Design and Implementation of a Dynamic Compiler

Matthai Philipose
University of Washington
October 10, 1996

1.0 Introduction

Compiler optimizations have traditionally focused on optimizing code at compile time [ASU86] , using values available statically (compile-time constants), or predictions of values deduced from profiling runs. Recently [KEH93] [LL96] [APC+96] , however, interest has grown in *dynamic compilation*, in which code is optimized around values available only at runtime. Dynamic value-based optimizations include load elimination, constant propagation and folding, branch and dead code elimination, complete loop unrolling, strength reduction and register allocation of frequently used memory locations.

Since optimizations based on values available only at runtime require the runtime optimization of code, the challenge is to minimize the runtime overhead of optimization, while still performing as many of the above optimizations as possible. Further, with modern architectures, it is important that the dynamically produced code is register-allocated well and that it is scheduled to make the best possible use of functional units (particularly in superscalar or VLIW processors).

Our solution to this trade-off between code quality and runtime overhead is to plan out the dynamic optimizations statically (based on variables whose values are expected to be fixed during runtime), so that runtime actions principally involve only copying and filling in statically generated and optimized templates. Since the templates are statically register allocated and scheduled in the context of their surrounding code, the runtime optimizer (which we call the *stitcher*) needs to do very little additional work scheduling and register allocating.

This paper will focus primarily on the design and implementation of parts of the dynamic compilation system that optimize and annotate the code templates at static compile time, and instantiate the templates at runtime. We will not examine the process by which the templates are identified. In section 2, we first present an overview of the entire dynamic compilation framework and identify the three aspects of the system that will be discussed in the rest of the paper. In section 3, we examine these three areas closely: the effect of classical optimization phases on the correctness and efficiency of the code templates, the mechanism (called *stitcher directives*) by which the static compiler communicates with the stitcher, and finally, issues related to the design, implementation and performance of the stitcher. In section 4, we present the initial results of applying our framework, both in terms of improvement in execution time of the applications themselves, and time spent in various parts of the stitcher. We identify deficiencies in the design of the stitcher and propose improvements. Section 5 compares our approach, specifically the runtime components of our framework against those of related projects. We suggest directions for future work in section 6 and present our conclusions in section 7.

2.0 Overview of our system

Figure 1 illustrates the high-level structure of the dynamic compilation framework.The system consists of a conventional optimizing compiler (labelled *static compiler* in figure 1) which has been augmented with analysis that prepares specially demarcated parts of the executable for runtime optimization, and a *dynamic compiler*, which performs optimizations on the code prepared by the static compiler.

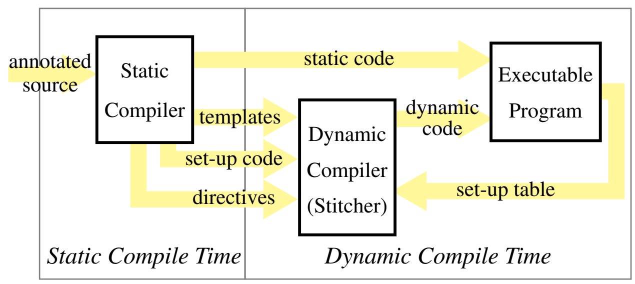

Figure 1: High-level structure of the dynamic compilation system

The input to the static compiler is source code (we currently support fully general C) that has been annotated by the programmer to indicate the extent of the region on which dynamic compilation should be performed (which we call the *dynamic region*), the variables (*root runtime constants*) based on which value-specific optimizations should be performed at runtime and the loops that should be fully unrolled, also at runtime. Finally, some applications contain dynamic regions that use root runtime constants whose values vary within a small set of values. We allow the user to annotate such runtime constants as *keys,* and automatically generate (and cache) a specialized version of the dynamic region for each set of key-value bindings.

The static compiler performs an iterative dataflow analysis (called a *binding time analysis or BTA,* after a similar analysis in the partial-evaluation literature [JGS93]) on the annotated code in order to:

- Identify all runtime constants that derive from the root runtime constants (by an analysis similar to constant propagation and folding), and
- Separate the computations that depend only on static or runtime constant values (called *set-up code*) from the remaining computations (which constitute *template code*) in the dynamic regions.

Intuitively, the set-up code is the part of the dynamic region that performs runtime constant folding, and therefore has to be evaluated just once at runtime. The template code uses the folded constants in computations that also involve dynamic variables, and therefore has to be re-evaluated with each execution of the dynamic region.

The template code contains special markers to identify each use of a runtime constant, and the beginning, end and continuation of loops that need to be fully unrolled. The set-up and template code regions are subjected to classical optimizations (such as common subexpression elimination, copy propagation and constant folding), followed by register allocation and scheduling. The templates used by the runtime compiler are therefore highly optimized, a major advantage of our approach.

The final action of the static compiler is to emit *directives* to the dynamic compiler that guide the dynamic compiler in turning templates into executable code, using the values computed by the set-up code. Examples of instructions that require runtime action (and the corresponding action required) are arithmetic instructions that use runtime constants (the value of the corresponding runtime constant needs to be patched in at runtime), branches whose conditions are determined by runtime constants (such branches and their not-taken paths can be eliminated) and instructions at the boundaries of loops to be unrolled by the stitcher (the bounding instructions demarcate the loop body that is stamped out multiple times for full unrolling).

At runtime, the set-up code performs all one-time calculations and stores them in a data structure (called the *set-up table*) visible to the stitcher. The stitcher uses the directives to identify template code instructions that require special processing, and possibly uses values in the set-up table (the exact location within the setup-table is specified by the directives) to fulfill the directive. All remaining template code instructions are simply copied to the stitched location by the stitcher.

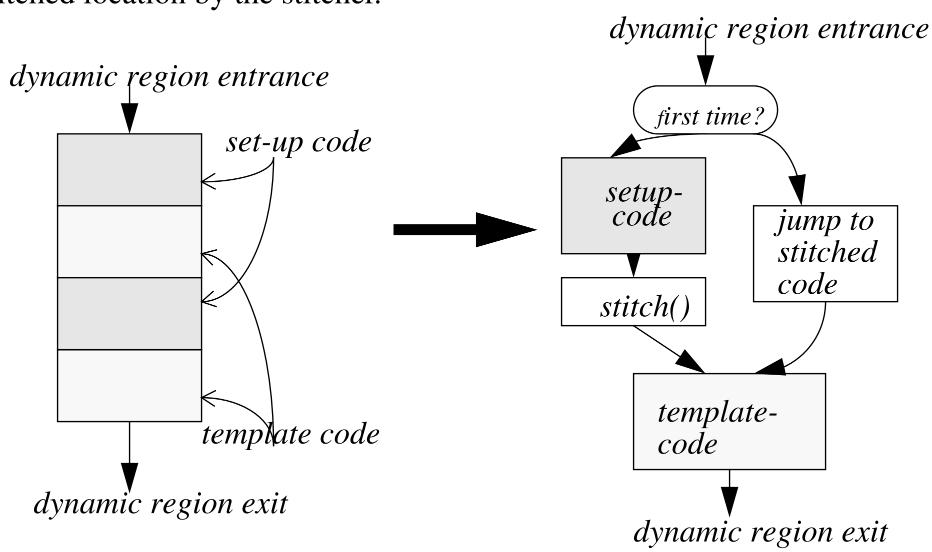

Figure 2: Restructuring the dynamic region to separate set-up code from template code

Figure 2 illustrates how a dynamic region is restructured into set-up and template regions. We insert before the dynamic region a branch which ensures execution of the set-up code followed by an invocation of the stitcher if the code is being executed for the first time, and the execution of a jump into the stitched region otherwise. At the end of the stitched region, we insert a matching jump to the code succeeding the dynamic region.

In the above design, the goal of minimizing dynamic compilation overhead corresponds to minimizing the time spent executing set-up and stitcher code. On the other hand, the goal of producing stitched code of as high a quality as possible is targeted both statically, by isolating and statically optimizing the templates, and dynamically, through actions of the stitcher. Extensive static optimization and fast, effective dynamic optimizations are therefore what we need from our framework. We examine below both the limits on the static optimizations that can be performed in our framework, and the speed and effectiveness of the design and implementation of the dynamic compiler.

3.0 Details on design and implementation

We implemented the static part of the dynamic compilation infrastructure by modifying the Multiflow compiler[LFK+93]. In addition to adding a phase that implemented the BTA, we modified an existing control flow graph (CFG) node type to represent the runtime constants identified by the BTA, used special labels to mark points in the code across which code-motion should be limited (for instance, we wish to prevent uses of runtime constants from being moved outside the dynamic region), and modified subsequent static optimization phases to prevent them from performing such code motion. Further, we modified the code emission phase to produce stitcher directives, if necessary, for each instruction emitted. Finally, we implemented the stitcher as a separate module which could be invoked from the modified dynamic region, as shown in figure 2. We discuss in the following subsections limitations on the optimization of template code, the design of and issues related to producing stitcher directives, and design, implementation and performance issues related to the stitcher.

3.1 Modifications to static optimization phases

Following the BTA, we run most classical optimizations[ASU86], such as global common subexpression elimination, copy/constant propagation, induction variable simplification, global register allocation, loop invariant motion and global trace scheduling, on the restructured CFG. However, the semantics of template code imply that many classical optimizations need to be modified before being applied to template code. We identify below exactly how template code differs from “normal” code, and then go on to discuss how specific optimizations need to be modified to handle this difference.

The intermediate representation (IR) for the template code produced by the BTA differs from that for normal code in two respects:

1. It contains **marker-labels** that denote the beginning and end of a dynamic region, and the beginning, end and continuation points of loops that are to be fully unrolled. Each beginning/continuation marker has embedded in it a pointer to a sub-table of the setup table[1]. The idea is that for code following the marker, the stitcher will read runtime constant values from this sub-table. The sub-table therefore defines a "context" which associates values with the runtime constants in code following the marker. In particular, multiple sub-tables can be associated with the body of a loop being unrolled, each sub-able corresponding to the runtime constants for a distinct iteration of the loop.
2. Runtime constants in the template are represented by **special *rtconst* nodes**, which have embedded in them the location *relative to the current sub-table* at which the value of the runtime constant may be found.

Given that the position of a rtconst relative to a marker determines the value the stitcher will associate with it, optimization phases that follow the BTA should not move operations using runtime constants outside the context in which they are defined i.e. beyond the markers that enclose them, or remove these markers. The specific optimization phases that require modification in order to prevent illegal motion of runtime constants are as follows.

1. **Copy propagation** must avoid moving runtime constants across the special markers. In particular, when runtime-constants are live across the end of a dynamic region, we need to explicitly copy the rtconsts into temporaries that may be used in the subsequent code. Copy-propagation would ordinarily propagate these rtconst nodes out of the dynamic region. Our current solution is to disable *all* copy propagation on runtime constants.
2. **Loop invariant code motion** must avoid hoisting operations that contain runtime constants derived from the induction variables of completely unrolled loops across the loop boundary markers. These runtime constants have different values associated with them for different iterations of the unrolled loop, and are therefore not actually loop invariant. Again, we currently prevent hoisting of these induction variables by preventing *all* loop-invariant motion of operations that use runtime constants.
3. The **scheduler** needs to avoid speculatively hoisting operations that contain rtconsts above marker labels. We currently prevent all speculative hoisting across markers.
4. **Static loop unrolling**, as conventionally implemented, stamps out a fixed number (typically four in the Multiflow compiler) of copies of the loop body, removing the "continue" branch between these iterations. In this case, we need to ensure that the optimization does not remove the continue *markers* between each copy of the loop-body stamped out. A cleaner option would be to order the static unrolling phase *before* the BTA, so that the final structure of the loop to be unrolled is exposed to the BTA. The nature or our markers currently prevents *any* static unrolling from taking place after the BTA.

1. Recall that the setup table contains the runtime constant values to be stitched into the template code

5. Some optimizations expose runtime constants that are not easily identified by our BTA. For instance, consider the action of **induction variable simplification** (IVS) on the following loop:

```
for ( i=1; i<x; i++)
   a[i]=7;
```

After induction variable simplification, the overhead of calculating the induction variable `i` in the above loop may be eliminated by rewriting the loop as:

```
y = a + 4*x;
for(p=(a+4); p<y; p+=4)
    *p=7;
```

In the case that `a` is a runtime constant, we may want the initialization of `p` and the calculation of the loop-bound `y`, in this loop to be placed in the set-up code.[2] Static loop unrolling is another optimization that rewrites induction variable limits and therefore exposes runtime constant computation. IVS and loop unrolling should therefore precede the BTA.

Another optimization that could expose new runtime constants is **data critical-path reduction (CPR)**[SK95], where long, sequential chains of computations are converted into sets of instructions that can be executed in parallel, using algebraic transformations. This is a common trick in compiling for a wide machine. For instance, the sequence of instructions (the rtc[i] denote runtime constants):

```
t1 = t0 + rtc1;

t2 = t1 + rtc2;

t3 = t2 + rtc3;
```

may be converted into:

```
t1 = rtc1 + rtc2;

t2 = t0 + rtc3;

t3 = t1 + t2;
```

2. In our implementation, the production of statements that add (or subtract) constants from runtime constants *after* the BTA leads to a significant increase in stitcher overhead. The problem is that (for convenience) we use a form of Multiflow's link time constant (ltconst) data type to represent our runtime constant (rtconst) data type. In our example, Multiflow represents `a+4` as a c*ompound ltconst* (an ltconst which stands for the sum of a constant with an ltconst), assuming the addition will be done at link time. Since this particular ltconst will only be processed at runtime (since it is actually an rtconst), the stitcher has to perform a runtime check whether each rtconst is compound, and perform the addition at stitch time. The value of the constant to be added on is stored in the stitcher directive.

Note that the height of the critical path in the second piece of code is 1 (i.e. the maximum length of the chain of data-dependences in the code is 1), whereas the un-transformed code has height 2. The calculation of t1 is a runtime-constant computation and should be moved to setup code. Phases that trigger such algebraic re-association should therefore precede the BTA.

Note that if the number of "new" runtime constants exposed by these optimization phases is small, we may choose to treat the runtime constants as ordinary temporaries whose values will be recomputed each time the dynamic region is executed. On the other hand, in some circumstances, it may be profitable to explicitly re-associate operands in chains of calculations so that computations between run-time constants may be grouped together and moved into setup code.

6. Finally, optimization techniques suitable for statically compiled code may be less suitable for code that will be runtime compiled. For example, an increasingly common technique to increase instruction level parallelism is the **speculative hoisting** of instructions above branches. However, if the branch in question is a *static branch* (i.e., it is controlled by a runtime constant), the results of instructions hoisted from the "not taken" path of the branch will clearly never be used, and speculatively hoisted instructions from the *taken* path are no longer speculative. We currently disable *all* speculative hoisting. This drastic solution did not affect us adversely because our experiments were performed on the DEC Alpha 21064, a 2-way issue machine. On machines that support higher levels of parallelism, we may need to tag speculatively moved instructions with the static branch they depend upon, so that correctly guessed instructions may be left in, and others removed at stitch time.

One attractive option is to place the BTA phase after as many of the above phases as possible, so that it is unnecessary to modify those phases to account for the results of the BTA. Since runtime-constants are codegened as immediate literals in instructions, and since the resource requirements for setup code and template-code need to be considered separately (since they do not execute simultaneously), the BTA would need to occur before register allocation or scheduling. Arguments against performing the BTA late include the fact that annotations for dynamic compilation are currently implemented as function calls in the IR; the presence of these calls could significantly reduce the effect of intra-procedural optimization phases that make conservative assumptions across function calls. Further, modern compiler back-ends (but not, it seems, the version of Multiflow we are working with) invoke optimizations like copy-propagation, dead-code elimination, data CPR and even loop-unrolling before the scheduling phase. Given the restriction that the BTA should precede scheduling, it is impossible to place the BTA phase late enough in these compilers.

3.2 Stitcher directives: the language interpreted by the stitcher

Once a template is laid out, we need to mark parts of the template code which require runtime action with the *type* of action and the *location* of any related information (e.g. the location in the runtime constants table of a particular rtconst, or the location of an instruction to be stitched) needed to perform the action. There are two possible ways in which the executable can be annotated.

1. At the point in the setup-code that computes the value to be inserted in the template, statically inline a *stitching-fragment*, i.e. a fragment of “stitcher code” that performs the required operation on the template. The advantage of this approach is that no time is wasted at runtime interpreting the annotation: the annotation is *executable*, rather than *interpretable*. The disadvantage is that construction of the stitcher code for various types of template instructions is tedious and error-prone, since it involves constructing fragments of machine code or intermediate language that represent the stitcher action, and inserting them at the appropriate point in the setup code.
2. Emit a *directive* that points to the template-code statement and contains the type of action and the location of related data. The directive is created statically in the data segment of the program, and can be interpreted by a separate “stitcher” function at runtime. The primary disadvantage of this approach is the runtime cost of interpreting the directive. The main advantage (and the reason we decided to adopt this approach for the initial implementation) is that the stitcher can be written in a high level language.

Stitcher directives are currently emitted as part of the code emission process. Table 1 lists the circumstances under which each type of directive is emitted.

TABLE 1. Stitcher directives and conditions under which they are generated

Directive (with associated data requirements)	When Generated
START(*inst*)	beginning of template code
END(*inst*)	end of template code
HOLE(*inst*, *operand #*, *const’s table index*)	hole marker operand in instr
CONST_BRANCH(*inst*, *test’s table index*)	hole marker as branch test
ENTER_LOOP(*inst, table header index*)	unrolled loop entry marker
EXIT_LOOP(*inst*)	unrolled loop exit marker
RESTART_LOOP(*inst, next table index*)	unrolled back edge marker
BRANCH(*inst*)	pc-relative instruction
LABEL(*inst*)	target of pc-relative instr.

Each directive occupies 24 bytes and contains the following fields:

1. **opcode**, which determines the action taken by the stitcher
2. **location of the instruction** to be patched, expressed as a pointer to the instruction.
3. **location of the runtime constant** to be patched in (if any), in the set-up table.
4. location of the **directive that corresponds to the branch target** (useful only if the instruction to be patched is a branch).
5. **location of the sub-table** of the set-up table for the next iteration of a loop (only for loop directives).

A problem with emitting directives in the order that code is emitted is that we have no way to determine if a given label or branch is within the dynamic region. As per table 1, we are therefore forced to emit directives even for branches and labels not within the dynamic region. We therefore generate as many extra directives as there are branches and labels in the non-dynamic region of the procedure. For large procedures, this extra demand on the data segment may be excessive. The underlying problem is that we keep track of being "inside" or "outside" the dynamic region only during the BTA which operates on the intermediate representation. We need access to this information at the point when the assembler for the template region is ready for emission.

In most cases, the values that go into the fields of a directive are encoded in the instruction to be patched, so that filling in a directive at the time that instruction is emitted is trivial. For instance, the *location in the set-up table* where the value of a given runtime constant may be found is embedded in the rtconst-operand data type, and the *locations of subtables* of loops are embedded in the markers demarcating the loops. We mark the *location of an instruction* that needs patching with a new label. The only tricky case occurs with directives for branch instructions. In this case, we need to enter the location of the *directive that corresponds to the branch target* of the current instruction. However, because this directive may not yet be emitted (because the branch target has not been emitted), its location may be unknown. We get around this problem by using a consistent naming scheme. For a branch instruction `br foo` where `foo` is the label of the target instruction, we always label the corresponding target directive `foo_target`. Thus, even if the target directive has not yet been emitted, we can deduce the name of its location from the corresponding branch instruction.

3.3 Stitcher

Given the structure of the *template code*, which provides a close approximation of what the dynamically compiled code should look like, the *set-up table*, which contains the values that will be patched into the template code, and the *stitcher directives*, which detail how the set-up-table values should be used in transforming the template, the design of the stitcher is straightforward. The stitcher loops over stitcher directives, "executing" the directives on the template code instructions they point to and copying the remaining instructions from the template. The duties of the stitcher can be summarized as follows:

1. It **lays out code** from the template i.e. it determines the linear order of basic blocks being stitched.
2. It **resolves labels** in the newly laid-out code.
3. It performs **value-based peephole optimizations** on instructions that use runtime constants, based on values recorded in the set-up table.
4. It **eliminates static branches** (i.e. branches controlled by runtime-constants) and corresponding dead code.
5. It completely **unrolls loops** marked for unrolling.

The challenge in implementing the stitcher lies in delivering this functionality with little runtime overhead, *and* producing efficient dynamically compiled code. In the following subsections, we introduce the data structures used by the stitcher, discuss how the stitcher uses these data-structures to provide the above functionality, and identify significant sources runtime overhead inherent in our design.

3.3.1 Data structures used by the stitcher

A conventional two-pass assembler would (in its first pass) scan linearly through the code, recording the location to which each label resolves, leaving the target field of each branch instruction unfilled, and recording the location of the unresolved branches. In its second pass, the assembler would patch all the branch instructions. The primary data structure an assembler of this type would have to manage would be a ***label-resolution data structure***, which is implemented as a "branch-to-label" list for recording the branches whose target is as-yet unresolved and a "label-to-address" hash table which records the location to which each label resolves.

Dynamic loop unrolling complicates label resolution. A template-code label located in the body of a loop to be unrolled may resolve to many locations (corresponding to different iterations of the loop body) in the stitched code. We therefore have to either modify the label-resolution data structure to tag each label with the loop number and iteration of the loop it refers to, or maintain ***one label-resolution structure for each loop iteration*** we unroll. The problem with the first approach is that we essentially have to hash on two extra keys per label (the loop number and the iteration number of the loop) to maintain the label-to-address hash table[3]. The second approach has the potential problem that at label resolution time, we need to access the label-to-address list for the appropriate loop-iteration, leading to extra indexing overhead for resolving each label. We solve the latter problem by *resolving all labels for a given iteration when the code for that iteration has been laid out*[4]. In fact, the immediate resolution also allows us to re-use a resolution data structure over remaining iterations of a loop: the information in the data structure at the end of an iteration can be "thrown away" since every branch recorded in the data structure is processed at the end of the iteration, and the label-to-address information for that iteration will not be relevant to future iterations.

Since the stitcher, unlike an assembler, needs to re-layout code in addition to resolving branch targets, it cannot be structured as a linear pass through the stitcher directives. Specifically, the requirement of omitting not-taken paths of static branches dictates that we cannot lay down every basic block, as a linear traversal would. Further, loop unrolling dictates that the entire body of the loop-to-be-unrolled (which may not lie in a contiguous area of memory and therefore cannot be scanned linearly) be copied as many times as the

---

3. The need to do away with extra hashing becomes more pressing because in our implementation, we *avoid the need to hash* altogether by statically assigning a canonical numbering to the labels.
4. The only complication arises when the loop body contains branches to targets outside the loop. Such targets may not be resolved until the loop with the branch target is processed. Such branches are therefore recorded in the resolution data structure of the target loop.

loop iterates, so that the unrolled version of the loop is one contiguous, straight-line piece of code. The stitcher therefore emits all basic blocks *within a given loop body* accessible from the loop head in depth first order, before laying out code for the next iteration or code beyond the current loop. The depth-first scheme requires that the stitcher manage an additional ***worklist of directives***, where each directive corresponds to a basic block yet to be stitched. When the stitcher comes across a directive for a dynamic branch, it places the directive corresponding to the taken path of the branch on the worklist and continues stitching along the not-taken path. At a static branch, the stitcher continues stitching along the side of the branch known to be taken and ignores the other branch, thereby eliminating the dead code associated with the static branch.

3.3.2 Runtime actions of the stitcher

The actions of the main stitcher loop over the stitcher directives are as follows.

1. Read in the current directive, and *dispatch* on its opcode to the code that implements it.
2. *Copy* any intermediate template instructions between the current instruction and that processed by the previous directive.
3. Perform the necessary *action* (see below) based on the current instruction.
4. *Locate* the next directive to be processed.

Recall that we maintain a worklist for each loop iteration, containing directives that have yet to be processed. If step 4 cannot find another directive in this worklist, we conclude that the entire loop body has been laid out for the current loop iteration, perform *label resolution* for the loop body and proceed either to the next iteration[5] or to code beyond the loop. If we are in the outermost loop and no directives remain to be processed, the stitcher exits.

In step 3 above, the actions performed by the stitcher on an instruction are of two main kinds.

1. Actions that perform book-keeping (i.e. maintain the label-resolution and worklist data structures), and
2. Actions that patch instructions.

Book-keeping actions are performed at directives that involve the control structure of the code:

- At every LABEL directive, we need to check if the label is resolved. If so, we stitch in a branch to the target, if not we record the newly resolved location in the resolution data structure.

5. Note that in this version of the stitcher, we assume that each iteration of a loop to be unrolled (i.e. an execution of the loop-body with a given set of induction-variable-to-value bindings) has a unique iteration succeeding it. We do not handle the case of multi-way unrolling, where loop-induction variables would be modified under a branch controlled by a non runtime-constant.

- At every BRANCH directive, we need to record the not-taken branch for later processing, in the worklist for the current loop.
- At every marker that delineates loop beginnings and ends we need to update the resolution data structure and worklist used to that of the child or parent respectively. With either kind of marker, we need to update the set-up table pointer to point to the appropriate sub-table.

Actions that patch instructions are triggered to fill holes in the template or to patch addresses at jumps. We perform the following peephole optimizations while patching instructions:

- **load elimination/scheduling:** We currently generate template code assuming that a runtime constant used in an arithmetic instruction can fit in the immediate field of the instruction. In the (common) case that this is true, the stitcher simply patches in the value of the runtime constant into the "hole" left in the instruction. In the case that the runtime constant is of "intermediate" size (between 9 and 15 bits), we build it (using a low latency instruction) into a *scratch register* (which is arranged to be left unallocated by the register allocator) before using it. Finally all bigger values are loaded from memory into the scratch register, and the target instruction is changed to use the scratch register instead of an immediate.

  Loading from memory can be expensive: recall that all runtime constants reside initially in a sub-table of the hierarchical set-up table. To load from the set-up table, one would first have to build the address of the sub-table into the scratch register, and then load off this address, a clear performance hit. Further, we expect that most of the runtime constants are small enough that to be placed in immediate fields: keeping around the entire set-up table therefore wastes memory. Our solution is to build a flat table, the address of which is kept in a reserved *table register*, into which each large runtime constant is stored as it is encountered. The stitched code simply performs a load off the table register. The load is currently stitched in before the intermediate instructions are copied (i.e. before step 2 in the above list of stitcher runtime actions), in an effort to hide its latency.
- **strength reduction:** We currently attempt to convert all multiplications by runtime constant integers into a combination of shifts, adds and subtracts, since an integer multiply is significantly more expensive (19-23 cycles) on the Alpha than these single cycle operations. We also convert divides by powers of two into right-shifting operators.
- **inserting prefetching hints:** Since the code segment and data segment are far from each other in the Alpha OS, and the stitched code is constructed in the data segment, we need to convert all relative branches to subroutines into jumps through registers. On the Alpha and many modern architectures, a jump through register (`jsr`) can be an expensive operation, since it almost invariable triggers a miss in the instruction cache. The Alpha offers the option of embedding the high fourteen bits of the target instruction in the `jsr`[SW95] . These bits are used as a hint to start an instruction-cache read a few cycles before the actual target of the jump is computed.

3.4 Runtime overhead inherent in the design

The barebones functionality we expect from the stitcher is to copy all template instructions that don't need patching, and to patch and copy any instructions with holes in them. From the previous subsection on the functioning of the stitcher, however, we can identify at least five sources of *additional* runtime overhead:

1. Directive interpretation overhead.
2. Setup-table initialization and access overhead.
3. Peephole optimization overhead.
4. Memory allocation overhead.
5. Code-alignment overhead.

We pay the first four types of overhead in the current implementation of our framework. The last factor that arises in a simple extension to our system which we considered implementing, but did not due to time constraints; this extension will probably become much more important in a wider-issue machine than the two issue Alpha 21064 we considered, and is therefore worth discussing here. We discuss each of these costs in some detail below.

Directive interpretation overhead is the time required to load the five fields of a stitcher directive, to switch on the opcode of the directive, to push and pop directives off the worklist of directives and to increment the current directive pointer to access the next directive to be executed. This overhead is paid for processing every directive, and as such, could constitute a significant part of the total runtime overhead.

An additional interpreting overhead is paid when the opcode for a directive dispatches to code for a certain *class* of machine operations, rather than that for a single operation. The stitcher-code for these directive-opcodes has to further dispatch on the opcode of the machine-instruction being processed. For instance, we currently have a single STATIC_BRANCH opcode to represent all branch operations guarded by a runtime constant. The opcode of the branch instruction itself may be any of `beq`, `ble`, `blt`, `bne` or `bgt`. The setup table contains the runtime constant on which this branch depends. At stitch time, the stitcher switches on the opcode of the branch instruction to figure out how to compare the runtime constant to zero, in order to determine the branch direction. This problem also arises with the HOLE directive: in order to perform strength reduction, we are interested in distinguishing between various types of holes, and we currently pay the overhead for doing so at stitch time.

A faster solution would split the offending directive into the different types of (more specific) directives. We would then perform the opcode check, and emit the corresponding more specific directive *statically.* In the case of static branches, for instance, we should replace the STATIC_BRANCH directives with directives such as BEQ_STATIC_BRANCH, BLE_STATIC_BRANCH, etc., and have a corresponding code fragment for each of these more specific directives in the stitcher.

If the stitcher and setup code were to be merged, the natural approach would be to make each fragment of stitching code in the merged version be as specific as possible, so that stitching-fragments have no avoidable runtime checks. Since the stitching-fragments for each type of template action will have to be carefully constructed instruction by instruction, the motivation to minimize runtime overhead and thereby keep the fragments as small as possible is high[6].

Setup-table access overhead is of two main types. First, as discussed under the book-keeping overhead of stitching in the previous section, the hierarchical structure of the setup-table implies that the stitcher has to update the pointer to the current sub-table of the setup-table at every entry, exit or continuation of a loop. Second, to process any directive that uses a runtime constant (HOLE and STATIC_BRANCH directives), the stitcher has to load the corresponding runtime constant from the current sub-table. This overhead is intrinsic to our design, since we decided to explicitly separate the setup and stitching phases and have them communicate through the setup table. Note again that the merged setup/stitcher discussed early in section 3.2 as one way to annotate programs does not suffer this memory access overhead: since runtime constant computations are immediately succeeded by the fragment of code that uses them, we can pass runtime constants through registers.

Peephole optimizations have two kinds of associated overhead. First, value-based peephole optimizations (such as strength reduction and load elimination) have to dispatch on the value of the runtime constant they optimize. Specifically, in order to decide whether a hole in an instruction should be patched by an immediate, a build into a register, or a load from table, the stitcher needs to test if the value of the runtime constant lies in the corresponding ranges. Second, computing the optimized version of the code is often time-consuming. Two particularly expensive optimizations we currently perform are strength reduction of integer multiplies into shifts, adds and subtracts and load elimination for small negative integers. In the latter optimization, since the Alpha currently allows only unsigned immediates, we try to replace operations that use a small, negative runtime constant with complementary optimizations that use the absolute value of the runtime constants. Since we expect a large portion of runtime actions to be simple peephole optimizations of the hole-filling type, we need to evaluate carefully the trade-off between the overhead of treating runtime constants of different sizes differently and the gains of faster, specialized code for small runtime constants.

The stitcher needs to **allocate memory** for the book-keeping data-structures it uses, and for the code it stitches (we call the memory into which the stitcher writes *stitch-space*). Ideally, we would like to avoid the runtime overhead of allocating stitch-space dynamically by allocating all the necessary memory statically. However, the need to unroll loops completely makes it difficult to get a static estimate of the amount of memory needed. Since the number of times loop bodies are unrolled is not known statically, the amount of stitch-space needed cannot be determined statically. A key insight into a fast solution to

6. The temptation to move functionality that adversely affects runtime performance (but is easier to implement in the stitcher module) may be a hidden disadvantage of structuring the stitcher as an interpreter written in a high-level language!

this problem is that as long as book-keeping data-structures are re-used between iterations of a fully-unrolled loop, the amount of memory needed for book-keeping will not grow with the number of iterations unrolled. We then need to allocate *only* stitch-space dynamically. Since our current scheme does re-use data-structures between loop-iterations (see section 3.3.1), we could get away with dynamically allocating just the stitch-space.[7] This design would be a good trade-off between the two extremes of *completely static allocation* of all memory (i.e. data structures and stitch-space are *both* statically allocated) which has the advantage of fast runtime behavior, but inflexible runtime limits on the amount of code fixed and sizes of data structures, and completely *dynamic allocation*, which is slower, but more flexible. A hybrid system of this type would still have to:

- Minimize the number of checks to determine if the stitch-space is sufficient, and
- Minimize the amount of stitch-space that remains unused.

A robust but fast stitch-space allocation system would begin with a certain minimum space statically allocated (the amount allocated could be determined by the static size of the template region for instance), and add space dynamically as needed. The subject of how to coalesce stitch-space checks for efficiency needs further study.

Finally, on processors that depend on the precise memory alignment of groups of instructions for good multiple issue (VLIW or superscalars with limited re-order capability), the stitcher needs to ensure that the stitched instructions are **grouped and aligned** correctly. For instance, the Alpha 21064 is capable of dual issue, but only if the first of the two instructions to be issued together is aligned on an 8-byte boundary. Further, the opcodes of the two instructions have to satisfy certain constraints (e.g. both instructions cannot be memory operations). On these processors, the static compiler takes great care to ensure that instruction alignment and grouping is optimal. The introduction of extra instructions at stitch-time can result in severe performance penalties since the alignment of *every* pair of instructions downstream may be destroyed.

A robust implementation of the stitcher should therefore keep track of the alignment and types of the instructions in the group currently being stitched. Since global re-alignment/re-grouping of the stitched instructions is too expensive, the stitcher needs at least to ensure that any instructions it adds or removes are stitched so that the next LIW it stitches will have the correct alignment. Our current implementation takes no special measures to ensure proper alignment of stitched code, and leaves us prone to performance penalties.

To summarize, we expect that our current implementation will have a significant one-time dynamic compilation overhead. On the other hand, except for possible instruction-pair alignment problems, we expect the quality of code to be fairly good: As section 3.1 indicates, barriers to (static) optimization of code within the dynamic region are not very restrictive[8]. The performance analysis of the next section confirms these intuitions.

7. Our current implementation opts for completely static allocation since it is easier to implement, and asserts a program error if the static allocation is exceeded; the amount of memory used is checked only once, just before the stitcher returns, so that the stitcher may well exceed its quota and overwrite adjacent memory.

4.0 Performance analysis

Table 2 gives the results of preliminary tests of our system on small (50-1000 lines of C) kernels. Our tests were conducted on a DEC Alpha 21064 machine (two-way issue, 200 MHz, 8kB on-chip data and instruction caches), and times were measured using an on-chip cycle counter. We measured the following parameters for each program:

- The time taken by the setup code and the stitcher to compute runtime constants and stitch the dynamic region (termed *dynamic compilation overhead* in table 2).
- The number of instructions stitched into the dynamic region, which together with the dynamic compilation overhead above, gives the number of extra *cycles/instruction stitched*.
- The time taken to execute a statically compiled version of the dynamic region and the time to execute the same region when dynamically optimized (the ratio of the former to the latter is termed *asymptotic speedup* in table 2).

The *brakemen point* is the number of times the stitched region has to be re-executed in order to amortize dynamic compilation overhead, and is calculated by dividing the dynamic compilation overhead by the number of cycles saved by each execution of the dynamic region.

TABLE 2. Summary of results

Benchmark	**Run-time Constant Configurations**	**Asymptotic Speedup** (static/ dynamic region times)	**Breakeven Point**	**Dynamic Compilation Overhead**: **set-up & stitcher** (1000s cycles)	**Cycles/ Instruction Stitched** (number of instructions stitched)
Reverse-polish stack-based desk calculator	$2xy - 3y^2 - x^2 + (x+5) * (y-x) + x + y - 1$	**1.7** (1690/997)	**916** interpretations with different x, y values	452 183	**734** (865)
Scalar-matrix multiply (adapted from [EHK96])	800×800 matrix, multiplied by all scalars 1..100	**1.6** (16000/10000)	**31,392** individual multiplications	260 34.3	**4032** (73)
Sparse matrix-vector multiply	200×200 matrix, 10 elements/row, 5% density	**1.8** (76200/43200)	**2645** matrix multiplications	83,700 3,580	**7390** (11810)
	96×96 matrix, 5 elements/row, 5% density	**1.5** (13200/8840)	**1858** matrix multiplications	7,070 1,030	**2478** (3269)
Event dispatcher in an extensible OS[BSP+96]	7 predicate types; 10 different event guards	**1.4** (4606/3228)	**722** event dispatches	638 357	**597** (1667)
Quicksort record sorter (extended from [KEH93])	4 keys, each of a different type	**1.2** (1140/960)	**3050** records	444 105	**8446** (65)
	12 keys, each of a different type	**1.2** (1310/1060)	**4760** records	790 400	**6869** (173)

8. Recall that even when we had to limit certain optimizations to preserve semantics of code containing rtconsts (section 3.1), we only prevented those optimizations on *instructions containing rtconsts*

The high-level goal of maximizing code quality corresponds to maximizing asymptotic speedup. The goal of minimizing dynamic compilation overhead corresponds to minimizing the number of extra cycles/instruction stitched, or equivalently, of minimizing the break-even point. An examination of table 2 shows that while asymptotic speedups are encouraging, the break-even points are unacceptably high. For instance, the stack-based interpreter requires the given polynomial to be evaluated at 916 different points, a situation possible in large corporate spreadsheets, but not likely in home/personal applications.

Table 3 indicates which runtime optimizations contributed to asymptotic speedups in each of the programs. The calculator, event dispatcher and record sorter are classic interpreter-type programs: in each case, the kernel loops over an array of opcodes describing the input problem (the problem description is the string representing the polynomial in the case of the calculator, a representation of guard predicates in the event dispatcher and an array containing the types of each field of the records to be sorted in the sorting program), and dispatches on the type of each opcode to code that implements the opcode. Since the problem descriptions remain fixed during a given execution of each of these programs, the loop over the descriptive array may be *unrolled completely*. For each of the unrolled iterations, we can *eliminate the load* of the opcode from the descriptive array, *eliminate the switch* based on the opcode and *eliminate as dead the code* that deals with the remaining opcodes. In the matrix-scalar multiply program, all performance gains come from replacing the multiply in the innermost loop with shifts, adds and subtracts. Finally, the sparse matrix-vector product kernel multiplies many vectors with a fixed list-based description of the sparse array; unrolling the loop over the list gives the same savings as with the interpreters above, except that no switch statement exists within the loop, so no branch or dead-code elimination results.

Table 3: Optimizations Applied Dynamically

Benchmark	Constant Folding	Static Branch Elimination	Load Elimination	Dead Code Elimination	Complete Loop Unrolling	Strength Reduction
Calculator	√	√	√	√	√	
Scalar-matrix multiply			√			√
Sparse matrix-vector multiply	√		√		√	
Event dispatcher	√	√	√	√	√	
Record sorter	√	√	√	√	√	

Having examined contributions to asymptotic speedup, we now turn our attention to the sources of the high dynamic compilation overhead. Table 2 indicates that setup computations and stitching make roughly equal contributions to the overhead. In order to understand the source of runtime overhead better, we modified gdb to count the instructions executed in three of the faster runs. Further, we counted the number of stitcher instructions spent interpreting directives; we estimated this number by counting the number of stitcher instructions devoted to reading in, dispatching on, and finding the next directive, for each kind of directive, multiplying these numbers by the number of times that directive executed, and summing the product for each category of instructions over all directives. Table 4 lists the instructions/

instruction stitched and the percentage of instructions spent interpreting directives, for the three programs:

TABLE 4. Instruction count measurements of dynamic compilation overhead

	Instructions exec'd/ Instructions stitched (setup + stitcher)	**CPI (setup +stitcher)**	**% Instructions spent interpreting directives**
reverse-polish calculator	162 (37 + 125)	4.5 (14.2+1.7)	5.6
matrix/scalar multiply	120 (averaged over all runs) (40 + 80)	33.6 (80 + 4.9)	5.2
quicksort with 4 keys	331 (101 + 330)	25.6 (67.4+7.0)	7.6

The CPI for the dynamic compiler are clearly much higher than expected. To find out where the extra cycles were going, we instrumented the stitcher and the setup to look for code that consumed more cycles than expected. For the quicksort program, we identified two major consumers of cycles. In both cases, gdb does not count any instructions as being executed while the overhead cycles are incurred. The cycles are therefore probably being spent within the kernel (these interrupt-servicing instructions are presumably not counted by GDB), fetching and zeroing newly accessed pages.

- 73% of the total dynamic compilation overhead cycles were spent inside calls to `malloc()` in setup-code, allocating sub-tables of the runtime constants table.
- 13% of the overhead cycles are incurred when touching statically allocate memory in the stitcher for the first time. This is memory allocated for the stitched code, label-resolution data-structures and the directive-worklist data-structure.

Avoiding repeated calls to `malloc()` by using a customized memory managed for the sub-tables of the runtime constants table should eliminate most of first overhead. The second overhead may be more difficult to conquer. On the Alpha 21064, for instance, the page-size is 8kB; at 4 bytes an instruction, we can stitch 2048 instructions onto a page; however, zeroing the page will take (assuming quad-word stores from the zero register) at least four instructions (including one store) per instruction stitched, assuming a cache hit and that the page doesn't have to be brought in from disk. This overhead is paid even before we begin stitching instructions.

5.0 Related work

We examine two systems for automatic value-specific dynamic compilation, and one for manually directed runtime optimization, paying close attention to the techniques they use to minimize runtime overhead while maintaining code quality, the extent to which they succeed in doing so, and the lessons we can learn from these systems.

The system most like ours is the Tempo system of Consel and Noel [CN96]. They support value-specific dynamic optimization of C programs, based on user-annotations. They too have a BTA which identifies template code, which is isolated and compiled into machine code that is copied and patched at runtime. Since their BTA is described in terms of abstract syntax trees (ASTs), it is not clear how they handle code that cannot be described by ASTs. They have avoided the interpretive overhead of separating the stitcher from the "setup" code that computes runtime constants, by merging the two into a single "generating extension" (a term borrowed from the Partial Evaluation literature). Further, since they only deal with code that can be described by ASTs, they can statically identify a linear order in which basic blocks of the dynamic region need to be stitched. They can do away with the "to-be-stitched" worklist structure we have and avoid the runtime overhead of maintaining this structure. Their dynamic compilation overhead should therefore be lower than ours. However, the quality of code they generate is probably not as good as that of ours, since they do not modify the compiler to support their system. Their approach is to split template from setup code and produce the generating extension, all at the *source code* level. Not modifying the compiler limits significantly the extent to which their template code can be optimized statically. As we showed in section 3.1, many key conventional optimization phases (e.g., copy propagation) need modification if their transformations are to preserve the correctness of the code templates. Consel and Noel currently turn off these optimizations on the template code, thus trading off some template code quality for convenience. No performance numbers are available as yet on the system. It would be interesting to see to what extent turning off optimizations affects their performance.

Another automatic dynamic compilation system is Leone and Lee's Fabius system[LL96], which performs dynamic compilation on a strictly functional subset of ML. Functions are written in an explicitly curried form, and a more specialized version of the function is dynamically compiled each time a new argument is available. The system therefore does not depend on the correctness of user annotations for its correctness. They too have merged setup code with stitch code, and handle only nested control structure, leading to similar savings in runtime overhead as with Consel and Noel above. Their approach is not template-based, in that they do not statically collect and globally optimize all the code that will be generated at runtime[9]. On the other hand, every dynamically generated instruction is *built* into a register and stored into the stitched region, thus avoiding the load that copying from a template would involve. Template-based systems like ours can avoid this extra load too: if the setup and stitcher code were merged, instead of constructing (at static compile time) a fragment of stitcher code to *copy* an instruction form the template, we could construct a fragment that *builds* that instruction, when this proves cheaper. Leone and Lee report extremely low dynamic compilation overhead (an average of six instructions per

9. They **do** have a modified register allocation algorithm to globally allocate registers across instructions labelled for runtime generation, *independently* of allocation for "setup" instructions, even though the two types of code are interleaved. This technique leads to an interesting observation: to have a system that optimizes code to be stitched (template code) independently of code to be executed immediately (set-up code), it is not necessary to physically separate the template code from the setup. An alternate technique might be to modify global optimization algorithms to optimize instructions marked as "template" independently of those marked as "setup" though these two kinds of code may be mixed in the dynamic region.

instruction stitched), but it is more difficult to evaluate the quality of their code. Runtime-compiled versions of ML code clearly run significantly run faster than non-runtime compiled versions; however, at least part of the reason for this speedup is that the runtime-compiled version avoids ML-specific overheads such as array-bounds checking. In comparisons with optimized C code, dynamically compiled code performed almost as well as optimized C for matrix multiply, and outperformed C by 80% asymptotically in a small packet-filter routine. The latter benchmark was too small to estimate if the lack of global optimization proved not to be a problem.

Finally, the ‘C system of Engler and colleagues [EHK96] allows the programmer to explicitly specify how pieces of code should be constructed and executed at runtime. Since the pieces of code to be composed into a single executable fragment are determined only at runtime, ‘C does not perform any static global optimizations across code to be stitched either. The latest implementation of ‘C (on the tcc compiler) seems to have low runtime overhead (break-even points seem low, though no cycles per instructions stitched figure is mentioned). However, asymptotic speedup is calculated relative to C programs with no global optimization or scheduling, so the effect of the lack of global optimizations on code quality is difficult to gauge.

To summarize, merging stitcher and setup code seems to be a common practice, and seems to correlate well with reduced runtime overheads. On the other hand, the importance of static global optimization on the performance of stitched code is not clear. Since the global optimization of stitched code is an important feature of our approach, we need to study it closely to either establish its importance or consider the implications of reduced static optimization on the design of our system. Optimizing the template code is not without its cost in complexity of system-design. For instance, producing merged setup/stitcher code after separating template code from setup code and *separately* optimizing the template is not straightforward to implement. In particular, global optimizers such as profile-based schedulers and loop-unrolling modules can change the order of instructions and the structure of the CFG for the dynamic region, so that finding the program point in the stitcher “corresponding” to a template code point, in order to insert a stitcher fragment, may be non-trivial. On the other hand, it seems likely that the importance of optimization will increase as stitched regions become bigger and more complex than they currently are, and as dynamic compilation systems are ported to wider machine that benefit more from optimization.

6.0 Future work

Directions for future work on the dynamic compilation system include the following:

- Speeding up the dynamic compiler.
- Simplifying the design of the dynamic compiler related extensions to the static compiler.
- Evaluating the performance of the system on large programs.
- Expanding the repertoire of optimizations performed at runtime.

- Automating the identification of dynamic regions and runtime constants.
- Exploring ways to hide the dynamic compilation overhead.

Each of these areas is discussed in the sub-sections below.

6.1 Speeding up dynamic compilation

It is clear from the results section that the overhead of dynamic compilation in our current system is high. Our goal of performing better performance is achieved only for systems that execute the dynamic region a very large number of times. We discuss below some techniques that should lower the break-even point. We use the list of section 3.3.3 (sources of runtime-overhead inherent in our system), and observations from section 4 (results) to identify possible improvements.

The use of the set-up table in order to communicate values between setup-code and the stitcher is the single largest source of runtime overhead. Most of the overhead comes from calls to `malloc` to create each sub-table of the set-up table. A simple, customized memory management routine which allocates from a large, statically allocated pool would eliminate most of this overhead.

A more comprehensive solution that would eliminate the overhead of accessing the table and that of interpreting the stitcher directives would be to **merge the setup-code with the stitcher**, as suggested in section 3.2. The broad idea is that code that uses a runtime constant would be stitched as soon as the setup code that computes it is executed; the values of the runtime constants can then be passed in temporaries (instead of a table in memory) to the subsequent stitching code. The primary difficulty in merging is that if setup and template code are optimized separately, as they currently are, the position and order of the template-instructions that use runtime constants could be significantly different from the order in which they are computed in the setup code. Since the order of computation in the setup-code is often fixed by data-dependences, it may be necessary to restrict the re-order of instructions in the template code.

As detailed in section 3.3.2, the stitcher performs a few peephole optimizations. These optimizations have at least two types of costs associated with them, which can be reduced by further work:

- The **cost of determining whether the current instruction is eligible** for peephole optimization, which is generally the result of a stitcher directive that is not specific enough. For instance, if a HOLE directive applies to an instruction, we need to check if that instruction is a multiply in order to determine whether a conversion to shift-and-adds is applicable (note that this check needs to be performed at *all* HOLEs). A solution would be to have the directive for holes with multiplies in them be different from that for other holes (i.e., have *more specific directives*).
- The **cost of performing the optimization**, which depends on the specific optimization being performed. Converting a multiply by an arbitrary integer into shifts, adds and subtracts can be slow, whereas inserting prefetching hints is fast. We should benefit by doing slower optimizations only when they are likely to be profitable. One way to limit

the use of peephole optimizations is to perform more aggressive optimizations only on paths that *static profiling* runs indicate are frequently used. Limiting the scope of optimizations will be even more important when more ambitious optimizations such as register allocation and scheduling are considered (see section 6.4 below).

Finally, we need to explore the extent to which on-demand (or *lazy*) stitching can benefit us. The idea here is to stitch a unit of template code only when it is about to be executed, so that template-code that is never executed is never stitched. Infrastructure for stitching code lazily would include mechanisms for jumping to the stitcher on demand, detecting when a piece of code has already been stitched, and restoring context for resuming execution when transitioning from "execute" to "stitch" mode, and back. Lazy stitching could be especially important in the case of multi-way loop unrolling (see the footnote on page 11), where we can possibly avoid stitching in loop iterations corresponding to a set of induction variable values unless that iteration of the loop needs to execute.

6.2 Simplifying the design of the dynamic compiler extensions

Clever placement of the BTA phase with respect to the other optimization phases in the compiler could reduce the extent to which the other optimization phases need to be modified in order to handle runtime constants. Section 3.1 lays the framework for deciding which optimizations need to precede the BTA and which need to succeed it.

As mentioned at the end of section 5, producing "merged" setup/stitcher code for templates that undergo arbitrary optimizations (specifically, the reordering and moving of instructions containing runtime constants) can be difficult. We need to examine to what extent we can turn down the level of optimization, so that, while performance of stitched code is not much affected, the placement and order of template holes follows some rules we can use in constructing merged setup/stitcher code. On the other hand, we need to keep in mind claims that aggressive global scheduling methods can increase performance on wide machines such as the Alpha 21164 (4-wide) and the PA-8000 (6-wide) by factors of 2 or more.

6.3 Handling large programs

The results in this paper are all based on measuring small programs. We need to examine large, widely used programs closely in order to validate our design. Of particular interest is the identification of classes of programs that have a high enough density of high-frequency operations that can be eliminated/strength-reduced using our techniques. Based on the effort required to compile the (moderately large) `dinero` cache simulator using our system, it seems that the following capabilities should aid in studying large programs:

- A prior idea of which variables are likely to be runtime constants (for instance, we may know that the structure describing the cache will not be modified at runtime in a cache simulator).

- The ability to profile programs in order to identify high-frequency parts that contain a high density of runtime constants. We should be able to adapt the DEC profiling tool ATOM for this propose. Note that we would like program-point level profiling rather than the function-level profiling that a tool like gprof provides.
- An "oracle", which given a specific set of annotations on a program, will run the program, record the values of runtime constants computed, inline these values as constants in the program and perform static value-based optimizations on the program. The resulting executable should provide an upper limit on the quality of code the dynamic compiler can produce.
- Fast watch-points to check if a variable declared to be a runtime constant is written to in a specific part of the program. Gdb currently allows us to do this, but is very slow. An alternate facility may be the ability to ask the BTA to notify the user if a given variable (suspected to be a runtime constant) is killed from the set of runtime constants being propagated by the BTA, within a specified part of the program.

6.4 Performing more optimizations at runtime

Examining large programs may reveal that some currently un-implemented runtime optimizations are essential for good performance in certain contexts. Optimizations that could be profitable include:

- **Run-time register allocation** of memory locations that are found to be "hot-spots" at runtime. It is unclear if current profiling tools can capture memory usage patterns cheaply. If so, we can allocate "left-over" registers (i.e. registers that have not been allocated within a given part of the program) to these frequently used memory locations[10]; note that the static register allocator can notify the stitcher of the identity of unallocated registers, a technique similar to that of Wall in his link-time register allocator [Wal86].
- **Runtime inlining** of functions within dynamic regions, especially small leaf routines whose identity may not be known at static compile time (e.g. accessor methods for objects, dynamically linked applets).
- **Runtime scheduling** of high-latency operations. One of the main benefits of conventional (static) loop unrolling is that operations can be scheduled between the iterations of the unrolled loop. Operations from one iteration can be hoisted into code for an earlier iteration, possibly reducing the length of the critical path through the loop. Our current implementation of runtime loop-unrolling performs no scheduling between iterations. We may benefit substantially from re-scheduling at least high-latency operations, such as those for memory, multiplication and division. An interesting aspect of scheduling at runtime is that a major portion of traditional scheduling, computing data-dependences, can be done statically, so that the runtime scheduler can concentrate on resource allocation.

10. Markus Mock manually inserted register-action directives in the stack-based RPN calculator example. The register actions assigned locations 0 to 2 of the RPN stack to registers and produced an asymptotic speedup of 4.1x, up from the 1.7x speedup reported in table 2.

As with peephole optimizations, the possibly high runtime cost of these optimizations means that they should be performed only on code that is likely to be executed frequently enough that the cost is repaid. These optimizations should therefore be limited to code that is on-trace (i.e. code determined by static profiling runs to be high-frequency), or in the absence of reliable static profile information, to code that has been found by some low-overhead dynamic profiling method to be executed frequently.

6.5 Techniques to hide dynamic compilation overhead

Some domains may permit us to overlap dynamic compilation with program execution. Given the right operating-system support (i.e. the ability to communicate values fast between two threads that share a data space), and true, lightweight concurrency in hardware (offered, for instance, by a simultaneous multi-threading processor), we should be able to communicate the values of the root runtime-constants to a concurrent thread that already has setup and template code read into memory. The new thread could then stitch the templates together while the original thread continues to execute the un-dynamically-optimized version of the program[11]. Even on a conventional processor, dynamic compilation could commence on a separate thread while the original thread is waiting for I/O.

7.0 Conclusions

In this paper, we have described and analyzed a system for performing value-specific optimizations on C programs at runtime. We have described techniques for modifying optimization phases of an existing compiler to support dynamic compilation and for communicating the nature of this work to the dynamic compiler via a high-level, modular interface. We have reported preliminary results from this system: kernels of size 50-1000 lines have shown speedups ranging from 1.8x to 1.2x, with relatively high runtime overhead. We have analyzed the source of the speedups, and discuss ways to decrease the overhead of the system. We have compared our system with selected related systems. Finally, we have suggested future steps towards further validating our system, broadening its scope of applicability and improving its performance. Our techniques for value-specific runtime optimization have proved profitable on small programs: we now need to prove their applicability on large, widely used programs.

References

[APC+96] J. Auslander, M. Philipose, C. Chambers, S.J. Eggers and B.N. Bershad. Fast, Effective Dynamic Compilation. In *Conference on Programming Language Design and Implementation,* February 1996.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and*

11.The lightweight integration of compilation with computation may prove to be a way in which an SMT architecture can "win a single thread benchmark competition". Forest Baskett (`http://www.cise.nsf.gov/mips/MSAWorkshop96/Baskett.htm`) claims the lack of such a win is an impediment to industrial interest in SMT.

Tools. Addison-Wesley, 1986.

[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.

[CEA+96] C. Chambers, S.J. Eggers, J. Auslander, M. Philipose, M. Mock, and P. Pardyak. Automatic dynamic compilation support for event dispatching in extensible systems. In *Workshop on Compiler Support for Systems Software*, February 1996.

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

[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.

[KEH93] D. Keppel, S.J. Eggers, and R.R. Henry. Evaluating runtime-compiled, value-specific optimizations. Technical Report 93-11-02, University of Washington, Department of Computer Science & Engineering, 1993.

[Kep96] D. Keppel. Runtime code generation. Technical report, University of Washington, Department of Computer Science & Engineering, 1996.

[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.

[PEK96] M. Poletto, D.R. Engler, and M.F. Kaashoek. tcc: a template-based compiler for ‘C. In *Workshop on Compiler Support for Systems Software*, February 1996.

[SK95] M. Schlansker and V. Kathail, Critical Path Reduction for Scalar Programs, *Proceedings of the 28th Annual International Symposium on Microarchitecture*, November 1995.

[SW95] R. L. Sites and R.T. Witek. *Alpha AXP Architecture Reference Manual*. Butterworth-Heinemann, 1995.

[Wal86] D.W. Wall. Global register allocation at link time. In *Symposium on Compiler Construction*, June 1986.