiled execution cycles (`s`) over dynamically compiled execution cycles (`d`). The break-even point is the number of executions of the dynamic region at which statically and dynamically compiled code (including dynamic compilation overhead) have the same execution times; it is calculated as `o/(s-d)`, where `o` represents cycles of dynamic compilation overhead. Dynamic compilation overhead is measured as cycles per dynamically generated instruction; we also include the number of instructions generated to place the instruction-specific overhead in context.

Asymptotic dynamic-region speedups for the applications ranged widely, from 1.2 to 5.0. The higher speedups for `mipsi` and `m88ksim` (5.0 and 3.7, respectively) occurred because most of the code in their dynamic regions could be 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.

Break-even points for the applications are well within normal application usage, showing that the greater efficiency of the dynamically generated code can more than compensate for the dynamic compilation cost. For example, dynamically compiling `dinero` pays off after simulating only 3524 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 *re*interpreted instructions (*i.e.*, the

Table 2: Optimizations Used by Each Program

Dynamic Region	Optimization								
	Complete Loop Unrolling[a]	**Static Loads**	**Unchecked Dispatching**	**Dynamic Dead-Assignment Elimination**	**Dynamic Zero&Copy Propagation**	**Static Calls**	**Dynamic Strength Reduction**	**Internal Dynamic-to-Static Promotions**	**Poly-variant Division**
`dinero: mainloop`		✓	✓				✓		
`m88ksim: ckbrkpts`	SW	✓	✓						
`mipsi: run`	MW	✓	✓			✓		✓	
`pnmconvol: do_convol`	SW	✓	✓	✓	✓				
`viewperf: project&clip`		✓	✓	✓	✓				
`viewperf: shader`	SW	✓	✓	✓	✓	✓			✓
`binary`	MW	✓	✓						
`chebyshev`	SW		✓			✓			
`dotproduct`	SW	✓	✓	✓			✓		
`query`	SW	✓	✓						
`romberg`	SW		✓						

a. SW = single-way, MW = multi-way

Table 3: Dynamic Region Performance with All Optimizations

Dynamic Region	Asymptotic Speedup	Break-Even Point	DC Overhead (cycles/instruction generated)	Number of Instructions Generated
`dinero: mainloop`	1.7	1 invocation (3524 memory references)	334	634
`m88ksim: ckbrkpts`	3.7	28 breakpoint checks	365	6
`mipsi: run`	5.0	1 invocation (484634 instructions)	207	36614
`pnmconvol: doconvol`	3.1	1 invocation (59 pixels)	110	2394
`viewperf: project&clip`	1.3	16 invocations	823	122
`viewperf: shade`	1.2	16 invocations	524	618
`binary`	1.8	836 searches	72	304
`chebyshev`	6.3	2 interpolations	31	807
`dotproduct`	5.7	6 dot products	85	50
`query`	1.4	259 database entry comparisons	53	71
`romberg`	1.3	16 integrations	13	1206

number and size of the loops in `mipsi`'s input program) relative to the total size of the input program. For many inputs, conditional specialization as described in section 2.2.5 could be used to limit dynamic compilation to those parts of `mipsi`'s input program that are heavily executed.

The main contributors to dynamic-compilation overhead are cache lookups, memory allocation, handling of dynamic branches, checks for dynamic zero and copy propagation, dead-assignment elimination, and strength reduction, operations to ensure instruction-cache coherence, instruction construction and emission, branch patching, hole patching, and the static computations. Although DyC is quite fast, each of these costs could be further reduced (`dinero`, in particular, suffers from our inefficient handling of dynamic branches). For example, we have not yet implemented the optimization we previously described as *linearization* [14], which would reduce the cost of saving and restoring values of static variables at dynamic branches by performing a renaming similar to SSA form [8].

Complete loop unrolling generates more instructions than the other optimizations and accounts for most of the instructions generated.

Table 4: Whole-Program Performance with All Optimizations

Application	Execution Time (sec.) Statically Compiled	Execution Time (sec.) Dynamically Compiled	Execution Time in the Dynamic Regions (% of total static execution)	Average Whole-Program Speedup
dinero	1.3	0.9	49.9	1.5
m88ksim	81.0	76.8	9.8	1.05
mipsi	20.8	4.5	~ 100	4.6
pnmconvol	10.8	3.6	83.8	3.0
viewperf	1.7	1.6	41.4	1.02

These numbers were particularly high in `mipsi`, because it unrolls the interpretation of its entire input program, and in `pnmconvol`, because each iteration is large. The number of instructions generated is small in `m88ksim`, because its dynamic region is a routine that checks breakpoints and the SPEC input contains none. With other inputs, the number of generated instructions should rise and the dynamic overhead per instruction fall. For example, our experiments with 5 breakpoints yielded 98 generated instructions at a cost of only 66 cycles per instruction.

In contrast to the applications, dynamic regions in the kernels are small, with simple code idioms. Consequently, optimizations they use can usually be applied to the entire region, but, on the other hand, not many optimizations can be applied. The high kernel speedups can be attributed to a key optimization or a particular input. For example, `chebyshev` is dominated by static calls to the cosine function, most of which are memoized through dynamic compilation. `dotproduct`'s static input vector was 90% zeroes and therefore most of the calculations were eliminated; our experiments on more dense vectors produced speedups similar to those of the other kernels, and with no zeroes the dynamically compiled version experiences a slowdown due to poor instruction scheduling.

DyC achieves speedups and break-even points on the kernels that are comparable to other dynamic compilation systems [26, 28]. That being said, however, quantitative comparisons of these systems are not particularly meaningful, because all execute on different underlying architectures. Our preliminary studies indicate that several microarchitectural features, in particular, instruction issue width, dynamic-scheduling support, and L1-instruction-cache size, are major determinants of dynamic-compilation performance.

4.3 Whole-Program Performance

Whole-program speedup due to dynamic compilation depends on the proportion of total run time that is spent executing the dynamic region. In our applications, the percentage of execution time spent in the dynamic region ranged from 9.8% for `m88ksim` to almost the entire execution for `mipsi` (see Table 4). Overall application speedup, including the cost of dynamic compilation, was not quite proportional to this fraction of asymptotic speedup (due to cache and other system effects) and ranged from 1.02 to 4.6.

4.4 Analysis of Individual Optimizations

To study the effectiveness of individual optimizations, we compared our normal configuration with all optimizations enabled against configurations each of which disabled one optimization; the results appear in Table 5. The second column gives the original speedup from Table 3 with all optimizations turned on, and later columns show the reduced speedup with a selected optimization disabled. Only those entries corresponding to applicable optimizations (those with a check mark in Table 2) are shown.

4.4.1 Complete Loop Unrolling

Despite its expansionary effect on code size and the consequence for instruction bandwidth requirements and cache footprints,[1] complete loop unrolling (single-way and multi-way) was the single most important optimization (column 3). Complete loop unrolling was responsible for much of the speedup in the applications, and without it, most programs experienced slowdowns relative to their statically compiled counterparts. Some of complete loop unrolling's benefits stem from the elimination of all loop overhead, but additional benefit is realized because it enables other dynamic optimizations. For example, static loads and dynamic strength reduction in `dotproduct` only apply when its loop induction variable is a static variable; this only occurs when the loop is completely unrolled. A similar dependence exists between multi-way loop unrolling and static calls, static loads, and internal dynamic-to-static promotions in `mipsi`, and single-way loop unrolling and static loads in `m88ksim`. `m88ksim` unrolls over a static table of breakpoints which eliminates loads of the table entries. `mipsi` multi-way unrolls 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 inter-dependence exists between complete loop unrolling and another optimization: for example, in `pnmconvol` complete loop unrolling opens opportunities to apply dynamic dead-assignment elimination; eliminating the dead assignments then enables us to unroll larger loops.

4.4.2 Static Loads

Static loads (column 4) 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.

4.4.3 Unchecked Dispatching

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. Under this policy, the dispatch is

[1] As mentioned in section 2.2.5, conditional specialization could be used to prevent unreasonable code expansion due to complete loop unrolling.

Table 5: Dynamic Region Asymptotic Speedups without a Particular Feature

Dynamic Region	With All Opts	Without:								
		Complete Loop Unrolling	Static Loads	Unchecked Dispatching	Static Calls	Dynamic Zero&Copy Propagation	Dynamic Dead-Assignment Elimination	Dynamic Strength Reduction	Internal Dynamic-to-Static Promotions	Poly-variant Division
`dinero:` `mainloop`	1.7		0.9	1.6				1.03		
`m88ksim:` `ckbrkpts`	3.7	0.4	0.6	1.6						
`mipsi:` `run`	5.0	0.9	0.9	5.0	0.9				0.9	
`pnmconvol:` `do_convol`	3.1	0.8	0.8	3.1		2.1	0.9			
`viewperf:` `project&clip`	1.3		1.1	1.3		1.1	1.3			
`viewperf:` `shader`	1.2	1.0	1.1	1.2	1.02	1.1	1.2			1.1
`binary`	1.8	0.6	1.3	0.6						
`chebyshev`	6.3	0.9		6.0	1.2					
`dotproduct`	5.7	0.3	0.9	3.4			0.7	0.7		
`query`	1.4	0.5	0.5	0.6						
`romberg`	1.3	0.8		1.2						

implemented using a load and an indirect jump. An unchecked dispatch requires about 10 cycles, according to our measurements. In contrast, a general-purpose hash-table-based dispatch (supporting the default cache-all policy) requires on average 90 cycles. In `mipsi`, this figure rises to 150 cycles per dispatch, due to collisions in its hash table.

Although cache-one-unchecked has better performance, it is potentially unsafe, because an annotator may mistakenly use it for static variables whose values do change. Our results indicate that for many real applications, the safe cache-all policy can be used without sacrificing much performance (column 5). With one exception, speedups with cache-all were identical or very close to speedups with cache-one-unchecked, because few of the cache lookups were actually executed. The cache-one-unchecked policy is important to `m88ksim`, however, because it enters its dynamic region for each simulated instruction; consequently, with cache-all `m88ksim` would incur a cache lookup for each instruction. The kernels were more sensitive to cache-all's overhead (in fact, `binary` and `query` suffered slowdowns relative to their statically compiled versions), because there were too few instructions executed in their small dynamic regions to amortize the cache-lookup cost.

These results demonstrate that the performance of some programs could benefit from careful engineering of the dispatch (*e.g.*, to avoid collisions). 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 version 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 inline caching [9, 16] and efficient dispatching algorithms for multimethods [10, 3], could further reduce the cost of the lookups, particularly where only one or a few possible combinations of values occur.

4.4.4 Infrequently Used but Pivotal Optimizations

Some optimizations were used infrequently, but, when used, were extremely profitable (static calls, zero and copy propagation, dead-assignment elimination, strength reduction, internal dynamic-to-static promotions, and polyvariant division (columns 6-11)). For example, treating calls to cosine as static in `chebyshev` turned a marginal 20% advantage over the statically compiled version into a 6-fold speedup. `mipsi` required all three of complete loop unrolling, static loads, and static calls to achieve its 5-fold speed-up; without any one of these `mipsi` slowed down.

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 cache by a factor of 2.7, causing slowdowns relative to the static code.

Strength reduction appears to be a similarly pivotal optimization, but until DyC does strength reduction automatically, we withhold judgement. Our manual source-code implementation may result in optimistic results, because it incurs dynamic compilation overhead only where strength reduction is known to be profitable. Conversely, our manual method may miss opportunities for strength reduction.

`viewperf`'s shader 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. Without polyvariant division, many of the other optimizations could not have been performed.

5. Related Work

As mentioned in the introduction, several previous systems performed selective dynamic compilation, including Tempo [6, 26], Fabius [21], `C [12, 27, 28], and our previous system [1]. Previous publications have compared DyC's features to these other systems in detail [13, 14], but in general, DyC supports more flexible treatments of polyvariant specialization and division than the earlier declarative systems, including the important idioms of multi-way loop unrolling and conditional specialization. DyC is unique in supporting automatic caching of dynamically compiled code, internal dynamic-to-static promotions, policy annotations controlling cache policies, and staged versions of dynamic zero and copy propagation and dead-assignment elimination. Tempo supports an automatic side-effect and alias analysis within a compilation unit 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 programers 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 previous 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 overhead). Most importantly for this paper, however, is that previous systems were only evaluated on kernel-sized benchmarks; our emphasis in this paper has been to develop and assess techniques targeting the needs of much larger programs.

An alternative to DyC's selective dynamic compilation is complete dynamic compilation, where the whole program is compiled dynamically, perhaps from some intermediate bytecode representation. Current so-called just-in-time compilers for Java follow this approach, as did earlier systems such as the dynamic optimizing compilers for Self [4, 5, 17, 18] and a dynamic compiler for Smalltalk [9]. 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. The key difference between these systems and staged dynamic compilation in DyC is that DyC reduces the cost of aggressive dynamic optimizations through static preplanning and selectivity, while complete dynamic compilers tend to curtail their optimization aggressiveness because of the limited amount of time available for analysis.

6. Conclusion

DyC builds on the successes previous dynamic compilation systems have had on small kernels, extending their repertoire of techniques in order to be effective on larger programs. Overall, DyC enabled speedups on dynamically compiled code of 1.2 to 5.0, which translated to speedups of 1.02 to 4.6 for applications as a whole, including the overhead of dynamic compilation. A few basic techniques are critical to achieving good speedups across all benchmarks, including single- and multi-way loop unrolling (conferred by DyC's general technique of program-point-specific polyvariant specialization), user-controlled caching policies, and static load annotations. As with classical optimizations, other techniques are not universally applicable, but they still make a major impact on particular subsets of benchmarks; such optimizations include dynamic strength reduction, dynamic zero and copy propagation, dynamic dead-assignment elimination, static calls, and internal dynamic-to-static promotions. Other techniques are not proven in our current benchmark suite, but could be very important for variations on these benchmarks. For example, interpreters and instruction simulators such as `mipsi`, could benefit from conditional specialization via polyvariant division in order to avoid specializing paths that are never or only infrequently executed.

As with other current dynamic compilation systems, DyC relies on programmer annotations to choose good dynamic regions and static variables. We view this work as a study evaluating the underlying *mechanisms* of dynamic compilation; the *policy* decisions are left to programmers. Our next major step is to build on this understanding by developing a system that works towards automating the policy decisions. Our long-term goal is a system that automatically performs dynamic compilation as one of many possible compiler optimizations, guided by static analyses and profile-driven feedback.

Acknowledgments

We owe thanks to David Grove and the anonymous PLDI reviewers for improving the quality of our discussion, and to Tryggve Fossum and John O'Donnell for the source for the Alpha version of the Multiflow compiler. This work was supported by ONR contract N00014-96-1-0402, ARPA contract N00014-94-1-1136, NSF Young Investigator Award CCR-9457767, and an Intel Graduate Fellowship.

References

[1] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effective dynamic compilation. *SIGPLAN Notices*, pages 149–159, May 1996. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*.

[2] B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 259–269, December 1997.

[3] Craig Chambers and Weimin Chen. Efficient predicate dispatching. Technical Report UW-CSE-98-12-02, Department of Computer Science and Engineering. University of Washington, December 1998.

[4] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 146–160, June 1989.

[5] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *Proceedings OOPSLA '91*, pages 1–15, November 1991. Published as *ACM SIGPLAN Notices*, volume 26, number 11.

[6] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of POPL '96: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, January 1996.

[7] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

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

[9] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302, January 1984.

[10] Patrick H. Dussud. TICLOS: An implementation of CLOS for the Explorer family. In *Proceedings OOPSLA '89*, pages 215–220,

October 1989. Published as *ACM SIGPLAN Notices*, volume 24, number 10.

[11] S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, R.L. Stamm, and D.M. Tullsen. Simultaneous multithreading: A foundation for next-generation processors. *IEEE Micro*, 17(5):12–19, August 1997.

[12] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. ‘C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Conference Record of POPL ’96: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 131–144, January 1996.

[13] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Annotation-directed run-time specialization in C. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 163–178, Amsterdam, The Netherlands, June 1997. New York: ACM.

[14] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*. To appear.

[15] M.D. Hill and A.Jay Smith. Experimental evaluation of on-chip microprocessor cache memories. In *ISCA ’84*, June 1984.

[16] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In P. America, editor, *Proceedings ECOOP ’91*, LNCS 512, pages 21–38, Geneva, Switzerland, July 15-19 1991. Springer-Verlag.

[17] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN ’94 Conference on Programming Language Design and Implementation*, pages 326–336, June 1994.

[18] Urs Hölzle and David Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems*, 18(4):355–400, July 1996.

[19] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Seman-tics-Based Program Manipulation*, pages 63–73, Amsterdam, The Netherlands, June 1997. New York: ACM.

[20] Neil D. Jones, Carstein K. Gomarde, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, NY, 1993.

[21] M. Leone and P. Lee. Optimizing ML with run-time code generation. Technical report CMU-CS-95-205, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 1995.

[22] Mark Leone and Peter Lee. Dynamic specialization in the Fabius system. *ACM Computing Surveys*, 30(3es):23–es, September 1998.

[23] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O’Donnell, and J. C. Ruttenberg. The Multiflow trace scheduling compiler. *Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[24] Mesa web page. http://www.ssec.wisc.edu/ brianp/Mesa.html.

[25] Netpbm web page. ftp://wuarchive.wustl.edu/graphics/graphics/ packages/NetPBM/.

[26] F. Noël, L. Hornof, C. Consel, and J. L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *International Conference on Computer Languages*, May 1998.

[27] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. *SIGPLAN Notices*, pages 109–121, June 1997. In *Proceedings of the ACM SIGPLAN ’97 Conference on Programming Language Design and Implementation*.

[28] M. Poletto, D. R. Engler W.C. Hsieh, and M. F. Kaashoek. ’C and tcc: A language and compiler for dynamic code generation. To appear in *Transactions on Programming Languages and Systems*.

[29] Emin Gun Sirer. Measuring Limits of Fine-Grain Parallelism. Princeton University Senior Project, June 1993.

[30] SPEC CPU, August 1995. http://www.specbench.org/.