Calpa: A Tool for Automating Dynamic Compilation

Markus Mock, Mark Berryman, Craig Chambers and Susan J. Eggers

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

Abstract

Declarative dynamic compilation systems are driven by user annotations that identify runtime constants. Although manually inserting annotations can achieve significant program speedups, it is a tedious and time-consuming process that requires careful inspection of a program's static characteristics and runtime behavior and much trial and error in order to craft beneficial annotations. Calpa is a system that generates annotations automatically for the DyC dynamic compiler. Calpa utilizes execution frequency and value profile information to drive a program analysis based on a (dynamic compilation) cost/ (dynamically generated code) benefit model to choose runtime constants and DyC's specialization policies. Although only a subset of Calpa's eventual capabilities has been implemented, for the programs tested so far, it 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.

1. Introduction

Dynamic compilation[1] transforms programs at run time, using information available only at run time, to optimize them more fully than statically compiled code. Dynamic compilers usually base their optimizations on runtime-computed values of particular variables and data structures (called runtime constants). Since dynamic compilation involves additional overheads during program execution that statically compiled programs don't incur, in order for the dynamically generated code to execute faster, the values of the runtime constants must remain invariant or quasi-invariant, and the code that uses them must be executed frequently.

Today's dynamic compilation systems choose runtime constants manually. In the best cases, programmers profile their code to determine changes in variable values and execution-time frequencies of basic blocks or procedures; in other cases they rely on their knowledge of how an application works to pick the runtime constants. Once chosen, the variables are usually made known to the dynamic compilation system through user annotations that trigger the value-specific analyses and transformations.

Tempo [5,16], Fabius [14], and DyC [7,8,9,10] are all dynamic compilation systems that rely on programmer hints, *e.g.*, annotations, to drive their dynamic optimizations. To runtime optimize a procedure in Tempo or Fabius, the user annotates its runtime-constant arguments by marking them as "static". DyC provides finer-grained dynamic optimization, by allowing users to annotate individual variables at arbitrary points within a procedure; the variables will be treated as runtime constants up to the end of their procedure, or until another annotation returns them to a dynamic status. DyC also has a set of policy annotations to regulate the aggressiveness of specialization and certain dynamic compilation costs.

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 compilation. Our experience with DyC serves as a case in point. To annotate the applications for our initial evaluation of DyC's runtime optimizations [8], 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 by *manually* inserting code into the applications, and then 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. Since DyC's policies control the aggressiveness of specialization and the cost of runtime compilation, choosing correct policies turned out to be laborious and time-consuming as well. All in all, although DyC's annotations are few and simple, finding good candidates for runtime-constants and picking appropriate policies turned out to be a very tedious process, taking us several weeks to annotate the applications for that study.

Calpa[2] is a system that combines profile information and program analysis to automatically derive the annotations that drive dynamic compilation in DyC. The profiler is a stand-alone program that automatically instruments an application to produce values and frequency data when it is executed. Calpa's analysis is also (currently) a separate module that contains a (dynamic compilation) cost/ (dynamically generated code) benefit model that predicts the effect of particular annotations on the run time of an application. It first identifies sets of variables, which, when annotated together, make the computations that use them static. In order to increase the number of simultaneously static computations, it then combines the sets, and uses the cost/benefit model to test the runtime effect of each combination. A new combination is only retained if the model predicts that it will produce faster dynamically generated code. The cost/benefit model consults information that is generated by the profiling tool, communicating through an interface, that, for instance, tells it that a particular variable was invariant during execution. The search terminates if all possible choices are exhausted, a preset time quota is up, or the gradient of improvement over the best solution found so far drops below a minimum threshold.

[1] We are referring to systems in which the analysis for dynamic optimization is staged across static compile and execution times. JITs do all analysis at runtime.
[2] Calpa was an important Inca oracle ritual, which was performed before making important decisions.

To use Calpa, a programmer first runs its profiling tool to automatically instrument an application, and then executes the instrumented program on some representative input. The results from the profiling run and the applications's original C source are then analyzed by Calpa's program analysis tool, which automatically produces annotated C code for the application. DyC then analyzes and optimizes the application, as though it had been annotated manually.

Using our first prototype of Calpa, we derived annotations for a set of small but common programs. The entire profiling and annotation process was completed for these programs within several minutes (one larger program required just under twenty minutes for instrumentation). Calpa produced all the annotations that we had identified with our manual process, and, in addition, for some programs, inserted others that improved their performance.

This paper presents Calpa's approach to automating dynamic compilation in DyC. We describe the different components of Calpa: our program analysis, the runtime profiling system, and the interface between them. We also report our initial performance results, including the time and space it took to profile the applications and to derive the annotations.

Calpa is a work in progress: our initial prototype includes a value and frequency profiler and most of the analysis. We plan to augment the profiler with an alias and heap analysis that will produce more precise information about the quasi-invariance of variables and data structures, and to make the cost/benefit model more detailed. This additional functionality will then enable us to tackle the larger applications that DyC can currently dynamically compile.

In the next section we summarize Calpa's targeted dynamic compilation system, DyC, and its annotations. Section 3 describes Calpa; it contains an overview of the Calpa system, a discussion of its analyses and cost/benefit estimation, the Calpa profiler and the interface between the analyses and the profiling tool. Section 4 describes the experiments we carried out to assess the effectiveness and the resource-use of Calpa. Section 5 discusses how we envision extending Calpa to annotate applications that our current implementation cannot yet handle. Section 6 discusses related work, and, finally, section 7 concludes.

2. DyC

2.1 System Overview

DyC is an annotation-driven, staged, dynamic compilation system that attains speedups on a group of medium-sized (< 15K lines of code) C programs that range up to 4.6 [8]. To produce this level of performance, DyC contains (1) a sophisticated form of partial-evaluation binding time analysis (BTA) that supports polyvariant specialization (enabling both single-way and multi-way complete loop unrolling), polyvariant division[1], static loads and static calls, (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* (variables that have a single value, or relatively few values, during program execution, *i.e.*, *runtime constants*) on which many calculations depend. (In addition to identifying static variables, programmers can specify policies that govern the aggressiveness of specialization and the cost of caching dynamically compiled code, enabling them to get finer control over the dynamic compilation process.) DyC's binding time analysis (BTA) automatically identifies those variables (called *derived static variables*), whose values are computed from annotated or other derived static variables and determines which parts of the program downstream of the annotations can be dynamically optimized (we call these *dynamic regions*). 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 BTA is program-point-specific and flow-sensitive: a dynamic region can start and stop at any program point (a dynamic region may have multiple exits), and a variable may be static at some program points and not at others.

DyC builds a custom dynamic compiler (also called a *generating extension* [13]) for each dynamic region 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, requiring neither an intermediate representation nor iterative analyses at run time.

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 current 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 runtime overhead.

[1] Polyvariant division allows the same piece of code to be analyzed with different combinations of variables being treated as runtime 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 runtime-constant variables. Program-point-specific polyvariance enables polyvariance to arise 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 static loop induction variables. Complete loop unrolling is unlike unrolling done by traditional static compilers in that the unrolled loop is eliminated rather than enlarged. For simple loops, such as those that merely increment a counter until an exit condition is reached, a linear chain of unrolled loop bodies results (we call this *single-way loop unrolling*). For more complex loops, however, one iteration may lead to several alternative loop iterations (*e.g.*, if 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 (we call this *multi-way loop unrolling*).

2.2 DyC's Annotations

`make_static` and `make_dynamic`

The basic annotations that drive runtime specialization in DyC are `make_static` and `make_dynamic`. `make_static` takes a list of variables, each of which is treated as a runtime constant at all subsequent program points, until DyC reaches either a `make_dynamic` annotation that lists the variable or the end of the variable's scope (which acts as an implicit `make_dynamic`). (This region of code is a *dynamic region*.)

The @ Annotation

By default, DyC assumes that the contents of data structures, even if referenced through a runtime-constant or compile-time-constant address, are dynamic. In many programs, however, at least some of these contents are invariant. In such programs, loads of invariant parts of static structures can be treated as static computations, done once as part of dynamic compilation. DyC allows programmers to annotate such loads as static, by prefixing them with the `@` symbol (for example, `t = @* p;`).

The `@` prefix is a potentially unsafe programmer assertion. Calpa's current profiler can determine whether a particular load always returns the same value during profiling; and the full Calpa system will use alias, heap, and side-effect analyses to ensure that this will always be the case, regardless of program input.

Similarly, the `@` symbol can annotate pure functions[1] 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.

Policies

Each variable listed in a `make_static` annotation can have an associated list of policies. These policies control the aggressiveness of polyvariant specialization and division, and the dynamically-generated-code caching behavior[2]. All of DyC's default policies are safe; unsafe policies, whose use can lead to changes in observable program behavior if their underlying assumptions are violated, are included to enable sophisticated users or tools, such as Calpa, to have finer control over dynamic compilation in order to obtain better performance.

The polyvariant vs. monovariant division policy controls whether control flow merge points should be specialized for a variable that may not be static along all incoming paths. Similarly, the polyvariant vs. monovariant specialization policy controls whether merge points should be specialized for different values of a variable that flow in along different incoming paths. Additional policies determine whether dynamically optimized code generation should be done ahead of time (eagerly) or just before it is about to be executed (on demand).

The cache policies govern how the runtime specializer caches and re-uses dynamically generated code. Each policy controls how many specialized versions of code are cached (one vs. all), and whether the current values of the static variables are used to determine which cached version to use (checked vs. unchecked).

A detailed discussion of all of DyC's annotations, the tradeoffs involved in their selection, and their effects on code generation, can be found in [7].

3. The Calpa System

Calpa Overview

Calpa combines program analysis and profile information to automatically derive the annotations that drive dynamic compilation in DyC. The Calpa system consists of two components: an instrumentation tool and a program analysis, shown in Figure 1. The programmer uses the instrumentation tool to instrument the C source code of an application, and then compiles the instrumented application with a standard C compiler.The instrumented application is executed on some representative input, producing a profile log of runtime information. Calpa's analysis tool uses this profile information and its static analyses to automatically generate DyC annotations and outputs annotated C source code. Finally, DyC compiles the automatically annotated C code, just as it would a manually annotated program.

Calpa uses its cost/benefit model to predict the effect of annotations on the run time of the application. The model's cost function estimates DyC's execution-time overhead (in cycles) of dynamically generating programs and the runtime checks of the code cache; the benefit function predicts the execution-time savings (also in cycles) when running the dynamically generated code.

The Calpa analysis tool (see Figure 2) first identifies sets of variables, which, when annotated together, make the computations that use them static. In order to increase the number of simultaneously static computations, it then combines the sets, and uses the cost/benefit model to estimate the runtime for the combination. If the new combination is predicted to produce faster dynamically generated code, it is retained as the current best choice. The cost/benefit model consults information generated by the profiling tool, communicating through an interface, that, for instance, tells the analysis that a particular variable was invariant during execution. The generation of new combinations terminates if either all possible choices are exhausted, a preset time quota is up, or the gradient of improvement over the currently best solution drops below a minimum threshold.

Calpa's profiler and analysis tool are implemented using the SUIF compiler infrastructure [19], together with the Machine SUIF

[1] Pure functions are functions that return the same value when called with the same arguments and have no side-effects.
[2] Actually, DyC uses two sets of caching policies, one used at entry points of dynamic regions, one at control flow merges, providing even finer control over caching decisions. Details can be found in [7].

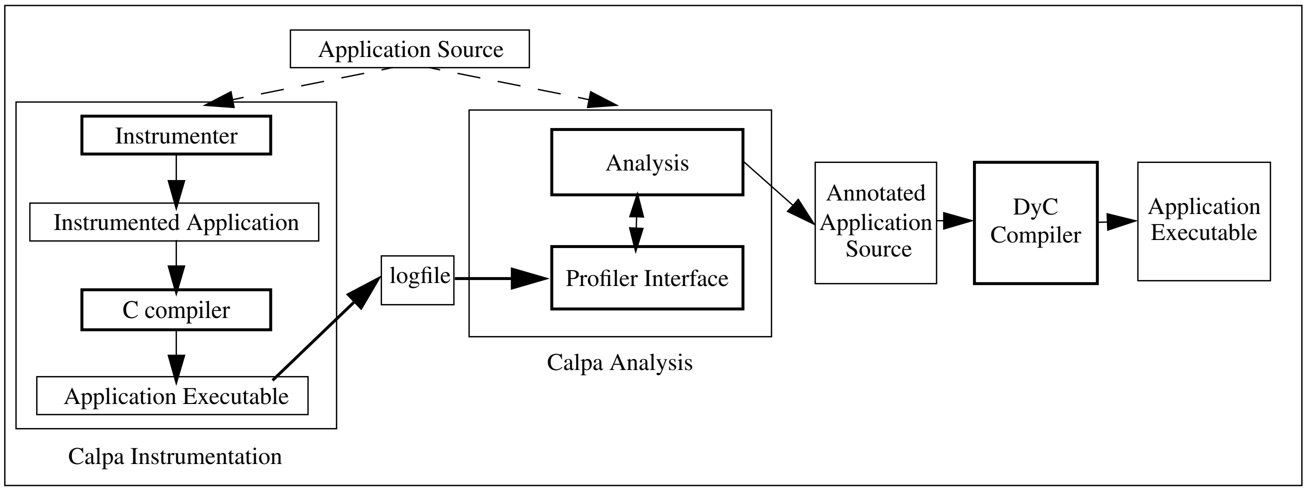

Figure 1. The Calpa System

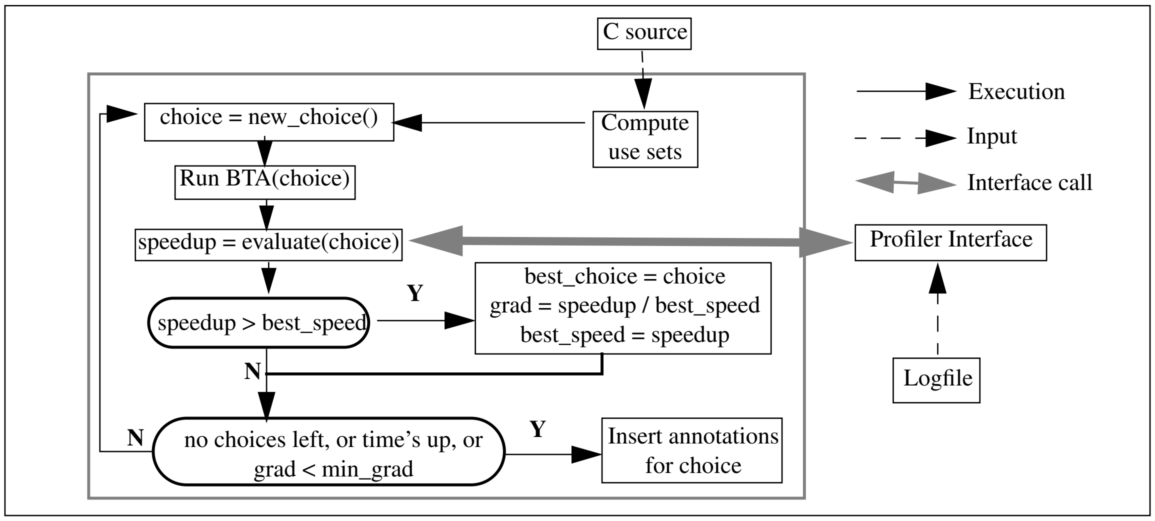

Figure 2. The Calpa Analysis Tool

libraries for CFG-construction and data flow analysis [11]. 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 DEC's Alpha C compiler cc (the instrumented application) or DyC (the annotated application).

3.1 Calpa's Profiler

To provide the necessary data for the cost/benefit analysis, the profiler collects information about program variables during execution. The profiler monitors three basic sets of information: basic block execution frequency (number of executions, not execution time), variable definitions, and variable uses. In addition, alias and heap analyses identify potential definition points of variables or data structures. Where these analyses cannot definitely determine that a monitored variable or data structure will be written, they identify program points where DyC needs to insert invalidation checks to trigger a re-specialization, should Calpa have decided to make the monitored variable or data structure a runtime constant. The frequency of these invalidation checks, and of the invalidation itself, is also measured by the profiler, supplying the cost model with essential information to compute the caching and specialization costs.

The profiler is implemented as a stand-alone pass in the Stanford SUIF Compiler. It linearly processes the intermediate representation (IR) instructions that represent a piece of source code. While analyzing this IR, it transforms the code by adding instructions that capture the desired data for a program's variables. When the instrumented code is re-compiled and executed, this data is written out to log files as the program executes. If file size is a concern, the output can be compressed (currently implemented by writing through a pipe to the GNU gzip utility).

The basic instrumentation tool has been implemented. We are currently integrating the alias analysis (Steensgaard's context- and flow-insensitive algorithm [18]) and heap analysis (Ghiya & Hendren's approach of identifying distinct data structures in the heap [6]), so that it can be used both by the Calpa annotation analysis and the profiling tool. The destination addresses of assignments are already monitored by the profiler, enabling it to observe the redefinition frequency of definitely unaliased variables (local non-pointer variables that never have their address taken), and to obtain a lower bound for possibly aliased variables. Once we have integrated the alias and heap analyses, we'll be able to support *invalidation-based caching* in DyC, further discussed in section 5.

3.2 Calpa's Analysis Tool

3.2.1 Use and Variety Sets

The first task in Calpa's analysis is the computation of sets of variables that are used together in the program, such that, when they are annotated together, the operations that use them will become static; we call these sets *use sets*. To illustrate use sets, consider the code fragment shown in Figure 3; next to each statement is the use set that makes the statement static. For example, the comparison between `i` and `size` is static, if and only if both `i` and `size` are static; the use set is `{i,size}`. The load of `u[i]` is static, if and only if both `i` and the array (contents) at address `u` is static, so the use set here is `{i,u}` (analogously for `velem = v[i]`). Since `uelem` and `velem` have exactly one definition, to make `uelem` and `velem` static, it suffices to make the variables that define them static (`{i,u}` and `{i,v}`, respectively). Hence, this use set is `{i,u,v}`. For `sum=sum+t` to be static, both `sum` and the variables that determine `t` need to be static, so the use set is `{i,sum,u,v}`. And finally, for `i=i+1` to be static, only `i` needs to be static (use set is `{i}`).

When use sets are combined, the combination of their associated instructions becomes static. In general, if there are `n` use sets, there are 2^n ways to combine them. However, typically 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}`. We call the set of sets that results from combining all use sets a *variety set*. In the example we get the following variety 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}, {i,size,sum,u,v}}`.

A variety set summarizes the choices of static variables that result in distinct sets of static instructions, *i.e.*, distinct benefits from dynamic compilation. Other combinations never have to be considered, since they will produce the same benefit as a set in the variety set. For instance, `{sum,i}` leads to the same instructions being static as `{i}`, which is represented in the variety set. Therefore, it suffices to evaluate the sets in the variety 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 example, the variety 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 use sets of all instructions; each use set is placed as an element into a set of use sets, called a *generator set*. The variety set is then constructed lazily from the generator set in a generate- (a new combination of sets) and-test (its potential performance benefits) loop that estimates the benefit and cost of each choice. For small applications, such as `dotproduct`, all choices can be enumerated, but for larger applications, the variety set grows too large. For these programs only a subset of the members of the variety set are generated, in an order that gives preference to sets with variables that are invariant (and consequently have lower specialization cost) over sets with variables with many different values.

3.2.2 Calpa's BTA

Calpa currently analyzes a program procedure by procedure, *i.e.*, no interprocedural analysis is performed[1]. For each procedure and choice of static variables, it runs a simple and fast binding time analysis (similar to Anderson's Cmix [1]) to compute the division of variables and instructions into static or dynamic. It also computes the program points that require dynamically-generated-code cache checks, important for the estimation of caching costs in the cost model. The BTA is implemented as an iterative data flow analysis over the control-flow graph of the procedure. 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.

3.2.3 Benefit Estimation

Calpa's current benefit (and cost) estimation is very simplistic and is only a first approximation of what we plan to have in the full system; we are refining the model as the implementation progresses, eliminating approaches that turn out to be too crude to produce good

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

Figure 3. Example of Use Sets

[1] The alias and heap analysis are interprocedural, though.

results.

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. In the current implementation, the number of saved cycles for each basic block is approximated by summing the latencies of its static instructions (represented in SUIF's lowest IR which matches machine instructions fairly well). The total latency is then multiplied by the execution frequency of the basic block. For single-instruction-issue processors this gives a fairly accurate estimate of the number of saved cycles (barring memory subsystem stalls). However, for today's multiple-instruction-issue processors, this figure is an overestimation. For these processors, execution time of a basic block (or larger unit) is largely determined by its critical path length; if instructions not on the critical path are eliminated, no reduction in execution time will occur. In the full Calpa system, we will use a more sophisticated estimator for saved machine cycles, that will take the critical path into account. Since full loop unrolling leads to an increase in code size which may overwhelm the processor's instruction cache, we plan to incorporate information about cache size into the model to avoid degrading performance by producing too much code.

3.2.4 Cost Estimation

The DyC system incurs two different costs when dynamically compiling a program: the periodic caching cost which is paid each time a cache point is executed, and a one-time specialization cost for producing a dynamically compiled region for particular static values.

The Specialization Cost

As discussed above, DyC produces and caches different versions of dynamically compiled code, each specialized for different values of static variables (this is polyvariant specialization). The number of versions produced depends on how many different values are encountered during execution. Calpa uses its 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 in the procedure.

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. If loop induction variables are specialized, different versions of the loop iteration, one for each value of the loop 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. 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].)

Caching Cost

Calpa's BTA identifies program points at which dynamically-generated-code cache checks are necessary. However, a cache cost does not have to be paid at all cache check points. For example, DyC allows the programmer to specify that it is safe to omit a cache check (we call this `cache_one_unchecked`) for invariant static variables. Then, for the variable's first encountered value, specialized code will be generated, cached and used thereafter without a cache check. Alternatively, a `cache_all_unchecked` policy is used for variables that step through a small sequence of (not necessarily distinct) values. When the cache check point is encountered, a new version of code is generated without a check. 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 for `cache_all_unchecked`. To guarantee the safety of the `cache_one_unchecked` policy, the invalidation point analysis ensures that the variable, once defined, will not be redefined.

The `cache_all_unchecked` policy is always safe when specialization is performed on demand; it may however, produce too much code if the profile run (using one input) produces a smaller sequence of values than the actual execution run (using a different input). Since the invalidation analysis has not yet been implemented, Calpa currently issues a warning when it chooses a potentially unsafe annotation; a Calpa switch can then force a safe default. (We used unsafe policies to generate the results in this paper, since we were able to verify the safety of unchecked annotations by inspection of the programs.)

When Calpa uses the safe, but more costly policy for a variable, it assesses a per-lookup fee to allocate memory for a cache key data structure and do a hash table lookup (85 cycles in the model), plus a small additional cost per variable (5 cycles) 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. (This is a first approximation of the actual behavior of cache checks in DyC; a future version of Calpa will use a more detailed breakdown of caching costs as identified in [10].) 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.

3.3 The Profiler-Analysis Interface Functions

The Calpa prototype was designed to do little analysis in the profiler itself, to ensure that a large set of queries could be answered without reexecuting the instrumented application. Instead, the profiler simply captures information about execution frequencies (for each basic block) and values of variables and data structures (at each definition and use point) and saves it to a profiling log. As profile logs get larger, longer sequences of values have to be processed by the interface functions, which takes more time. To save I/O time, the analysis

[1] DyC typically takes between tens to a few hundred cycles per instruction generated [10].

```
1. long frequency(program_point p)
2. long frequency(function_id f)
3. pattern_list pattern(var_id v)
4. pattern_list pattern_per_call(var_id v, int k)
5. pattern_list offsets(var_id v)
6. pattern_list index_pattern(var_id v, int index)
```

Figure 4. Interface Functions

issues a particular query to the interface only once, and memoizes the result for its subsequent uses.

The cost/benefit estimation components of Calpa communicate with the profile log through a set of interface functions, shown in Figure 4. The first two functions return the number of times program point `p` and function `f` were executed; this information is used when computing caching costs and estimating the benefit from static instructions. The second pair of functions returns a list of values that were successively assigned to variable `v` during call number `k`; this data is used to compute the specialization cost for `v` and the generation order of specialization choices in the variety set (sets of variables with high invariance, *i.e.*, a small number of values, are generated first). Functions 5 and 6 obtain the list of indices that were used to load values from an array variable `v` and the list of values loaded. Both are used to calculate the specialization cost of array variables.

4. Experiments and Results

Our experiments profile and automatically annotate a number of small, but common kernel applications. `binary` implements a binary search over an array of keys, `dotproduct` computes the dot-product of two vectors `u` and `v` of integers; `query` tests a query for a match in a database; and `romberg` does numeric integration of a function `((x*x)+(2.0*x)+1.0)` by iteration. In addition to the four small programs, we also profiled `perl`, an interpreter for the scripting language of the same name, taken from the SPEC95 [17] benchmark suite.

For the kernels we produced profile information for two inputs of different sizes, resulting in 2 profile logs. For `binary` this was an array of keys of size 16 and 4095; for `dotproduct`, vectors of size 10 and 100 that were 90% zero-filled; for `query`, a query with 7 and 21 comparisons; for `romberg`, an iteration bound of 2 and 8. For `perl` we used just one input, a program that tests a list of numbers for their primality. The same inputs were used for profiling and the final performance runs that determine application speedup. 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.

4.1 Instrumentation and Profiling Results

To assess the viability of our profiling tool, we measured (1) the impact of instrumentation on the size and execution time of the applications, (2) the time cost of instrumenting, and (3) the size of the resulting log files. The results, summarized in Table 1, show that instrumenting programs increased their size by a factor of 8-14. In addition, the run time of the instrumented executables was 2 to 4 orders of magnitude slower than that of the original code. (In contrast, Calder et al. [4] report average slowdowns in the range of 10X to 33X for their profiling schemes. We believe that Calpa's order of magnitude slower performance may be attributed to its large amount of file I/O, since it is tracking much more information, e.g., complete value sequences instead of just a small number of invariance metric values, and writing it out to a logfile.)

Program	Code Size (lines)	Instrumented Code Size (lines)	Code Expansion	Original Run Time (second)[a]	Instrumented Run Time (small / large input) (seconds)	Instrumentation Time	Profile Log File Size (small / large input) (KB)
binary	147	1200	8.2	< 0.1	0.1 / 1.3	2.2 seconds	16 / 683
dotproduct	134	1433	10.7	< 0.1	0.1 / 0.5	2.5 seconds	34 / 314
query	149	1795	12.0	< 0.1	3.3 / 5.2	3.3 seconds	1966 / 3071
romberg	158	1273	8.1	< 0.1	0.1 / 1.7	2.7 seconds	39 / 903
perl	23,679	334,958	14.1	< 0.1	101.3	19.6 minutes	15,524
perl (5 functions)	23,679	217,626	9.2	< 0.1	62.7	11.7 minutes	7,456

Table 1: Profiling results

a. `time's` finest granularity is .1 of a second. All execution times were less than .1.

Instrumenting the C source code itself was relatively slow, averaging about 40 lines of code per second[1]. The main reason turned out to be the slowness of a particular SUIF library routine which we use to copy IR instructions. Still, instrumentation time for the kernels was a matter of seconds, and for the moderate size application, `perl`, still quite doable. For the small programs, monitoring and storing data for all variables produced profile log files that ranged from 17 KB to just over 3MB. `perl` required the compression option to keep the log files from growing into the gigabyte range. By selectively instrumenting the 5 top functions that accounted for 80% of the execution time in `perl`, we were able to reduce its instrumentation time by a third, the run time of the instrumented code by over 60%, and, most importantly, the profile data size by a factor of 2. Restricting profiling to the most frequently executed functions is a good way to reduce profiling costs,

[1] With the exception of `perl`, where instrumentation speed was 20 lines per second, for a yet unknown reason.

since only these functions are typically chosen to dynamically compile (the frequent execution pays back the dynamic compilation overhead). We will also investigate summarizing log file data to further reduce log file size.

4.2 Annotation Generation Results

Based on the profile data, Calpa generated annotations for the small programs (`Perl` requires the as yet unimplemented alias analysis). We measured the execution time of the analysis tool and compared the set of Calpa's annotated variables to those chosen by a programmer. The results are shown in Table 2.

Program	Annotated Static Variables	Variety Set Size (2^n maximum elements)	Annotation Time (seconds)
`binary`	size & contents of the input array induction variable for the search loop **the search key**	32 (32896)	3 / 25
`dotproduct`	the contents of vector `u` **the contents of vector `v`**	48 (4224)	5 / 27
`query`	a query **comparison selector**	52 (1088)	61 / 75
`romberg`	the iteration bound	534 (1106)	35 / 128

Table 2: Application & Annotation Characteristics. This table shows, for each program, the variables that were automatically annotated, the size of the variety set generated (compared to the maximum possible), and the analysis tool annotation time, for the small and large inputs. The variables in bold are those that Calpa annotated, but the human missed.

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 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; specializing for these values improved `binary`'s speedup from 2.3 to 3.1 for the large input size.
- 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. Annotating both vectors instead of one, improved the speedup from 2.5 to 5.8 for the smaller vector size; for the large size, speedup was improved from 6.6 to 22.6.
- For both `query` and `romberg` the set of variables manually annotated in our previous study was chosen. Calpa also annotated another local variable that is used as a selector between different comparison operations in `query`, and the function being integrated (which is evaluated repeatedly for only a few different arguments) in `romberg`. However, because of bugs in DyC, we were unable to test the effect of the additional annotations.

Calpa's use set analysis generated variety sets that were generally a small fraction (.1% to 4.8%) of the theoretical maximum of 2^n, where `n` is the number of static variables. The only exception was romberg, whose use sets were independent enough that their combination created many distinct variety sets.

The column named "annotation time" shows the time to generate annotations from both the small and large profile data logs. In general, it took only a few seconds to a few minutes to generate the annotations.

5. Future Work

Calpa's current implementation is a first step towards automatic dynamic compilation. It currently lacks features that will be necessary to deal with most larger and even some small C programs. For example, the profiler identifies assignments to variables by their name only. When assignments are made through aliased variables, these are not correctly attributed. We plan to incorporate an alias and heap analysis to produce more accurate profiling information for named variables and dynamically allocated data structures, respectively.

For larger programs the size of the variety set will be too large to be completely checked. We are working on a smarter search algorithm that uses genetic programming techniques to maintain a population of best choices (instead of just one currently) and combines them to produce other good choices more quickly than the current search algorithm.

Calpa's profiler would benefit from knowing the total time spent in a particular basic block, to better restrict the set of profiled functions to those that comprise the majority of execution time. (Although not currently a bottleneck, variety set generation in Calpa's analysis tool could be frequency-driven as well.) In addition to profiling more selectively, we are also investigating ways to generate more compact, preprocessed profile logs by computing, in the profiler, summary data that are now computed by the analysis, and by not monitoring variables whose values change more frequently than a customizable maximum threshold.

Calpa may improve its cost/benefit analysis by basing benefits on a basic block's critical path length, incorporating the cost/benefit effect of specific optimizations, and using a more detailed accounting of caching costs; it could also inline procedures to extend dynamic regions interprocedurally, replacing the manual @ on function calls.

DyC only handles (dynamically generated) code cache checks that double-hash static variable values. A cheaper implementation would simply consult an invalidation bit that, if set, would trigger respecialization. Should DyC implement invalidation-based caching, Calpa would need to identify invalidation points through analysis; the profile information about the frequency and targets (memory addresses) of such invalidation points has already been implemented.

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 an automated optimization approach (such as ours), 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 [20]. However, to exploit the invariance information, a mapping of the instruction-level information back to the source code is necessary. After identifying the invariant variables, they manually specialize the code for these values. While this occasionally allows them to do more complex code transformations than DyC, it is more time-consuming and error-prone than DyC's automatic generation of annotations and subsequent program optimization; their data, however, could also be used to drive an automatic dynamic compilation system.

Value prediction [15] 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.

While there is a large body of research on dynamic compilation, only a small fraction relates to automation. Autrey and Wolfe [2] 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. A loop nesting analysis could be added to Calpa; however, it appears that coupling profiled data with static analysis identifies more opportunities for dynamic compilation than static analysis alone. TypeGuard and MemGuard are two tools used in the Synthetix project [21] 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.

One important component of Calpa is the cost/benefit model that is used to estimate the impact of annotations on the performance of code generated by DyC. IPERF [12] 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 [22] 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).

6. Conclusions

In declarative dynamic compilation systems, such as DyC, finding the right annotations is a major stumbling block to profitable runtime 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 programs, sometimes achieving better program speedups than the manual process. Our next step is to improve the analysis and profiling of our system and tackle larger, real world applications, with the long-term goal of making dynamic compilation just another in a series of optimizations performed by compilers.

Acknowledgments

We'd like to thank Brian Grant and Matthai Philipose for valuable comments on an earlier draft, and Mike Smith and Glenn Holloway for Machine SUIF source and technical help in using it.

References

[1] L. O. Anderson. Partial Evaluation of C and Automatic Compiler Generation. In *Proceedings of the 4th International Conference on Compiler Construction CC'92*. Springer-Verlag, Berlin, Germany, LCNS vol. 641, pages 251-257, October 1992.

[2] T. Autrey and M. Wolfe. Initial Results for Glacial Variable Analysis. In *Proceedings of 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 120-134, August 1996.

[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. In *Journal of Instruction-Level Parallelism*, vol. 1, (1999), pages 1-37, March 1999.

[5] C. Consel and F. Noël. A General Approach for Runtime 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.

[6] R. Ghiya and L. J. Hendren. Connection Analysis: A Practical Interprocedural Heap Analysis for C. In *Proceedings of 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 515-533, August 1996.

[7] 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.

[8] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. An Evaluation of Staged Runtime Optimizations in DyC In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 293–304, May1999.

[9] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Annotation-directed Runtime Specialization in C. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 163–178, June 1997.

[10] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. A Cost/benefit Analysis of Staged Runtime Optimizations in DyC. To be submitted to *TOPLAS*.

[11] G. Holloway and C. Young: The Flow and Analysis Libraries of Machine SUIF. In *Proceedings of the 2nd SUIF Compiler Workshop*, August 1997.

[12] C-H. Hsu and U. Kremer. IPERF: A Framework for Automatic Construction of Performance Prediction Models. In *Proceedings of the 1st Workshop on Feedback-Directed Optimization*, October 1998.

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

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

[15] M. Lipasti and J. P. Shen. Exceeding the Dataflow Limit via Value Prediction. In P*roceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226-237, December 1996.

[16] F. Noël, L. Hornof, C. Consel, and J. L. Lawall. Automatic, Template-based Runtime Specialization: Implementation and Experimental Study. In *International Conference on Computer Languages*, pages 132-42, May 1998.

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

[18] B. Steensgaard. Points-to Analysis in almost Linear Time In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 21-41, January 1996.

[19] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Andersen, S. Tjiang, S-W. Liao, C-W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. In *SIGPLAN-Notices,* vol.29, no.12, pages 31-37, December 1994.

[20] A. Srivastava and A. Eustace. ATOM, a System for Building Customized Program Analysis Tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196-205, June 1994.

[21] SYNTHETIX TOOLKIT, http://www.cse.ogi.edu/DISC/projects/synthetix/toolkit/.

[22] K-Y. Wang. Precise Compile-Time Prediction for Superscalar-Based Computers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 73-84, June 1994.