Automatic Dynamic Compilation Support for Event Dispatching in Extensible Systems

Craig Chambers, Susan J. Eggers,
Joel Auslander, Matthai Philipose, Markus Mock, Przemyslaw Pardyak

Department of Computer Science and Engineering
University of Washington
Box 352350, Seattle, WA 98195-2350
(206) 685-2094; fax: (206) 543-2969
{chambers,eggers,ausland,matthai,mock,pardy}@cs.washington.edu

Abstract

This paper describes extensions to an automatic dynamic compilation framework to support optimized event dispatching in the SPIN extensible operating system.

1 Introduction

In several extensible operating systems, application-level software that replaces or augments system services is downloaded into the kernel at run-time [Yok92,CT95,RL95,vDHT95,BSP$^+$95]. One of them, SPIN, relies on an implicit event invocation mechanism to connect the disparate and dynamically-loaded parts of the kernel. Modules (either pre-defined or dynamically-loaded) register interest in particular events, and the system implicitly notifies them whenever an interesting event is raised. A straightforward implementation of event dispatching would maintain a global list of guard/handler procedure pairs (or one global list per event). The guard procedure encodes the circumstances (expressed as a boolean predicate over the arguments to the event and other global system state) under which the module is interested in the event, and the handler procedure carries out the module's desired actions whenever those circumstances arise. When a module registers interest in an event, it adds a pair of procedures to the global list. When an event is raised, the system invokes all guard predicate procedures on the list, and for those guards that return true, the corresponding handler procedures are invoked.

This implicit event invocation mechanism provides a flexible base for having different modules interact in a decoupled fashion. However, the straightforward implementation of event processing is too slow for frequently-raised events with many handlers. Improving upon this simple implementation is difficult, since guard/handler pairs are added and removed dynamically as the system runs. One promising strategy applies run-time compilation to dynamically produce specialized machine code for event dispatching, for a particular list of guard/handler pairs; when the list changes, the specialized machine code is regenerated for the new list. An initial implementation of dynamic compilation for event dispatching [PSB] unrolls the loop over the guard/handler pairs into a linear sequence of machine code. The code for each iteration of the unrolled loop is produced by copying hand-generated machine code templates, patching in the guard and handler procedure addresses, and performing simple inline-expansion of guard procedures that are short. This implementation of event dispatching performs two to five times faster than the straightforward data-structure-based implementation.

We have developed a framework for automating much of the dynamic compilation process [APC$^+$]. Given some simple programmer annotations, our compiler automatically performs constant propagation, constant folding, branch folding, and loop unrolling based on variables and data structures identified as invariant at run-time. To minimize dynamic compile time, a static compiler plans out much of the optimization effort ahead of time; at run-time, a simple dynamic compiler copies pre-compiled machine code templates and fills in the appropriate run-time constant values.

In this paper we show how our dynamic compilation framework, when extended in several ways, can produce optimized event dispatching code largely automatically. The generated dispatchers include a number of optimizations of guard and handler code currently not supported by the hand-written version, such as the elimination of redundant guard predicates and the collapsing of multiple related predicates into a single n-way branch, both after inlining. Several of the extensions to our framework are likely to be useful in effective dynamic compilation of other applications as well.

The next section of this paper describes our current dynamic compilation framework and briefly compares it to some other dynamic compilation systems. Section 3 shows how our techniques apply to the event dispatching application, and describes several extensions to our framework that are either necessary or desirable to achieve high-quality generated code. The last section briefly summarizes.

2 The Dynamic Compilation Framework

The framework for automating dispatcher optimizations [APC$^+$] is a compiler infrastructure for dynamic compilation that optimizes portions of programs at run-time (called *dynamic regions*) with respect to values of variables that are unknown until run-time, but remain constant once computed (called *run-time constants*). In our current implementation programmers add simple annotations to their programs which trigger dynamic compilation: the annotations identify dynamic regions, the variables that are run-time constants at the start of the regions, and which loops over run-time constant data structures should be completely unrolled.

The framework itself is composed of two compilers: an optimizing *static compiler* and a fast *dynamic compiler.* The static compiler produces pre-compiled *machine-code templates* (see Figure 1), whose instructions contain *holes* that are filled in with run-time constant values. The static compiler also generates *set-up code* to calculate the values of run-time constants that can be derived from the original, programmer-annotated set, and *directives* that instruct the dynamic compiler how to produce executable code from the templates and the set-up code’s computed constants. It is the set-up code and directives that provide the mechanism for performing optimizations at run-time. The second component of the framework is a fast dynamic compiler (called the *stitcher*) that simply follows the directives, copying the machine-code templates and filling in the holes with the appropriate constants.

The rest of this section discusses the static and dynamic compilers in more detail.

2.1 The Static Compiler

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

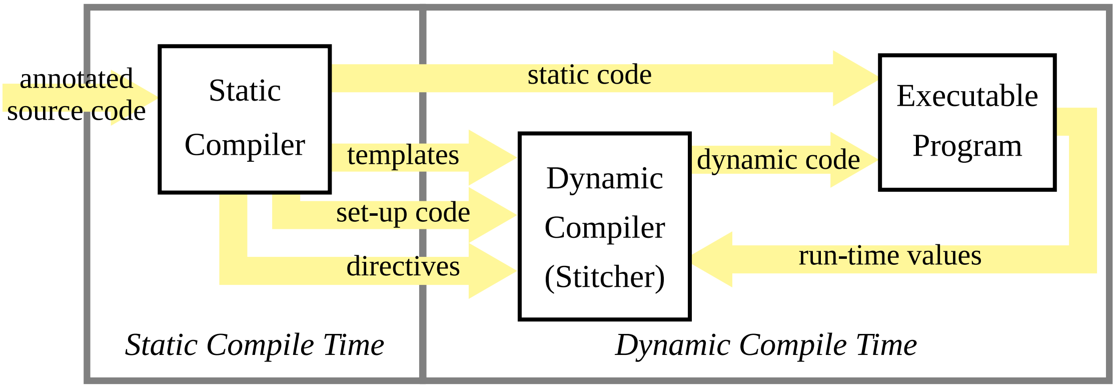

Figure 1: Overall Static/Dynamic Compiler Structure

Run-time Constant Identification. The first step computes the set of variables and expressions that are constant[*] at each point in the dynamic region. This analysis is similar to binding time analysis in off-line partial evaluators [SZ88,JGS93] (except that our analysis is at the level of control flow graphs rather than abstract syntax trees) and to traditional constant propagation and folding (except that our analysis must cope with knowing only that a variable will be constant, not what the constant value is). We have developed a pair of interconnected analyses, one that computes the set of run-time constant variables at each program point within the region, and another that refines that solution by computing reachability information downstream of branches whose predicates are run-time constants. When executed in parallel, they provide an analysis that identifies run-time constants over a potentially unstructured control flow graph program representation.

The run-time constant analysis is a forward (iterative) dataflow analysis that computes the set of variables that are run-time constant for each program point. At the start of a dynamic region, the set of constants is the set of variables specified by the programmer; each instruction in the region can either add to or remove constants from the set. For example, if the operands of an instruction are constants, the result is a constant as well, and so it is added to the set. The analysis places no restrictions on the type of data that can be treated as a run-time constant; in particular, the contents of arrays and pointer-based data structures are assumed by default to be run-time constants whenever accessed through run-time constant pointers. (We include a mechanism to override this default for partially-constant data structures.)

To make the run-time constants analysis effective for an arbitrary control flow graph, we perform a reachability analysis in parallel with the run-time constants analysis. The reachability analysis computes the restricted conditions, expressed in terms of outcomes of branches whose predicates are run-time constants, under which a given program point can be reached after dynamic compilation. The successors of a run-time constant branch will have restricted reachability: true for one path, false for the other. At merges, the reachability analysis can determine whether merge predecessors are mutually exclusive; if so, any variable identified as constant on all predecessor paths will be considered constant after the merge, even if different predecessors bound the variable

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

to different run-time constants. This analysis works well for `if` and `switch` branches whose predicates are run-time constant, and can cope with unstructured flow graphs.

Unrolled loops are handled by treating the loop head merge as one in which the incoming predecessors have mutually exclusive reachability. This treatment enables loop induction variables (and values derived from induction variables) to be identified as run-time constants.

Extracting Set-up Code and Templates. The second step of compiling dynamic regions divides each region subgraph into two separate subgraphs, one for set-up code, the other for template code. The set-up subgraph contains the calculations that define run-time constants (as identified by the first phase), plus code to store the constants into a table for communication with the stitcher; this code is run at most once per dynamic region. The template subgraph contains all the calculations that depend at least in part on non-constant variables, with "holes" for constant operands; the instantiated template code is executed each time the region is entered. Once constructed, the two subgraphs replace the original dynamic region.

Optimization. The third step optimizes the control flow graph for each procedure, applying all standard optimizations without regard to template boundaries. For the most part, the analyses can treat template holes as compile-time constants of unknown value. In some situations the optimizations were updated to work correctly with holes; for example, since holes shouldn't be treated as legal values outside a dynamic region, they are not copy propagated across region boundaries.

Code Generation. The final step generates machine code and stitcher directives. The directives instruct the stitcher how to patch the template holes, adjust pc-relative offsets, eliminate untaken branches and completely unroll loops.

2.2 The Stitcher

Given the preparation by the static compiler, the stitcher has only to follow the directives to instantiate the machine-code templates. Most of the tasks are straightforward, such as copying blocks of template code and filling in hole values. It also performs simple peephole optimizations on the template code that exploit the actual values of the run-time constant operands.

2.3 Putting Them Together

Our dynamic compilation framework was designed to produce high-quality dynamically-compiled code, at low dynamic compilation cost. To generate high-quality code, the static compiler applies standard global optimizations to the machine-code templates, optimizing them in the context of their enclosing procedure. It also plans out the effect of run-time constant-based optimizations, so that the final, optimized templates contain only the calculations that remain after these optimizations have been performed. Lastly, the entire analysis is embedded within an optimizing compiler (in our case, the Multiflow compiler [LFK+93]). The low dynamic compilation overhead is achieved by presenting the stitcher with almost completely constructed machine code. Initial results on two C utility programs exhibit speed-ups of 1.6 and 1.9 over statically-compiled code with negligible dynamic compilation time overhead [APC+].

Our dynamic compilation framework is novel in several respects. First, it is capable of handling the full functionality of C, without restricting its normal programming style.* Second, it can derive run-time constants automatically via its two interconnected dataflow algorithms; when executed in parallel, they provide an analysis that handles unstructured control flow well. Finally, since the analyses is integrated into an optimizing compiler, dynamically-compiled code is heavily optimized with its surrounding code, without limitations on the kinds of optimizations that can be applied.

Consel and Noël [CN96] describe a similar framework for dynamic compilation which follows a more traditional partial evaluation approach. Their binding time analysis is done over abstract syntax trees, and as such is conservative in the face of unstructured control flow. They are able to identify run-time constants interprocedurally, while our analysis is currently intraprocedural. Our implementation is integrated with an existing optimizing compiler, enabling us to adapt the compiler's optimizations to work properly in the face of template code with holes; their implementation relies on recognizing certain assembly code idioms for holes and assumes the optimizing compiler will not disturb these idioms. Annotations in the two frameworks are used differently as well.

Leone and Lee [LL96] describe a simpler framework for compiling a first-order purely functional language. They do not perform aggressive static optimizations or planning, leading to poorer-quality generated code and slower dynamic compilation (since some optimizations are performed at run-time rather than statically). Engler *et al.* [EHK96] describe a more manual but flexible approach, which provides the programmer with first-class code fragments that can be assembled and dynamically-compiled under programmer control. This approach allows programmers to perform their own high-level optimizations, such as run-time constant folding and loop unrolling, but, conversely, requires the programmers to perform their own optimizations. Low-level optimizations across code fragments are not supported.

Still two other systems optimize operating systems code less automatically: Pardyak et al. [PSB] optimized the implicit event invocation mechanism in SPIN; Pu et al. [PAAB+95] proposed a specialization model for Synthetix and applied it to file reads.

3 Dynamic Compilation of Event Dispatching

Dynamic compilation greatly speeds up event dispatching in SPIN [PSB] and similar systems. Conceptually, a naive, statically-compiled implementation of dispatching interprets a relatively invariant data structure, namely the linked list of guard/handler pairs (see example code in Figure 2†). This interpreter would iterate through the linked list, evaluating the guard procedure at each element, and invoking the corresponding handler procedure if the guard returns true.

Since the set of handlers for events is fixed for relatively long periods of time, we would treat the interpreter loop over this data structure as a dynamic region, and dynamically-compile it with

* A Modula-3 front-end will be added for dynamically compiling SPIN extensions.

† We are simplifying the semantics and interfaces to event dispatching, guards, and handlers to focus on the aspects most interesting for dynamic compilation. Annotations for the dynamic region, the run-time constants (`event` and `handlers`), the key to the cache of specialized versions of the dynamic region (`event`), and the loop to unroll fully are indicated in bold.

```
struct HandlerList {
  int (*guard_proc)(EventName event, ArgList* args);
  void (*handler_proc)(EventName event, ArgList* args);
  struct HandlerList *next;
} *handlers;
void process_event(EventName event, ArgList *args) {
  dynamicRegion foreach(event) constant(handlers) {
    unrolled for(; handlers != NULL; handlers = handlers->next) {
      if ((*handlers->guard_proc)(event, args)) {
        (*handlers->handler_proc)(event, args);
      }
    }
  }
```

Figure 2: Annotated Event Dispatcher

respect to the current set of installed handlers. Our dynamic compiler framework can automatically produce a specialized version of the body of the dynamic region for each different `event` name being dispatched, with the body of the dispatching loop unrolled fully for all installed handlers. Within each iteration, the `guard_proc` and `handler_proc` code addresses are known statically, and therefore their procedures can be called directly.

Although this applies some of the benefits of dynamic compilation to dynamic dispatching, more opportunities for optimization exist. Below we identify the extensions we need to make to our framework to generate better code for event dispatching.

Support for infrequently changing quasi-constants. Our framework needs to support constants that occasionally change, such as the `handlers` global variable above. This can easily be done with an additional annotation that denotes quasi-constants [PAAB^{+}95] placed on the code that changes them. The annotation would cause the compiled code that had been cached for this dynamic region to be flushed. A simple implementation would enable the flushing of all dynamic regions whenever any quasi-constants changed; more sophisticated systems would allow regions to be named and flushed selectively, or would automatically track which regions depended on which quasi-constants. A harder problem is synchronizing cache flushing when there is code concurrently executing within the dynamic region [PAAB^{+}95]; in the SPIN dispatcher, updates to the `handlers` list affect only future event dispatches; the old dispatcher code is garbage-collected once all executing dispatches have completed.

Inlining of run-time constant procedures. SPIN dispatching makes heavy use of run-time constant code pointers (as illustrated by `guard_proc` and `handler_proc` in Figure 2). The guard procedures are often very simple, and many guards contain partial redundancies in the form of predicate subexpressions that perform the same test or similar tests (such as multiple guards testing the system-call-number argument against different constant values). If simple guard procedures were inlined, the run-time overhead of the procedure call and return would be eliminated, and a number of important post-inlining optimizations could be performed: checks against the constant true could be folded away, repeated predicate subexpressions across (inlined) guards could be identified and eliminated, and multiple tests of a value against different constants could be collapsed into a single, multi-way branch. When these optimizations are completed, the simple linear list of guard/handler call pairs is transformed into a decision-tree-like structure, implemented with executable code rather than passive data structures.

Simple inlining of run-time constant procedures can be easily implemented in our current framework: fragments of machine code would simply be spliced into the dynamic region in place of the call. However, much of the benefit of inlining guards comes from the simplifications that are made after inlining. This is much more difficult to support with our current framework, since we have intentionally stripped away all of the sophistication of the dynamic compiler to make it fast. To support post-inlining optimizations, we will need to augment our machine-code template representation of programs that is available at dynamic compile time to include enough information to perform selected optimizations after inlining. We consider this extension and the studies of the trade-offs among different run-time program representations, their dynamic compilation times, and the quality of their dynamically-generated code to be a relatively long-term research effort.

In the shorter term, we are side-stepping this run-time inlining issue in our experimental prototype by recasting the guard procedures as explicit data structures akin to Lisp S-expressions. Instead of writing Modula-3 code for a guard, the programmer will construct a data structure that represents the test that the guard performs.[*] A separate S-expression optimizer will transform the linear list representation of guard/handler pairs into the faster decision-tree factored representation (actually, to avoid unnecessary duplication of code, the decision “tree” is a DAG). Finally, the event dispatching interpreter will become an interpreter over the decision tree rather than the initial handler list. We will use this alternative implementation to gain an understanding of the potential benefits due to dynamic compilation, different forms of redundancy elimination and factoring in the guard decision tree, and simple inlining of the guards and (short) handlers.

Multi-way loop unrolling. Unrolled loops are usually thought of as linear sequences of loop bodies. But an interpreter over a decision tree has multiple successor paths down which the loop can proceed. Each of these paths needs its own unrolled loop, to produce a tree of loop bodies isomorphic to the decision tree’s structure. Moreover, since the decision tree is, in fact, a DAG, with some subgraphs shared, loop unrolling needs to detect when unrolled loop iterations should be shared.

Both of these extensions can be accommodated by annotating the unrolled loop with a loop induction variable (such as the current S-expression being evaluated) that will be maintained as a run-time constant. If the S-expression variable is assigned different constant values along different cases within the loop iteration (such as the two successor S-expressions after an `if` S-expression), the static compiler will ensure (through tail duplication) that different control flow paths are maintained for each distinct value of the annotated variable, and that each loop back-edge with a different value will lead to a distinct unrolled loop iteration. If different loop back edges end up with the same value for the annotated variable (this can happen with shared subtrees in the decision DAG), then the stitcher will ensure that both back-edges lead to the same unrolled iteration; consequently, the compiled code will be shared in the same shape as the sharing in the decision DAG being interpreted.

[*] A more sophisticated implementation would have the Modula-3 compiler automatically produce these S-expressions.

Specializing event dispatchers for different event invocation sites. Our current framework supports specialized versions of a dynamic region for each distinct value of some parameter (e.g., the `event` variable in Figure 2). This will allow each version of the dispatching code to include only the guard evaluations that pertain to a particular event; the other guards will have been eliminated, due to constant-folding of comparisons of the `event` variable against other constant event names.

It is also possible to exploit more invariant information at particular sites where events are being raised. For example, if an event invocation site always passes a particular constant to the event (such as the number of the system call being invoked), then an even more specialized event dispatcher is possible, where all comparisons against the constant argument are folded away. To support this, we would need some mechanism to indicate which arguments to the event invocation are run-time constants. We anticipate handling this by extending the notion of dynamic regions to operate more interprocedurally; a call site would be annotated as a dynamic region and would include the callee procedure.

4 Summary

This paper described extensions to a dynamic compilation framework that will support optimized dispatching in the SPIN extensible operating system. The dispatch-related optimizations include support for infrequently changing quasi-constants, inlining of run-time constant procedures, multi-way loop unrolling, and specializing event dispatchers for different event invocation sites.

Acknowledgments

We would like to thank John O'Donnell (Equator Technologies, Inc.) and Tryggve Fossum (Digital Equipment Corp.) for the source for the Alpha AXP version of the Multiflow compiler. The research is supported by ARPA contract N00014-94-1-1136, NSF PYI Award MIP-9058439, NSF NYI Award CCR94-57767, and ONR grant N00014-92-J-1395.

References

[APC+96] J. Auslander, M. Philipose, C. Chambers, S.J. Eggers, and B.N. Bershad. Fast, effective dynamic compilation. *Conference on Programming Language Design and Implementation*, May 1996.

[BSP+95] B.N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Symposium on Operating Systems Principles*, November 1995.

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

[CT95] R.H. Campbell and S.-M. Tan. Microchoices: An object-oriented multimedia operating system. In *5th Workshop on Hot Topics in Operating Systems*, May 1995.

[EHK96] D.R. Engler, W.C. Hsieh, and M.F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Symposium on Principles of Programming Languages*, January 1996.

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

[LFK+93] P.G. Lowney, S.M. Freudenberger, T.J. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. O'Donnell, and J.C. Ruttenberg. The Multiflow trace scheduling compiler. *Journal of Supercomputing*, 7, 1993.

[LL96] M. Leone and P. Lee. Optimizing ML with run-time code generation. In *Conference on Programming Language Design and Implementation*, May 1996.

[PAAB+95] C. Pu, T. Autrey, C. Consel A. Black, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Symposium on Operating Systems Principles*, November 1995.

[PSB] P. Pardyak, S. Savage, and B.N. Bershad. Language and runtime support for the dynamic interposition of system code. See http:/www.cs.washington.edu/research/projects/spin/www/papers/WCS/dispatcher.ps.

[RL95] J. Itoh R. Lea, Y. Yokote. Adaptive operating system design using reflection. In *5th Workshop on Hot Topics in Operating Systems*, May 1995.

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

[vDHT95] L. van Dorn, P. Homburg, and A.S. Tanenbaum. Paramecium: An extensible object-based kernel. In *5th Workshop on Hot Topics in Operating Systems*, May 1995.

[Yok92] Y. Yokote. The Apertos reflective operating system: The concept and its implementation. In *Object-Oriented Programming Systems, Languages and Application*, October 1992.