Calpa: A Tool for Automating Selective Dynamic Compilation

Markus Mock, Craig Chambers, and Susan J. Eggers
Department of Computer Science and Engineering
University of Washington
Box 352350, Seattle WA 98195-2350
{mock, chambers, eggers}@cs.washington.edu

Abstract

Selective dynamic compilation systems, typically driven by annotations that identify run-time constants, can achieve significant program speedups. However, manually inserting annotations is a tedious and time-consuming process that requires careful inspection of a program's static characteristics and run-time behavior and much trial and error in order to select the most beneficial annotations. Calpa is a system that generates annotations automatically for the DyC dynamic compiler. Calpa combines execution frequency and value profile information with a model of dynamic compilation cost and dynamically generated code benefit to choose run-time constants and other dynamic compilation strategies. For the programs tested so far, Calpa generates annotations of the same or better quality as those found by a human, but in a fraction of the time. The result was equal or better program speedups from dynamic compilation, but without the need for programmer intervention.

1. Introduction

Dynamic compilation optimizes programs at run time, based on information available only at run time, thus offering the potential for greater performance than purely statically compiled code. Some dynamic compilation systems, including Smalltalk-80 [27], Self [28, 29], and just-in-time compilers for Java (for example, [30]), perform virtually all compilation during program execution. Others are selective about which parts of programs they dynamically compile. *Selective* dynamic compilation systems can focus the additional run-time effort of dynamic compilation on those portions of the program that most benefit from dynamic compilation, leaving the remainder of the program to be compiled statically.

Selective dynamic compilers usually base their optimizations on run-time-computed values of particular variables and data structures (called *run-time constants*). A region of a procedure that references these run-time constants (called a *dynamic region*) is *specialized* at dynamic compile time for the particular run-time values of these variables, with the dynamic compiler performing various constant-propagation- and loop-unrolling-like optimizations on the dynamic region. The specialized code can be reused for any future executions of the dynamic region where the run-time constants have the same values as was assumed when the code was specialized. If a dynamic region is invoked with different values for the run-time constants, multiple specialized versions of the dynamic region can be generated and maintained. On entry to the dynamic region at run time, a *dispatcher* selects the appropriate specialized version, based on the values of the run-time constants, or invokes the dynamic compiler to produce a new version.

Dynamic compilation involves additional overheads during program execution that statically compiled programs don't incur, namely, specialization and dispatching. In order for dynamic compilation to be profitable, the run-time benefits dynamic compilation obtains must outweigh these run-time costs. This requires that the code that is specialized to the values of the run-time constants be sufficiently better optimized and reused sufficiently often to more than recoup the costs of producing it and dispatching to it; typically, this precludes performing any expensive analysis at run time. Hence, the effectiveness of a selective dynamic compilation system depends critically on choosing good run-time constants and dynamic regions. On the positive side, since the actual values of the constants don't have to be known until run time, dynamic compilation systems can optimize code in situations where a compile-time specializer can either not be used at all, or may suffer from code explosion because code may have to be specialized for *all possible* values instead of only the values that *actually occur* at run time.

Previous dynamic compilation systems require the programmer to choose run-time constants and dynamic regions manually. The selections are communicated to the dynamic compilation system through declarative annotations. Fabius [17] and Tempo [5,20] allow the user to annotate formal parameters of procedures as run-time constants, causing the annotated procedures to be specialized for the particular values of the annotated parameters. DyC [10,12,11] provides finer-grained dynamic optimization, by allowing users to annotate individual variables at arbitrary points within a procedure; the variables are treated as run-time constants up to the end of their scope, or until another annotation returns them to run-time-variable status. DyC also has a set of annotations with which programmers can regulate the aggressiveness of specialization and certain other dynamic compilation costs.

For the most part, annotations do not affect the behavior of the program, only how it is implemented. However, because they do not include a whole-program side-effect analysis, Tempo and DyC include some annotations that the statically compiled portions of the program may render unsafe; if the programmer uses an unsafe annotation incorrectly, the program's behavior can be changed through dynamic compilation. (Fabius avoids the need for unsafe annotations by handling only purely functional programs.)

To manually select annotations that will produce program speedups. programmers must gain a good knowledge of the application's run-time behavior, perhaps aided by an execution frequency profile of the various regions of the program and a log of the values of variables and selected data structures. They must understand the effects of candidate annotations on the relative quality of the dynamically compiled code versus the statically compiled code, and on the run-time cost to produce it and dispatch to it. They must anticipate how often the specialized version(s) will be reused on a typical application execution. Finally, when using an unsafe annotation, they must be confident that the annotation's assumptions about program behavior are satisfied.

Consequently, manually annotating programs so that they achieve good speedups is difficult and time consuming, and often becomes the bottleneck for many applications that could benefit from dynamic optimization. Our experience with DyC serves as a case in point. To annotate the applications for our initial evaluation of DyC's run-time optimizations [12], we first profiled them with `gprof`. We then examined the functions that comprised the most execution time, searching for invariant or *quasi-invariant* function parameters, i.e. function parameters with only one or a few values, respectively. In cases when invariance was too difficult to infer by inspection, we logged the values of the functions' parameters by manually inserting code into the applications, and then searched the logs. 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. Since some of DyC's annotations control the aggressiveness of specialization and the cost of run-time compilation, choosing the best combination here turned out to be laborious and time-consuming as well. All in all, although DyC's annotations are few and conceptually simple, finding good candidates for run-time constants and picking appropriate specialization strategies turned out to be a very tedious process, taking us several weeks to annotate the applications for that study.

Calpa[1] [19] is the first system to automate this process. Calpa combines program analysis and profile information to automatically derive the annotations that drive dynamic compilation (in our case, of C programs using DyC). Calpa consists of two modules: an *instrumentation tool* and an *annotation selection tool*. The programmer first runs the instrumentation tool on the program to produce an instrumented version of the program that will generate and summarize value and frequency data. (Figure 1 presents an overview of the process for using Calpa and DyC.) The instrumented version is then executed on some representative input, yielding an execution profile of this run-time information. The programmer then invokes the annotation selection tool on the program and the profile. The annotation selector searches the space of possible dynamic compilation annotations, using an internal model of the costs and benefits of dynamic compilation and the execution profile to estimate the overall impact of each candidate annotation. The selection tool reports its choices by producing an annotated program. This annotated program is then compiled by DyC to yield a final executable program that contains specialized dynamic compilers. In essence, Calpa uses DyC as a back-end to carry out its chosen dynamic compilation strategy; Calpa encapsulates the *policy* decisions about where, on what and how aggressively to dynamically compile, while DyC encapsulates the *mechanism* of performing dynamic compilation. Unlike programmer-inserted annotations, Calpa's selected dynamic compilation strategy is guaranteed to be safe, i.e., it will not change the observable behavior of the program.

To test Calpa's effectiveness in automatic annotation against the "gold standard" of manual annotation, we used Calpa to derive annotations for some kernels and applications we had previously hand-annotated. The entire profiling and annotation process was completed for the kernels within seconds and for the applications either in minutes or hours, depending on the program. Calpa produced all the annotations that we had identified with our manual process, and, in addition, for some programs,

[1] Calpa was an important Inca oracle ritual, which was performed before making important decisions.

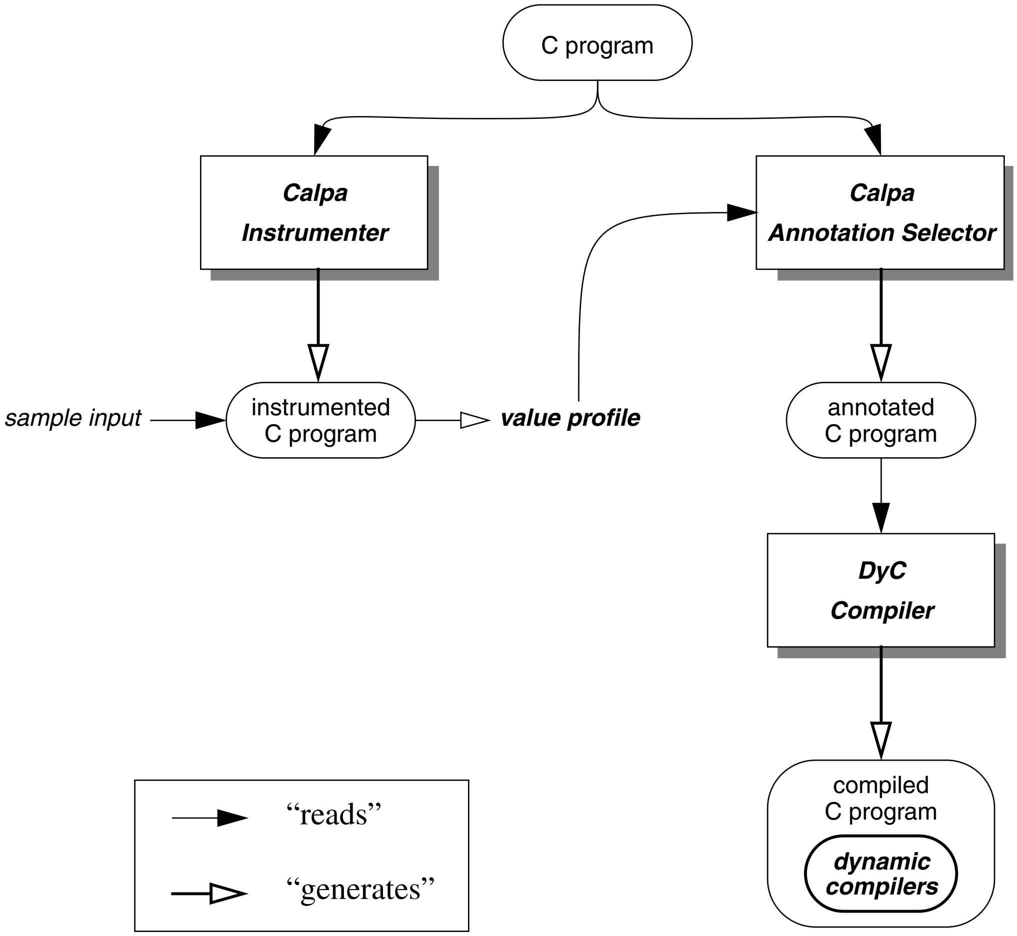

Figure 1. Overview of Calpa

inserted others that improved their performance.

This paper presents Calpa's approach to automating selective dynamic compilation. We describe Calpa's instrumentation and annotation selection tools and the interface between them. We also report our initial performance results using a prototype implementation of Calpa, including the time and space it takes to instrument and profile the applications and to derive the annotations, and the speedups of Calpa-annotated, dynamically optimized programs relative to those that were manually annotated. The paper makes the following contributions:

- We present a program analysis that combines various static analyses with value profile information to identify good candidate variables and dynamic regions for dynamic compilation.
- We develop a multi-faceted cost/benefit model based on run-time specialization that enables the automatic evaluation of the costs and benefits of dynamically compiling different annotation choices.
- We demonstrate experimentally that these techniques produce annotations for a dynamic compiler that results in comparable or better program speedups than manual annotations, at a small fraction of the time a human annotator would need to understand and profile the programs. While each of the techniques by itself is not new, Calpa's merit lies in its special combination of techniques that provides useful value.

In the next section we summarize Calpa's targeted dynamic compilation system, DyC. Section 3 describes Calpa. Section 4 describes the experiments we carried out to assess the effectiveness and the resource use of Calpa. Section 5 discusses related work and section 6 concludes.

2. Overview of DyC

DyC is an annotation-driven, selective dynamic compilation system that attains speedups of up to 4.6× on a group of medium-sized (< 15K lines of code) C programs [12]. To achieve this level of performance, DyC contains

(1) a sophisticated form of partial-evaluation-style binding time analysis (BTA) that supports program-point-specific polyvariant division and specialization,[1] (2) low-cost, dynamic versions of traditional global optimizations that include zero and copy propagation and dead-assignment elimination, and (3) dynamic peephole optimizations, such as strength reduction.

To trigger dynamic compilation, programmers annotate their source code to identify *static variables* (run-time constants) on which many calculations depend. Loads from memory can be annotated as static if their contents are static, and called procedures can be annotated as static if their results are static whenever given static arguments. DyC's BTA analyzes code downstream of the annotations, intraprocedurally, to separate those computations that depend solely on the annotated static variables, plus additional static variables that are derived from them, (called the *static computations*) from the computations that depend at least in part on run-time variables (called the *dynamic computations*). Static computations are executed just once, at dynamic compilation time. By delaying final compilation of the dynamic computations until run time, the results of the static computations can be treated as embedded constants in the dynamic computations. A part of a procedure that contains static computations and the directly dependent dynamic computations constitutes a dynamic region. Since the BTA 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.

For each dynamic region, DyC builds a custom dynamic compiler (also called a *generating exten*sion [16]) that generates code at run time, using the values of the static variables once they become known. To minimize dynamic compilation overhead, DyC performs much of the analysis and planning for dynamic optimization during static compile time, freeing the custom dynamic compiler from needing an intermediate representation and iterative analyses at run time.

When a dynamic region is entered at run time, the region's dispatcher checks an internal cache of previously dynamically generated code for a version that was compiled for the current values of the annotated variables. If one is found, it is executed. If not, the dispatcher invokes the region's custom dynamic compiler to generate code specialized to the current values of the annotated variables. The custom compiler evaluates the static computations and emits machine code for the dynamic computations. When done, the newly generated code is saved in the dynamic-code cache and then executed. Invoking the dynamic compiler and dispatching to dynamically generated code are the principal sources of run-time overhead.

If a load is annotated as static, the dynamic compiler assumes that the contents of the referenced memory location remains the same for all invocations of the dynamic region. If the memory location is later updated, then the affected dynamically compiled code should be thrown away and redynamically compiled for the new value. In the current DyC system, whenever a store changes the contents of a memory location upon which code might be specialized, the program must invoke DyC's `invalidate()` operation, which flushes the compiled code caches of all affected functions, and resets the dynamic region's code pointer to point to the dynamic compiler. The dynamic compiler will then produce new code for the invalidated dynamic region, based on the new contents of the memory location.

DyC is driven primarily by annotations that identify initial run-time constant variables and data structures. Optional policy annotations allow the programmer to specify whether specialization and division should be mono- or polyvariant, whether code downstream of conditional branches or switches should be dynamically compiled eagerly or lazily, and whether the dynamic code cache for each region should keep many, one, or zero code versions. A final annotation identifies program points where `invalidate()` should be inserted. DyC's annotations are described in detail elsewhere [11].

3. The Calpa System

Calpa combines program analysis and profile information to automatically derive annotations that drive selective dynamic compilation systems like DyC. These systems specialize parts of programs for particular values of run-time constants, dynamically generate the specialized code, and cache it for later reuse. They implement a variety of dynamic optimizations and techniques for caching and dispatching to the compiled code. Calpa models all these

[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 commences at arbitrary points in programs, not just at function entries. Polyvariant specialization can result in complete loop unrolling by creating a specialized copy of a loop body for each set of values of the run-time-constant loop induction variables. Complete loop unrolling is unlike unrolling done by traditional static compilers in that the 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 (which we call *single-way loop unrolling*). For more complex loops, however, one iteration may lead to several alternative 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*).

capabilities and automatically selects and places annotations to govern them.

Calpa consists of two components, an instrumentation tool and an annotation selection tool, both of which analyze C source code. The job of the instrumenter is to provide the selection tool with profile information from an application execution that will aid it in making reasonable annotation choices. It does this by automatically instrumenting its input application to track the execution frequencies of basic blocks and log the values of variables at all definitions and uses. The Instrumenter performs a pointer analysis to compute a conservative approximation to the set of variables referenced by each load and store instruction. Finally, to enable cache invalidation costs to be assessed, the instrumenter tracks the execution frequencies of store instructions and the targets they are updating.

Calpa's annotation selection tool first identifies, for each instruction in potential dynamic regions, the minimal set of static variables that cause the instruction that uses them to be static. In order to increase the number of simultaneously static computations, the selection tool combines the sets, and uses its cost/benefit model, with key parameters derived from the profile log information, to estimate the run time for each combination. If a new combination is predicted to produce faster dynamically generated code, it is retained as the current best choice. After all combinations have been considered, Calpa automatically generates the selected DyC annotations and outputs annotated C source code, which DyC compiles.

At Calpa's core is the cost/benefit model that predicts the effect of annotations on the run time of a dynamically compiled application. The model's cost function estimates the run-time overhead of dynamically generating and dispatching to specialized code. It is comprised of several subfunctions, all of whose parameters are derived from actual costs in DyC. It includes the following:

- The basic cost function accounts for specializing a region of code for a particular combination of run-time constant values. The function takes into account the frequency of static variable value changes, i.e., the number of re-dynamic compilations.
- Optimization-specific cost functions reflect the additional dynamically generated code needed to implement code-expanding optimizations, such as complete loop unrolling.
- Other cost functions account for dispatching to the correct dynamically generated code version. Calpa handles two basic dispatch models: *cache-lookup* and *invalidation-based caching*. In cache-lookup, the values of the static variables and data structures are used as a lookup key into the dynamic-code cache. If the key matches, the cached code is executed; otherwise a new version is dynamically compiled. Invalidation-based caching dispatches according to the value of the pointer that is set by the `invalidate()` operation. If the pointer contains a code address, the previously generated code version is reused, avoiding a code-cache lookup; otherwise, it will contain a pointer to the customized dynamic compiler that will generate a new version. Both models have several implementations (including one that combines them) that trade off dispatch speed, lookup key size, and frequency and cost of invalidations [8].

The benefit function predicts the execution-time savings when running the dynamically generated code. It takes into account the savings obtained by executing static instructions only once for each dynamic code version and executing the optimized, dynamically generated code instead of the original instructions. Since instructions not on the critical path of a specialized procedure are unlikely to contribute to saved cycles on a wide-issue processor, Calpa computes the critical path of the procedure to determine which instructions qualify; because it ignores instructions off the critical path, Calpa conservatively underestimates the potential benefit of executing them only once during specialization. All qualifying instructions are weighted by their latency and the frequency of their execution.

The following subsections provide more detail on Calpa's features that comprise our current prototype and on which our experimental results are based.

3.0.1. Calpa's Instrumenter

To provide the necessary data for the cost/benefit analysis, the instrumenter instruments an application to collect information about program variables and program point execution frequencies during its execution. The instrumenter inserts code into the application to monitor and summarize three kinds of information: basic block frequencies (currently the number of executions, not execution time), variable definitions, and variable uses. When a variable or data structure is accessed via a pointer, the instrumenter uses the results of its pointer analysis[1] to insert code that associates the pointer's value with a corresponding variable or data structure from its points-to set.[2] This serves two purposes: first, it attributes the use or definition of a particular value to the variable or data

[1] Calpa's current default pointer analysis uses Das' improvement [32] over Steensgaard's almost linear-time context- and flow-insensitive algorithm [23] for pointers to static and stack-allocated variables and data structures; heap-allocated data is handled by creating one distinct variable for each allocation site. In lieu of Das' algorithm the standard Steensgaard algorithm, or an extension proposed by Shapiro & Horwitz [31] can also be chosen for stack-allocated variables. In addition, Ghiya & Hendren's algorithm [7] can be run for pointers to heap-based data structures, after one of the previously mentioned points-to algorithms has been run.

```
      i = 0                     {}
L1:   if i >= size goto L2      {i,size}
      uelem = u[i]              {i,u[]}
      velem = v[i]              {i,v[]}
      t = uelem * velem         {i,u[],v[]}
      sum = sum + t             {i,sum,u[],v[]}
      i = i + 1                 {i}
      goto L1                   {}
L2:
```

Figure 2. Example of Candidate Static Variable Sets

structure that is defined or used; second, it identifies potential invalidation points of variables and data structures, so that the instrumenter can insert code to monitor the points' execution frequencies. To account for definitions that take place in library functions for which no source code is available, Calpa conservatively assumes that any variable or data structure that escapes to a library function is possibly modified by it.

An instrumented application logs values of definitions and uses as they occur, storing them as a per-invocation histogram of values and their number of occurrences. Each value-occurrence pair is tagged by the invocation number of the procedure in which it was produced (to obtain more accurate cost information for loop indices that are dependent on procedure parameters (see section 3.1.2)). Data is kept in a hash table in memory up to a user-specifiable maximum memory usage. When necessary, the values for the least recently used variables or data structures are written out to disk to make room for new values. To limit overall log size, monitoring is dynamically disabled for variables and data structures for which more than a user-specified number of distinct values have been recorded. (Because of dynamic compilation overhead, it is unlikely that these data will be run-time constants.) Before the application terminates, the profile data is written to a log file.

3.1. Calpa's Annotation Selection Tool

3.1.1 Candidate Static Variable and Candidate Division Sets

The first task in Calpa's annotation selection tool is the computation of the basic sets of variables which, if annotated as static, cause the operations that use them to become static. We call these sets *candidate static variable (CSV) sets*. The dot-product code fragment shown in Figure 2 illustrates CSV sets; next to each statement is the CSV set that makes the statement static. In general, the CSV set for an instruction is the set of variables used as source operands to the instruction, or the empty set if there are no source variable operands. For example, the comparison between `i` and `size` is static if and only if both variables are static; hence the statement's CSV set is `{i,size}`. For memory loads, the contents of the memory must also be static for the load to be static. For example, the load of `u[i]` is static if and only if both `i` and the array `u` is static, so the statement's CSV set is `{i,u[]}`, where `u[]` represents both the address `u` and its contents. Since DyC's binding time analysis will determine that instructions are static if their arguments are computed by unique static instructions, Calpa treats operands with a single reaching definition specially. In this case, the static variables for the operand are those that correspond to the CSV set of the reaching definition (instruction). For example, in the statement `t=uelem*velem`, since `uelem` and `velem` each have exactly one definition, the CSV set for this statement is `{i,u[]}` ∪ `{i,v[]} = {i,u[],v[]}`.

Calpa can build bigger dynamic regions with greater degrees of run-time optimization by merging CSV sets into larger sets of variables, called *candidate divisions* (*CDs*). The set of static instructions for a CD is the union of the instructions whose CSV sets are subsets of the CD. In general, if there are n CSV sets, there are 2^n ways to combine them. However, since CSV sets tend to overlap, not all combinations will produce different sets. For instance, if we combine `{u[],i}` and `{u[],v[],i}`, we obtain the same set as when we combine `{v[],i}` and `{u[],v[],i}`. In the dot-product example above, we get the following set of possible CDs (called the *CD set*): `{}`, `{i}`, `{i,size}`, `{i,u[]}`, `{i,v[]}`, `{i,u[],v[]}`, `{i,size,u[]}`, `{i,size,v[]}`, `{i,size,u[],v[]}`, `{i,sum,u[],v[]}`, and `{i,size,sum,u[],v[]}`.

The CD set captures all possible combinations of annotated variables that result in distinct collections of static instructions, i.e., distinct dynamic regions. Other combinations of static variables never have to be considered, since they will produce the same results as some CD already in the CD set. For instance, `{sum,i}` leads to the same instructions being static as `{i}`, which is

[2] Where these analyses determine that only one monitored variable or data structure can be accessed (a points-to set of size 1), no run-time matching is performed.

already included in the CD set. Therefore, it suffices to evaluate only the CDs in the CD set to estimate the potential benefits that specialization can achieve, rather than evaluating all 2^k possible combinations of k variables; for instance, in the dot-product example, the CD set size is only 11, whereas there are 32 sets in the power set of the five variables.

For each procedure, Calpa first computes the CSV sets of all instructions. The CD set is then enumerated, using a gradient search strategy. Variables are first sorted by their number of distinct values, obtained from the profile. Then, beginning with the CSV sets that contain the more invariant variables, new candidate divisions are generated and their run-time benefit and cost estimated. The search process remembers the best candidate division choice and its estimated speedup. The search terminates if all choices have been enumerated (feasible for small applications, such as `dotproduct`), a set time quota has expired, or the gradient of improvement over the best choice so far drops below a preset threshold.

3.1.2 The Cost Model

The DyC system incurs three different costs when dynamically compiling a program: a one-time specialization cost for producing a dynamically compiled region for particular static values, periodic dispatching costs which are paid each time a section of dynamically generated code is executed, and an invalidation check cost for variables and data structures for which invalidation-based caching is used.

Specialization Cost

The specialization cost is roughly proportional to the number of dynamic instructions generated for all code versions. Therefore, when computing its specialization cost estimate, Calpa uses the profile data to estimate the number of different code versions that will be produced. Its estimate of the total number of instructions generated at run time is proportional to the product of the number of specializations and the number of dynamic instructions generated for each.

In more detail, for each procedure and each of the procedure's choice of static variables, the following steps are performed. First, for each basic block the number of dynamic instructions `d` is computed. Then, for each static variable, `d` is multiplied by the number of the variable's profiled values `v`. If a loop induction variable is static, different versions of the loop iteration, one for each value of the induction variable, will be produced. To account for this additional generated code, a second multiplier is used for each static loop induction variable, based on the number of profiled induction variable values. For multi-way loops, in which the choice of the next loop path is determined at run time, Calpa scales `d * v`, based on the number of different paths and how often each is executed. Finally, to obtain the estimated specialization time, the estimate for the total number of instructions generated is multiplied by a constant factor. (We currently use 40 cycles per instruction generated[1]).

When loops are completely unrolled, Calpa constrains code blowup by guarding the loop-unrolling annotation with a condition that is generated by the specialization cost model. It generates the expression `d * range(i) < limit`, where `range(i)` is the actual number of different values observed for the loop induction variable `i` and `limit` is a predetermined constant (modifiable by a Calpa command-line argument) to guard the unrolling. Hence, the loop unrolling will only be performed if the number of values of `i` remains below the threshold of `limit` generated instructions. If the preset limit is too large (causing performance degradation due to cache effects) or too conservative, `limit` can be changed. Currently Calpa makes the programmer do this; however, future work will automate this process by observing the resulting cache behavior of the generated application and dynamically adjusting the limit.

Code Caching Cost

Using a simple binding-time analysis, Calpa identifies program points at which dynamically-generated-code cache lookups are necessary. However, a cost need not be incurred at all cache lookup points. For example, DyC includes a caching policy (*cache_one_unchecked)* to specify that it is safe to omit a cache lookup for invariant static variables – in this case, specialized code is generated for the variable's first encountered value, cached and used thereafter without a cache lookup. Alternatively, *cache_all_unchecked* is used for variables that step through a small sequence of values. When the cache lookup point is encountered, a new version of code is generated without a lookup. This policy is useful for complete (single-way) loop unrolling, where the induction variable is monotonically incremented.

Calpa uses profile information as a hint to decide whether an unchecked policy should be used; if a variable has only one value, it becomes a candidate for *cache_one_unchecked*; if it has only a few values and the variable is a loop index, then it may be suitable for *cache_all_unchecked*. To guarantee the safety of the *cache_one_unchecked* policy, the invalidation point analysis will ensure that the variable, once defined, will not be redefined. The *cache_all_unchecked* policy is always safe when specialization is performed on demand. Calpa can choose eager specialization if it can determine that the loop termination condition is static; if it cannot (e.g., for

[1] DyC typically takes between tens to a few hundred cycles per instruction generated, depending on the dynamic optimizations being applied [9].

Table 1. Workload

Program	Size (lines)	Description	Input
`binary`	111	binary search over an array	array of 4K entries
`dotproduct`	136	dot-product of two integer vectors	vectors of size 100 that were 90% zero-filled
`query`	226	database query for an exact match	a query with 21 comparisons
`romberg`	134	function integration by iteration	an iteration bound of 8
`dinero` (version III)	2,397	cache simulator [13]	L1 cache (8KB, 32B blocks, direct-mapped)
`pnmconvol`	333	image convolution routine that is part of the netpbm toolkit for image transformations	3×3 convolution matrix on a 2.2 MB image file
`m88ksim`	11,549	Motorola 88000 simulator, taken from the SPEC95 benchmark suite [21].	SPEC-provided null breakpoint set

multi-way loop unrolling), it chooses the safe, lazy option.

When Calpa uses the more costly *cache_all* policy (which does a cache lookup), it assesses a per-lookup fee to allocate memory for a cache key and do a hash table lookup (85 cycles in the model), plus a small additional cost to construct the cache key from the set of variables whose values must be checked in order to dispatch to the correct version of code (5 cycles per variable). Since the caching cost is paid each time the cache point is executed, it is multiplied by the execution frequency of the point, obtained from the profile.

Invalidation Cost

When Calpa uses invalidation-based caching for a variable or data structure, it first computes its invalidation points. The cost of each invalidation point is the product of its execution frequency and some fixed cost (100 cycles in the model). The sum over all invalidation points is the variable or data structure's total invalidation cost.

3.1.3 The Benefit Model

Calpa's benefit estimation identifies all instructions that are made static by the particular CD being evaluated. For each procedure and choice of static variables, it runs a simple and fast binding time analysis (a simplified version of DyC's BTA [11]) to compute the derived static variables and the division of variables and instructions into static or dynamic. (The BTA also computes the program points that require dynamically-generated-code cache lookups, important for the estimation of caching costs, described above.) In the current implementation, the BTA assumes that the variables are specialized throughout the procedure. In some cases, specializing a smaller region of the procedure may result in the same number of static instructions, but with a smaller specialization or caching cost. In such a case, the current implementation will tend to overestimate these costs.

The benefit of specializing a procedure for a particular choice of static variables is computed by estimating the number of saved cycles that will result from their instructions becoming static.[1] Since static instructions off the critical path of the specialized procedure are unlikely to contribute to saved cycle time, Calpa first computes the critical path of the procedure. The total number of cycles saved is obtained by multiplying the latency of each static instruction on the critical path by its profile-derived execution frequency.

4. Experiments and Results

Calpa's instrumentation and annotation selection tools are implemented using the SUIF compiler infrastructure [26], together with the Machine SUIF libraries for CFG-construction and data flow analysis [14]. All analysis, instrumentation, and insertion of annotations are carried out on the SUIF intermediate representation (IR). Processed IR is converted back to C code and then compiled with the GNU gcc compiler (for the instrumented application) or DyC (for the annotated application).

Our experiments profile and automatically annotate a number of commonly used kernels and applications, described in Table 1. In this initial study, the same inputs were used for profiling and the final performance runs that determine application speedup. Studying the variance of value profiles across different inputs and the effects of varying levels of precision of different alias analysis algorithms is beyond the scope of this paper; we will analyze Calpa's sensitivity to these factors once we finish calibrating our cost/benefit model. All experiments were done on a lightly loaded DEC Alpha 21164 workstation with 1.5GB of physical memory. Timings were obtained by using the UNIX `time` command; reported times are wall clock time.

[1] We use instruction latencies, and currently assume L1 cache hits, again a conservative estimate.

Table 2. Profiling Results. This table shows the effects of instrumentation on application code size and execution time, and the resulting profile log size.

Program	Instrumentation Time	Original Binary Size	Instrumented Binary Size [a]	Binary Expansion Factor	Original Run Time (seconds)	Instrumented Run Time	Profile Log File Size
`binary`	0.2 seconds	25 KB	224 KB	9.0	< 0.1	1.9 seconds	275 KB
`dotproduct`	0.1 seconds	25 KB	224 KB	9.0	< 0.1	0.3 seconds	92 KB
`query`	0.4 seconds	27 KB	224 KB	8.3	< 0.1	7.8 seconds	939 KB
`romberg`	0.3 seconds	26 KB	224 KB	8.6	< 0.1	0.4 seconds	102 KB
`dinero`	4.6 seconds	57 KB	448 KB	7.9	1.3	13.8 minutes	6.8 MB
`pnmconvol`	1.2 seconds	66 KB	288 KB	4.4	3.0	17.1 minutes	266 KB
`m88ksim`	10.7 minutes	213 KB	2.6 MB	12.6	180.1	3.5 hours[b]	7.5 MB

a.Instrumented binary sizes include required portions of a 409KB instrumentation library.
b.Using a limited amout of binary patching improved m88ksim's instrumented run time from 18 to 3.5 hours.

4.1. Instrumentation and Profiling Results

To assess the viability of our profiling tool, we measured (1) the time to instrument the applications, (2) the impact of instrumentation on the size and execution time of the applications, and (3) the size of the resulting log files. The results are summarized in Table 2.

Instrumenting the source code was fast, ranging from fractions of a second for the kernels to 10 minutes for `m88ksim`.

Instrumenting programs increased their size by roughly an order of magnitude. For most of the applications, this increase is predominantly due to including portions of Calpa's instrumentation library. Making this a dynamically linked library and removing its debugging support would greatly reduce the code space cost of instrumentation.

The run time of the instrumented executables was 1 to 3 orders of magnitude slower than that of the original code. In contrast, Calder et al. [4] report average slowdowns in the range of 10× to 33× for their profiling schemes. Calpa's much slower profiling performance is caused by its straightforward tracking of *all* variables, at all definition and use points. While Calpa currently disables tracking for variables with more than a thousand values, a much lower cut-off point seems more appropriate, i.e., the costs of *re*dynamic compiling for each of a thousand values will surely swamp the gains of executing the specialized code. In addition, simple static analysis could be applied to reduce the number of redundant points of instrumentation, e.g., attributing dominated use counts to the instrumented, dominating use in the same control region (assuming no intervening definition), and binary patching schemes could be used to eliminate, for example, cut-off-point checking. Both techniques would reduce the costs associated with over-profiling variables [33]; for instance, binary patching of disabled tracking calls alone improved the run time from 18 to 3.5 hours for m88ksim. However, despite its straightforward instrumentation approach, Calpa's instrumented run times were a matter of seconds or minutes for the kernels and medium-sized applications, and hours for m88ksim.

Monitoring and storing data for all variables produced profile log files that ranged from 92KB to 275KB for the kernels and 266KB to 7.5 MB for the applications. Summarizing profile data in memory, rather than simply saving values to a file as they stream out of the instrumented program, turned out to be a good time-space trade-off; in particular, high repeat counts for value uses indicate that a non-summarizing profile log would be much larger.

4.2. Annotation Generation Results

Based on the profile data, Calpa generated annotations for all programs. We measured the execution time of the annotation selection tool and compared the set of Calpa's annotated variables to those chosen by a programmer. The results are shown in Table 2.

For all programs Calpa generated the same set of annotations that had been produced by the manual methodology. In addition, despite the current simple functionality of the profiling tool and the coarse-grain estimates of the analysis cost/benefit model, Calpa occasionally annotated other variables. The additional annotations increased the speedups of the dynamically compiled programs over their statically compiled versions. Although preliminary, the results demonstrate the promise for automatic dynamic compilation based on the approach taken by Calpa. A discussion of the individual programs follows.

- In `binary`, the procedure search is called repeatedly for the same array of values. We had manually

Table 3. Application & Annotation Characteristics. This table shows, for each program, the variables that were automatically annotated, and the selection tool annotation time. The variables in **bold** are those that Calpa annotated, but the human missed.

Program	Annotated Static Variables	Annotation Time
binary	size & contents of the input array induction variable for the search loop **the search key**	6 seconds
dotproduct	the contents of vector `u` the loop index and duration bound **the contents of vector `v`**	2 seconds
query	a query	15 seconds
romberg	the iteration bound	26 seconds
dinero	cache configuration parameters	27 minutes
pnmconvol	4 loop indices in two doubly-nested loops 3 color arrays image format flag maximum size of the image array	75 seconds
m88ksim	an array of breakpoints loop index flag that indicates whether breakpoint-checking is enabled	8.0 hours

annotated as static this array, its size and the loop induction variables used to search it. This resulted in a complete unrolling of the search loop. Calpa identified the same variables for annotation, and, in addition, decided to make the search key static. The driver routine that calls search uses only 3 different key values to do the search, making the search key appear quasi-invariant, an artifact of this particular use of the search routine by the driver. Therefore, Calpa chose to specialize for the search key values thereby increasing binary's speedup from 2.3 to 3.1.

- For `dotproduct` Calpa also generated the same annotations as had been done manually (vector u). In addition, it annotated the second vector v, which was also constant, an artifact of the use of the dotproduct procedure by the driver routine. Annotating both vectors instead of one, improved the speedup from 6.6 to 22.6.
- For the remaining programs, the set of variables automatically annotated by Calpa exactly matched the manual annotations. Speedups for those programs were 1.4 (`query`), 1.2 (romberg), 1.5 (`dinero`), 1.1 (`m88ksim`), and 3.0 (`pnmconvol`).

The column named "annotation time" shows the time to generate annotations from the profile data logs. It took only a few seconds to generate the annotations for the kernels, and minutes for most applications (the exception, m88ksim, took eight hours).

5. Related Work

Calder *et al*. [3] were the first to expose quasi-invariant behavior by profiling. They then went on to show [4] that the values found during profiling could be used to potentially guide automated optimization by demonstrating via hand optimization that two codes could get substantial benefit from using the value profiles. Their value profiler identifies (quasi-) invariant variables and their top `n` values; two metrics are defined to measure the invariance of variables, and a cache is used during profiling to store the values. They do not collect information about particular sequences of variable values, which we need to compute precise caching and specialization costs. An advantage of their tool is that no recompilation of a program is necessary in order to profile it, since they instrument the executable using ATOM [22]. However, to exploit the invariance information, a mapping of the instruction-level information back to the source code is necessary.

Value prediction [18] is a hardware technique that is complementary to our compiler- and profiler-based approach of value-specific optimizations. It uses processor hardware to predict instruction results and speculatively executes subsequent data-dependent instructions, based on the predicted values.

Dynamo [2] is a run-time optimizer that tries to improve performance by identifying frequently taken paths (traces) through a program. Speedup results from accumulating the traces in a code cache and executing the streamlined code instead of the original code with branches. To obtain control over an application's execution and monitor its performance, execution starts in Dynamo, which interprets native code until it finds a hot path. Hot paths execute from the code cache and return control to Dynamo when finished. When a path is no longer hot, the code cache is flushed and monitoring and interpretation of the code is resumed. If too much time is spent in interpretation mode, Dynamo bails out to native execution. Because of bail-out, Dynamo can limit the maximum slowdown an application may suffer. Since Dynamo tries to speed up an application in executable format, its approach is complimentary to Calpa's. Applications annotated by Calpa and compiled by DyC could be run under Dynamo, which might further improve their performance.

While there is a large body of research on dynamic compilation, only a small fraction relates to automation. Autrey and Wolfe [1] proposed a loop-level analysis to identify variables that are modified much less frequently than they are referenced, which they call *glacial variables*. Variables that are defined at loop nesting level `n`, and not modified at any higher nesting levels are identified as candidates for dynamic compilation at loop nesting level `n`. They do not report results of applying their analysis in a real dynamic compilation system. TypeGuard and MemGuard are two tools used in the Synthetix project [24] to identify where invariant values are modified. In TypeGuard the programmer tags fields in C struct types with a *guard specification*. TypeGuard then analyzes the C program and identifies all places where the tagged fields are potentially modified. This information is used by the Synthetix system to trigger respecialization for the field's new value. However, due to pointer type casts, TypeGuard can not safely identify all potential definition points. In addition, since struct types rather than particular variables are guarded, scalar variables cannot be handled, because too many (mostly spurious) messages would be generated. MemGuard puts a static data structure into a protected memory page. On a write to the page, the page fault handler triggers respecialization for the new values in the data structure. The MemGuard approach requires a modification to the operating system, something we did not want to do.

IPERF [15] is a framework for the automatic construction of performance prediction models. It uses a database of performance models of computation, the memory hierarchy, and virtual address translation and tries to fit a linear combination of these as closely as possible to observed performance. In its model of computation it distinguishes only between different compiler optimization levels (e.g., `-O2`); in contrast, Calpa takes both the estimated benefit from dynamically compiling code and the compilation cost into account when deciding whether to use dynamic compilation. Wang [25] proposed a framework for the performance prediction of superscalar-based computers to guide the optimization in the PTRAN2 compiler for High Performance Fortran. He reported results for straight-line code only (no loops or other control structures). Dean *et al*. [6] estimated the benefit of function inlining in an object-oriented language by performing the inlining and subsequent optimization in a trial; their approach was able to amortize the trial cost by reusing the benefit estimate for call sites whose receiver arguments belonged to the same type groups. Calpa's approach of specializing conditionally, using conditions that are based on variable values and architecture-specific parameters, such as I-cache size, is similar to Debray's concept of resource-bounded partial evaluation [34], where a cost-benefit model is used to prevent a partial evaluator from specializing too aggressively, e.g., producing too much code for the I-cache.

6. Conclusions

In selective dynamic compilation systems like DyC, finding the right annotations is a major challenge to profitable run-time optimization. Sometimes several person-weeks are spent in a tedious, trial-and-error process until successful annotations are found. We have shown that Calpa can quickly produce annotations for small- and medium-sized programs, using its unique combination of techniques, and its particular choice of parameters (for example, cutoff limits for value profiling). While the current combination provides useful value, it represents only one particular point the design space. Our next step is to explore this design space by varying Calpa's parameters and examining the trade-offs of different choices, in particular, studying the sensitivity of Calpa results with respect to different value profiles, alias algorithms, and cut-off limits. With that study, we hope to come closer to the long-term goal of making dynamic compilation just another in a series of optimizations performed automatically by compilers.

Acknowledgments

We'd like to thank Brian Grant, Matthai Philipose and our anonymous reviewers for valuable comments on an earlier draft, and Mike Smith and Glenn Holloway for Machine SUIF source and technical help in using it. This work was supported by ONR contract N00014-96-1-0402, NSF grant CCR-9503741, and NSF Young Investigator Award CCR-9457767.

References

[1] T. Autrey and M. Wolfe. Initial results for glacial variable analysis. In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 120–134, August 1996.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, June 2000.

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

[4] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, 1:1–37, March 1999.

[5] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Symposium on Principles of Programming Languages*, pages 145–156, January 1996.

[6] J. Dean and C. Chambers. Towards better inlining decisions using inlining trials. In *Proceedings of the ACM Conference on LISP and Functional Programming '94*, pages 273–282, June 1994.

[7] R. Ghiya and L.J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 24(6):547–578, December 1996.

[8] B. Grant, C. Chambers, and S.J. Eggers. Efficiently dispatching to run-time specialized code. Submitted for publication.

[9] B. Grant, M. Mock, M. Philipose, C. Chambers, and S.J. Eggers. The benefits and costs of DyC's run-time optimizations. Submitted for publication.

[10] B. Grant, M. Mock, M. Philipose, C. Chambers, and S.J. Eggers. Annotation-directed run-time specialization in C. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 163–178, June 1997.

[11] B. Grant, M. Mock, M. Philipose, C. Chambers, and S.J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1-2):147–199, October 2000.

[12] B. Grant, M. Philipose, M. Mock, C. Chambers, and S.J. Eggers. An evaluation of staged, run-time optimizations in DyC. In *Conference on Programming Language Design and Implementation*, pages 293–304, May 1999.

[13] M.D. Hill and A.J. Smith. Experimental evaluation of on-chip microprocessor cache memories. In *Proceedings of the International Symposium of Computer Architecture*, pages 158–166, June 1984.

[14] G. Holloway and C. Young. The flow and analysis libraries of machine SUIF. In *Proceedings of the 2nd SUIF Compiler Workshop*, August 1997.

[15] C-H. Hsu and U. Kremer. A framework for automatic construction of performance predication models. In *Proceedings of the 1st Workshop on Feedback-Directed Optimization*, October 1998.

[16] N.D. Jones, C.K. Gomarde, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[17] M. Leone and P. Lee. Optimizing ML with run-time code generation. In *Conference on Programming Language Design and Implementation*, pages 137–148, May 1996.

[18] M.H. Lipasti, C.V. Wilkerson, and J.P. Shen. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, October 1996.

[19] M. Mock, M. Berryman, C. Chambers, and S.J. Eggers. Calpa: A tool for automating dynamic compilation. In *2nd Workshop on Feedback-Directed Optimization*, November 1999.

[20] 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*, pages 132–142, May 1998.

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

[22] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. *SIGPLAN Notices*, 29(6):196–205, June 1994. Conference on Programming Language Design and Implementation.

[23] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[24] SYNTHETIX TOOLKIT. http://www.cse.ogi.edu/projects/synthetix/toolkit/.

[25] Ko-Yang Wang. Precise compile-time performance prediction for superscalar-based computers. In *Conference on Programming Language Design and Implementation*, pages 73–84, June 1994.

[26] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M.S. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12), December 1994.

[27] L.P. Deutsch and A.M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Conference Record of POPL '84: Symposium on Principles of Programming Languages*, pages 297-302, January 1984.

[28] C. Chambers and D. Ungar. Making Pure Object-Oriented Languages Practical. In *Proceedings OOPSLA '91*, pages 1-15, November 1991.

[29] U. Holzle and D. 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.

[30] V.C. Sreedhar and M. Burke and J.-D. Choi. A Framework for Interprocedural Optimization in the Presence of Dynamic Class Loading. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 208-218, June 2000.

[31] M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. In *Conference Record of POPL '97: Symposium on Principles of Programming Languages*, January 1997.

[32] M. Das. Unification-Based Pointer Analysis with Directional Assignments. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 35-46, June 2000.

[33] O. Traub and S. Schechter and M.D. Smith. Ephemeral Instrumentation for Lightweight Program Profiling. Technical report, Harvard University, 2000.

[34] S.K. Debray. Unfold/Fold Transformations and Loop Optimization of Logic Programs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 297-307, June 1988.