An Infrastructure for Profile-Driven Dynamic Recompilation*

Robert G. Burger
SAGIAN
A Division of Beckman Coulter
P. O. Box 78668
Indianapolis, IN 46278
burgerrg@sagian.com

R. Kent Dybvig
Computer Science Department
Indiana University
Lindley Hall 415
Bloomington, IN 47405
dyb@cs.indiana.edu

Abstract

Dynamic optimization of computer programs can dramatically improve their performance on a variety of applications. This paper presents an efficient infrastructure for dynamic recompilation that can support a wide range of dynamic optimizations including profile-driven optimizations. The infrastructure allows any section of code to be optimized and regenerated on-the-fly, even code for currently active procedures. The infrastructure incorporates a low-overhead edge-count profiling strategy that supports first-class continuations and reinstrumentation of active procedures. Profiling instrumentation can be added and removed dynamically, and the data can be displayed graphically in terms of the original source to provide useful feedback to the programmer.

1 Introduction

In the traditional model of program optimization and compilation, a program is optimized and compiled once, prior to execution. This allows the cost of program optimization and compilation to be amortized, possibly, over may program runs. On the other hand, it prevents the compiler from exploiting properties of the program, its input, and its execution environment that can be determined only at run time. Recent research has shown that dynamic compilation can dramatically improve the performance of a wide range of applications including network packet demultiplexing, sparse matrix computations, pattern matching, and mobile code [10, 8, 13, 16, 22, 23]. Dynamic optimization works when a program is staged in such a way that the cost of dynamic recompilation can be amortized over many runs of the optimized code [21].

This paper presents an infrastructure for dynamic recompilation of computer programs that permits optimization of code at run time. The infrastructure allows any section of code to be optimized and regenerated on-the-fly, even the code for currently active procedures. In our Scheme-based implementation of the infrastructure, the garbage collector is used to find and relocate all references to recompiled procedures, including return addresses in the active portion of the control stack.

Since many optimizations benefit from profiling information, we have included support for profiling as an integral part of the recompilation infrastructure. Instrumentation for profiling can be inserted or removed dynamically, again by regenerating code on-the-fly. A variant of Ball and Larus's low-overhead edge-count profiling strategy [2], extended to support first-class continuations and reinstrumentation of active procedures, is used to obtain accurate execution counts for all basic blocks in instrumented code. Although profiling overhead is fairly low, profiling is typically enabled only for a portion of a program run, allowing subsequent execution to benefit from optizations guided by the profiling information without the profiling overhead.

A side benefit of the profiling support is that profile data can be made available to the programmer, even as a program is executing. In our implementation, the programmer can view profile data graphically in terms of the original source (possibly split over many files) to identify the "hot spots" in a program. The source is color-coded according to execution frequency, and the programmer can "zoom in" on portions of the program or portions of the frequency range.

As a proof of concept, we have used the recompilation infrastructure and profiling information to support run-time reordering of basic blocks to reduce the number mispredicted branches and instruction cache misses, using a variant of Pettis and Hansen's basic-block reordering algorithm [24].

The mechanisms described in this paper are directly applicable to garbage-collected languages such as Java, ML,

*This material is based on work supported in part by the National Science Foundation under grant numbers CDA-9312614 and CCR-9711269. Robert G. Burger was supported in part by a National Science Foundation Graduate Research Fellowship.

Scheme, and Smalltalk in which all references to a procedure may be found and relocated at run time. The mechanisms can be adapted to languages like C and Fortran in which storage management is done manually by maintaining a level of indirection for all procedure entry points and return addresses. Although a level of indirection is common for entry points to support dynamic linking and shared libraries, the level of indirection for return addresses would be a cost incurred entirely to support run-time recompilation.

Section 2 describes the edge-count profiling algorithm incorporated into the dynamic recompilation infrastructure and briefly discusses how profile data is associated with source code and presented to the programmer. Section 3 presents the infrastructure in detail and its proof-of-concept application to basic-block reordering based on profile information. Section 4 gives some performance data for the profiler and dynamic recompiler. Section 5 describes related work. Section 6 summarizes our results and discusses future work.

2 Edge-Count Profiling

This section describes a low-overhead edge-count profiling strategy based on one described by Ball and Larus [2]. Like Ball and Larus's, it minimizes the total number of profile counter increments at run time. Unlike Ball and Larus's, it supports first-class continuations and reinstrumentation of active procedures. Additionally, our strategy employs a fast log-linear algorithm to determine optimal counter placement. The recompiler uses the profiling information to guide optimization, and it may also use the data to optimize counter placement, as described in Section 3. The programmer can view the data graphically in terms of the original source to identify the "hot spots" in a program.

Section 2.1 summarizes Ball and Larus's optimal edge-count placement algorithm. Section 2.2 presents modifications to support first-class continuations and reinstrumentation of active procedures. Section 2.3 describes our implementation. Section 2.4 explains how the profile data is correlated with the original source.

2.1 Background

Figure 1 illustrates Ball and Larus's optimal edge-count placement algorithm using a Scheme procedure that removes all occurrences of a given item from a list.

A procedure is represented by a control-flow graph composed of basic blocks and weighted edges. The assembly language instructions for the procedure are split into basic blocks, which are sequential sections of code for which control enters only at the top and exits only from the bottom. The branches at the bottoms of the basic blocks determine how the blocks are connected, so they become

```
(define remq
  (lambda (x ls)
    (cond
     [(null? ls) '()]
     [(eq? (car ls) x) (remq x (cdr ls))]
     [else (cons (car ls) (remq x (cdr ls)))])))
```

	bne	arg2, *empty-list*, L1	B1
	ld	ac, *empty-list*	B2
	jmp	ret	
L1:	ld	ac, *car-offset*(arg2)	B3
	bne	ac, arg1, L2	
	ld	arg2, *cdr-offset*(arg2)	B4
	jmp	reloc *remq*	
L2:	st	arg2, 4(fp)	B5
	st	ret, 0(fp)	
	ld	arg2, *cdr-offset*(arg2)	
	ld	ret, L3	
	add	fp, 8, fp	
	jmp	reloc *remq*	
L3:	sub	fp, 8, fp	B6
	ld	arg2, 4(fp)	
	ld	ret, 0(fp)	
	ld	arg1, ac	
	ld	ac, *car-offset*(arg2)	
	sub	ap, *car-offset*, xp	
	add	ap, 8, ap	
	st	ac, *car-offset*(xp)	
	st	arg1, *cdr-offset*(xp)	
	ld	ac, xp	
	jmp	ret	

```
> (remq 'b '(b a b c d))
| (remq b (b a b c d))
| (remq b (a b c d))
| | (remq b (b c d))
| | (remq b (c d))
| | | (remq b (d))
| | | | (remq b ())
| | | | ()
| | | (d)
| | (c d)
| (a c d)
(a c d)
```

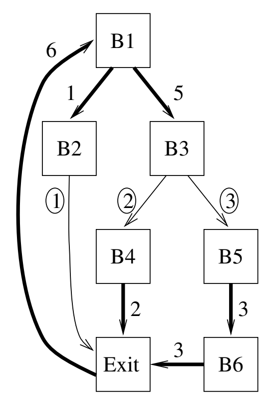


Figure 1: *remq* source, basic blocks, sample trace, and control-flow graph with thick, uninstrumented edges from one of the twelve maximal spanning trees and encircled counts for the remaining, instrumented edges.

```
(define fact
  (lambda (n done)
    (if (< n 2)
        (done 1)
        (* n (fact (- n 1) done)))))
```

	bge	arg1, *fixnum* 2, L1	B1
	ld	cp, arg2	B2
	ld	arg1, *fixnum* 1	
	jmp	*entry-offset*(cp)	
L1:	st	arg1, 4(fp)	B3
	st	ret, 0(fp)	
	sub	arg1, *fixnum* 1, arg1	
	add	fp, 8, fp	
	ld	ret, L2	
	jmp	reloc *fact*	
L2:	sub	fp, 8, fp	B4
	ld	arg1, 4(fp)	
	ld	ret, 0(fp)	
	ld	arg2, ac	
	jmp	reloc ∗	

Figure 2: Source and basic blocks for *fact*, a variant of the factorial function which invokes the *done* procedure to compute the value of the base case.

```
> (call/cc
    (lambda (k)
      (fact 5 k)))
| (fact 5 <proc>)
| | (fact 4 <proc>)
| | | (fact 3 <proc>)
| | | | (fact 2 <proc>)
| | | | | (fact 1 <proc>)
1
```

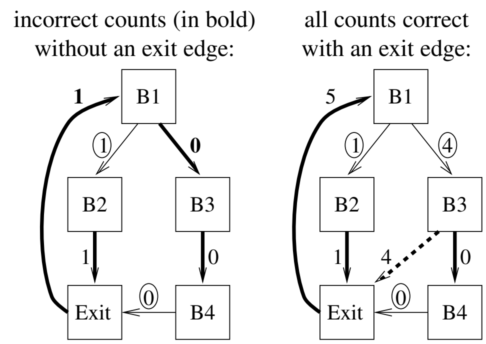


Figure 3: Trace and control-flow graphs illustrating nonlocal exit from *fact* with and without an exit edge.

the edges in the graph. The weight of each edge represents the number of times the corresponding branch is taken.

The key property needed for optimal profiling is conservation of flow, i.e., the sum of the flow coming into a basic block is equal to the sum of the flow going out of it. In order for the control-flow graph to satisfy this property, it must represent all possible control paths. Consequently, a virtual "exit" block is added so that all exits are explicitly represented as edges to the exit block. Entry to the procedure is explicitly represented by an edge from the exit block to the entry block, and the weight of this edge represents the number of times the procedure is invoked.

Because the augmented control-flow graph satisfies the conservation of flow property, it is not necessary to instrument all the edges. Instead, the weights for many of the edges can be computed from the weights of the other edges using addition and subtraction, provided that the uninstrumented edges do not form a cycle. The largest cycle-free set of edges is a spanning tree. Since the sum of the weights of the uninstrumented edges represents the savings in counting, a maximal spanning tree determines the inverse of the lowest-cost set of edges to instrument.

The edge from the exit block to the entry block does not correspond to an actual instruction in the procedure, so it cannot be instrumented. Consequently, the maximal spanning tree algorithm is seeded with this edge, and the resulting spanning tree is still maximal [2].

2.2 Control-Flow Aberrations

The conservation of flow property is met only under the assumption that each procedure call returns exactly once. First-class continuations, however, cause some procedure activations to exit prematurely and others to be reinstated one or more times. Even when there are no continuations, reinstrumenting a procedure while it has activations on the stack is problematic because some of its calls have not returned yet.

Figure 2 gives a variant of the factorial function that illustrates the effects of nonlocal exit and re-entry using Scheme's *call-with-current-continuation* (*call/cc*) function [11].

Ball and Larus handle the restricted class of exit-only continuations, e.g., *setjmp*/*longjmp* in C, by adding to each call block an edge pointing to the exit block. The weight of an exit edge represents the number of times its associated call exits prematurely. Figure 3 demonstrates why exit edges are needed for exit-only continuations. Ball and Larus measure the weights of exit edges directly by modifying *longjmp* to account for aborted activations. We use a similar strategy, but to avoid the linear stack walk overhead, we measure the weights indirectly by ensuring that exit edges are on the spanning tree (see Section 2.3).

With fully general continuations, the weight of an exit edge represents the net number of premature exits, and a negative weight indicates reinstatement. Figure 4 demonstrates the utility of exit edges for continuations that reinstate procedure activations.

```
> (fact 4
    (lambda (n)
      (call/cc
        (lambda (k)
          (set! redo k)
          (k n)))))
```

```
| (fact 4 <proc>)
| | (fact 3 <proc>)
| | | (fact 2 <proc>)
| | | | (fact 1 <proc>)
| | | | 1
| | | 2
| | 6
| 24
24
```

```
> (redo 2)
| | | | 2
| | | 4
| | 12
| 48
48
```

incorrect counts (in bold) without an exit edge: all counts correct with an exit edge:

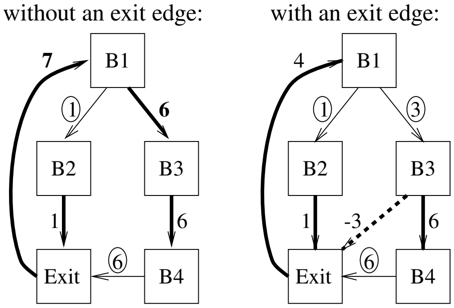


Figure 4: Trace and control-flow graphs illustrating reentry into *fact* with and without an exit edge.

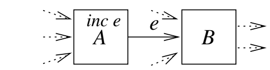
2.3 Implementation

To support profiling, the compiler organizes the generated code for each procedure into basic blocks linked by edges that represent the flow of control within the procedure. It adds the virtual exit block and corresponding edges and seeds the spanning tree with the edge from the exit block to the entry block as described in Section 2.1. It then computes the maximal spanning tree and assigns counters to all edges not on the maximal spanning tree. Finally, the compiler inserts counter increments into the generated code, placing them into existing blocks whenever possible.

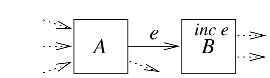
For efficiency, our compiler uses the priority-first search algorithm for finding a maximal spanning tree [27]. Its worst-case behavior is $O((E + B)\log B)$, where B is the number of blocks and E is the number of edges. Since each block has no more than two outgoing edges, E is $O(B)$. Consequently, the priority-first algorithm performs very well with a worst-case behavior of $O(B \log B)$.

Another benefit of this algorithm is that it adds uninstrumented edges to the tree in precisely the reverse order for which their weights need to be computed using the conservation of flow property. As a result, count propagation does not require a separate depth-first search as described in [2]. Instead, the maximal spanning tree algorithm generates the list used to propagate the counts quickly and easily. This list is especially important to the garbage collector, which must propagate the counts of recompiled procedures (see Sections 3.1 and 3.3).

If A's only outgoing edge is e, put the increment code in A.



If B's only incoming edge is e (and B is not the exit block), put the increment code in B.



Otherwise, put the increment code in a new block C that is spliced into the graph.

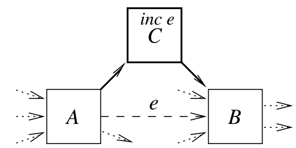


Figure 5: Efficient instrumentation of edges.

Figure 5 illustrates how our compiler minimizes the number of additional blocks needed to increment counters by placing as many increments as possible in existing blocks. The increment instructions refer to the edge data structures by actual address.

Instrumenting exit edges is more difficult because there are no branches in the procedure associated with them. We solved this problem for the edge from the exit block to the entry block by seeding the maximal spanning tree algorithm with this edge and proving that the resulting tree is still maximal. Unfortunately, exit edges rarely lie on a maximal spanning tree because their weights are usually zero. Consequently, there are two choices for measuring the weights of exit edges.

First, we could modify continuation invocation to update the weights directly. Ball and Larus use this technique for exit-only continuations by incrementing the weights of the exit edges associated with each activation that will exit prematurely [2]. This approach would support fully general continuations if it would also decrement the weights of the exit edges associated with each activation that would be reinstated. The pointer to the exit edge would have to be stored either in each activation record or in a static location associated with the return address.

Second, we could seed the maximal spanning tree algorithm with all the exit edges. The resulting spanning tree might not be maximal, but it would be maximal among spanning trees that include all the exit edges.

Our system's segmented stack implementation supports constant-time continuation invocation [17, 5]. Implementing the first approach would destroy this property. Moreover, if any procedure is profiled, the system must traverse all activations to be sure it finds all profiled ones. Although the second approach increases profiling overhead, it affects only profiled procedures, the overhead is still reasonable, and the programmer may turn off accurate continuation profiling when it is not needed. Consequently, we implemented only the second approach.

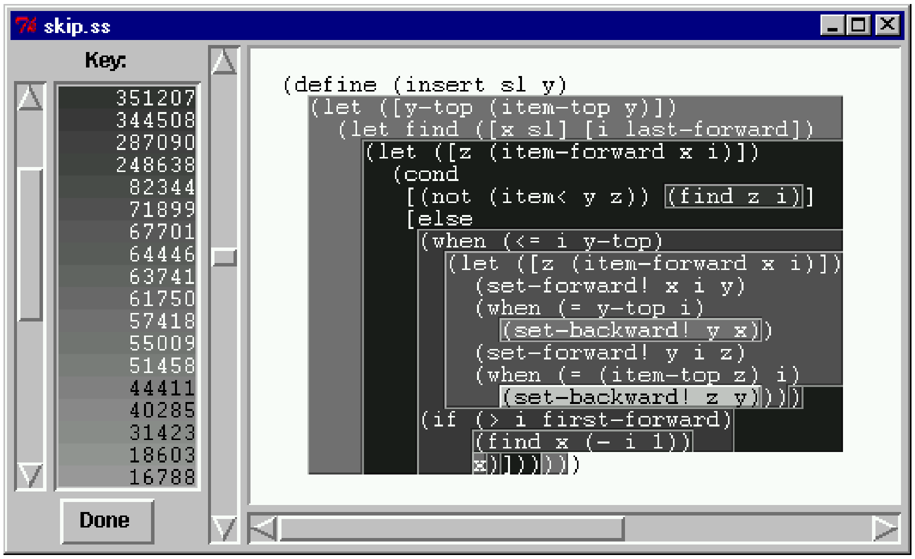


Figure 6: A section of code displayed using darker shades for more frequently executed code and lighter shades for less frequently executed code.

2.4 Graphical Display of Profile Information

Although the profile data is intended primarily for use by the dynamic recompiler, it is also useful for providing feedback to the programmer. In order to provide useful feedback, some way of associating profile data with source code is needed. The compiler attaches a source record containing a source file descriptor and a byte offset from the beginning of the file to each source expression as it is read. The compiler propagates these source records through the intermediate compilation passes and associates each source record with an appropriate basic block. Within the representation of a basic block, each source record associated with the block is included within a special "source" pseudo instruction. These pseudo instructions do not result in the generation of any additional machine code.

The default location for an expression's source record is in the basic block that begins its evaluation. For procedure calls, however, the source expression corresponds to the basic block that makes the call. In most situations, the count for this block is the same as the count for the block that begins to evaluate the entire call expression. A difference arises when continuations are involved, and in this case we have found it more useful to know how many times the call is actually made.

Each constant and variable reference occurs in just one basic block, so the source record goes there. When a constant or reference occurs in nontail position, its count can usually be determined from the count of the closest enclosing expression. An exception arises when the constant or reference occurs as the "then" or "else" part of an **if** expression in nontail position. Consequently, the compiler generates source instructions for constants and references only when they occur in tail position or as the "then" or "else" part of an **if** expression. By eliminating the source instructions for the remaining cases, the compiler significantly reduces the number of source records stored in basic blocks without sacrificing useful information.

To display the block counts in terms of the original source, we follow three steps. First, we determine the count for each block by summing the weights of all its outgoing edges. Second, we build an association list of source expressions and block counts from the source records stored in each basic block. Third, we use this list to determine the source files that must be displayed by the graphical user interface and to guide a color-coding of the code according to the counts. The programmer can "zoom in" on portions of the program or portions of the frequency range and can also click on an expression to obtain its precise count. Figure 6 gives an example.

The programmer is not limited to displaying information for one procedure at a time. The data for all profiled procedures can be displayed at one time with a separate window for each source file. The data is sorted by frequency to help programmers identify hot spots. This technique proved useful in profiling the profiler itself, helping us identify inefficiencies in the maximal spanning tree and block look-up algorithms.

3 Dynamic Recompilation

This section presents the dynamic recompilation infrastructure. Section 3.1 gives an overview of the recompilation process. Section 3.2 describes how the representation of procedures is modified to accommodate dynamic recompilation. Section 3.3 describes how the garbage collector uses the modified representation to replace code objects with recompiled ones. Section 3.4 presents the recompilation process, which uses a variant of Pettis and Hansen's basic-block reordering algorithm to reduce the number of mispredicted branches and instruction cache misses [24].

3.1 Overview

Dynamic recompilation proceeds in three phases. First, the candidate procedures are identified, either by the user or by a program that selects among all the procedures in the heap. Second, these procedures are recompiled and linked to separate, new procedures. Third, the original procedures are replaced by the new ones during the next garbage collection.

Because a procedure's entry and return points may change during recompilation, the recompiler creates a translation table that associates the entry- and return-point offsets of the original and the new procedure and attaches it to the original procedure. The collector uses this table to relocate call instructions and return addresses. Because the collector translates return addresses, procedures can be recompiled while they are executing.

Before the next collection, only the original procedures are used. This invariant allows each original procedure to share its control-flow graph and associated profile counts, if profiling is enabled, with the new procedure. Because the new procedure's maximal spanning tree may be different from the original's, the new procedure may increment different counts. The collector accounts for the difference by propagating the counts of the original procedure so that the new procedure starts with a complete set of accurate counts.

3.2 Representation

Figure 7 illustrates our representation of procedures, highlighting the minor changes needed to support dynamic recompilation. A more detailed description of our object representation is given elsewhere [14].

Procedures are represented as closures and code objects. A closure is a variable-length array whose first element is the address of the procedure's anonymous entry point and whose remaining elements are the values of the procedure's free variables. The anonymous entry point is the first instruction of the procedure's machine code, which is stored in a code object. A code object has a six-word header before the machine code. The info field stores the code object's block structure and debugging information.

The relocation table is used by the garbage collector to relocate items stored in the instruction stream. Each item has an entry that specifies the item's offset within the code stream (the code offset), the offset from the item's address to the address actually stored in the code stream (the item offset), and how the item is encoded in the code stream (the type). The code pointer is used to relocate items stored as relative addresses.

Because procedure entry points may change during recompilation, relocation entries for calls use the long format so that the translated offset cannot exceed the field width. Consequently, the long format provides a convenient location for the "d" bit, which indicates whether an entry corresponds to a call instruction. Use of the long format does not significantly increase the table size because calls account for a minority of relocation entries. Because short entries cannot describe calls, they need not be checked.

Three of the code object's type bits are used to encode the status of a code object. The first bit, set by the recompiler, indicates whether the code object has been recompiled to a new one and is thus *obsolete*. Since an obsolete code object will not be copied during collection, its relocation table is no longer needed. Therefore, its reloc field is used to point to the translation table instead. The list of original/new offset pairs is sorted by original offset to enable fast binary searching during relocation.

The second bit, denoted by "n" in the figure, indicates whether the code object is *new*. This bit, set by the recompiler and cleared by the collector, is used to prevent a new code object from being recompiled before the associated obsolete one has been eliminated. The third bit, denoted by "b" in the figure, serves a similar purpose. It indicates whether an *old* code object is *busy* in that it is either being or has been recompiled. This bit is used to prevent multiple recompilation of the same code object. The recompiler sets this bit at the beginning of recompilation. Because the recompiler may trigger a garbage collection before it creates a new code object and marks the original one obsolete, this bit must be preserved by the collector. It is possible to recompile a code object multiple times; however, each subsequent recompilation must occur after the code object has been recompiled and collected. (It would be possible to relax this restriction.)

In order to support multiple return values, first-class continuations, and garbage collection efficiently, four words of data are placed in the instruction stream immediately before the single-value return point from each non-tail call [17, 1, 5]. The live mask is a bit vector describing which frame locations contain live data. The code pointer is used to find the code object associated with a given return address. The frame size is used during garbage collection, continuation invocation, and debugging to walk down a stack one frame at a time.

3.3 Collection

Only a few modifications are necessary for the garbage collector to support dynamic recompilation. The primary

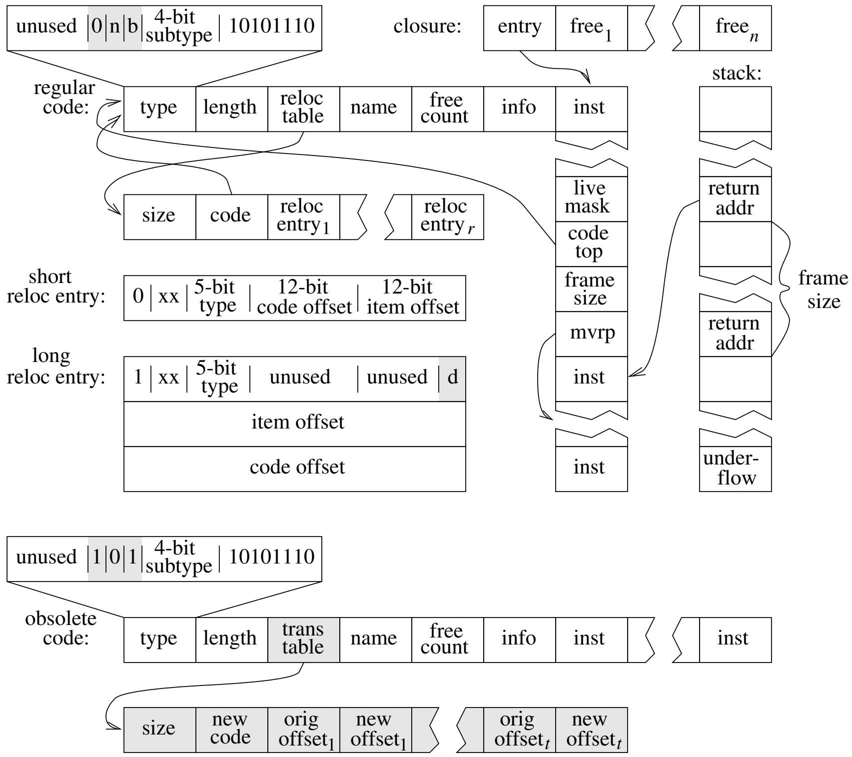


Figure 7: Representation of closures, code objects, and stacks with the infrastructure for dynamic recompilation highlighted.

change involves how a code object is copied. If the obsolete bit is clear, the code object and associated relocation table are copied as usual, but the new bit of the copied code object's type field is cleared if set.

If the obsolete bit is set, the code object is not copied at all. Instead, the pointer to the new code object is relocated and stored as the forwarding address for the obsolete code object so that all other pointers to the obsolete code object will be forwarded to the new one. Moreover, if the obsolete code object is instrumented for profiling, the collector propagates the counts so that they remain accurate when the new code object increments a possibly different subset of the counts. The list of edge/block pairs used for propagation is found in the info structure. Since the propagation occurs during collection, some of the objects in the list may be forwarded, so the collector must check for pointers to forwarded objects as it propagates the counts.

The translation table is used to relocate call instructions and return addresses. Call instructions are found only in code objects, so they are handled when code objects are swept. The "d" bit of long-format relocation entries identifies the candidates for translation. Return addresses are found only in stacks, so they are handled when continuations are swept.

Since our collector is generational, we must address the problem of potential cross-generational pointers from obsolete to new code objects. Our segmented heap model allows us to allocate new objects in older generations when necessary [14]; thus, we always allocate new code objects in the same generation as the corresponding obsolete code objects. This allows us to keep the code space free of cross-generational pointers from older to newer objects.

3.4 Block Reordering

To demonstrate the dynamic recompilation infrastructure, we have applied it to the problem of basic-block reordering. We use a variant of Pettis and Hansen's basic-block reordering algorithm to reduce the number of mispredicted branches and instruction cache misses [24]. We also use the edge-count profile data to decrease profiling overhead by re-running the maximal spanning tree algorithm to improve counter placement.

The block reordering algorithm proceeds in two steps. First, blocks are combined into chains according to the most frequently executed edges to reduce the number of instruction cache misses. Second, the chains are ordered to reduce the number of mispredicted branches.

Initially, every block comprises a chain of one block, itself. Using a list of edges sorted by decreasing weight, dis-

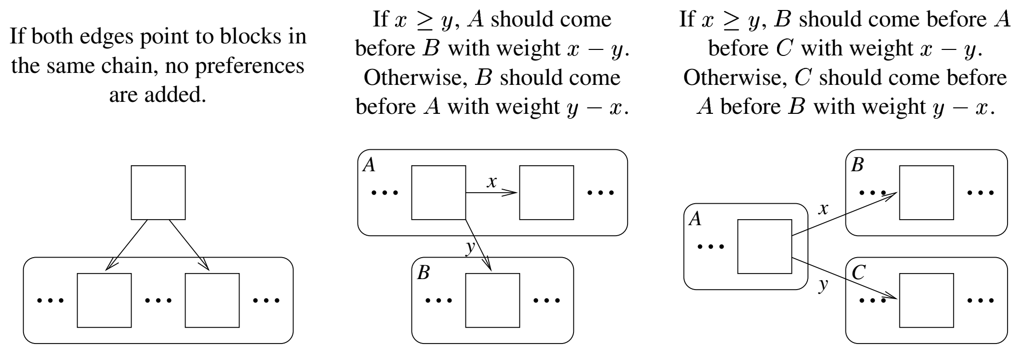


Figure 8: Adding branch prediction preferences for architectures that predict backward conditional branches taken and forward conditional branches not taken.

tinct chains A and B are combined when an edge's source block is the tail of A and its sink is the head of B. When all the edges have been processed, a set of chains is left.

The algorithm places ordering preferences on the chains based on the conditional branches emitted from the blocks within the chains and the target architecture's branch prediction strategy. Blocks with two outgoing edges always generate a conditional branch for one of the edges, and they generate an unconditional branch when the other edge does not point to the next block in the chain. Figure 8 illustrates how the various conditional branch possibilities generate preferences for a common prediction strategy. The preferences are implemented using a weighted directed graph with nodes representing chains and edges representing the "should come before" relation.

As each block with two outgoing edges is processed, its preferences (if any) are added to the weights of the graph. Suppose there is an edge of weight x from chain A to B and an edge of weight y from chain B to A, and $x > y$. The second edge is removed, and the first edge's weight becomes $x - y$, so that there is only one positive-weighted edge between any two nodes. A depth-first search then topologically sorts the chains, omitting edges that cause cycles. The machine code for the chains is placed in a new code object, and the old code object is marked obsolete and has its reloc field changed to point to the translation table.

4 Performance

To assess both compile-time and run-time performance of the dynamic recompiler and profiler, we used a set of benchmarks comprised of *Chez Scheme* 5.0g recompiling itself, Soft Scheme [29] checking its pattern matcher, Digital Design Derivation System 1.0 [4] deriving a Scheme CPU [6], Similix 5.0 partially evaluating itself [3], and the Gambit-C 2.3.1 benchmark suite. All measurements were taken on a DEC Alpha 3000/600 running Digital UNIX V4.0A and are reported in detail elsewhere [7].

Initial instrumentation has an average run-time overhead of 50% and an average compile-time overhead of 11%. Without block reordering, optimal count placement reduces the average run-time overhead to 37%, and the average recompile time is only 15% of the base compile time. With block reordering, the average run-time overhead drops to 27%, while the average recompile time increases very slightly to 16%.

The reduction of profiling overhead from 50% to 37% is quite respectable and shows that optimal counter placement can be an effective tool for use in long program runs intended to gather precise overall profiling information. Moreover, profile-driven run-time optimizations such as block reordering further reduce profiling overhead. Nonetheless, optimized code will run faster with profiling disabled. This suggests a strategy for dynamic optimization in which a program is dynamically optimized and profiling instrumentation removed after an initial portion of a program run during which profiling information is gathered. In addition, profiling can be enabled for particular procedures at various times during a run to provide a sort of statistical sampling, although this would likely have to be done at the programmer's direction.

To assess the effectiveness of the block-reordering algorithm, we measured the number of mispredicted conditional branches and the number of unconditional branches. The Alpha architecture encourages hardware and compiler implementors to predict backward conditional branches taken and forward conditional branches not taken [28]. Current Alpha implementations use this static model as a starting point for dynamic branch prediction. We computed the mispredicted branch percentage using the static model as a metric for determining the effectiveness of the block-reordering algorithm. Like Pettis and Hansen's algorithm, ours achieves near-optimal misprediction rates, significantly reduces the number of unconditional branches, and provides a modest 6% reduction in run time. Our algorithm is also fast, for the average recompile time is just 12% of the base compile time. Because the Alpha performs dynamic branch prediction, the 6% reduction in run time is likely due mostly to a reduction in instruction cache misses.

5 Related Work

Our edge-count profiling algorithm is based on one described by Ball and Larus [2], who applied their algorithm in a traditional static compilation environment. We have extended their algorithm to support first-class continuations and reinstrumentation of active procedures. We have also identified an algorithm for determining optimal counter placement that is faster and eliminates the need for a separate depth-first search for count propagation.

We have used Pettis and Hansen's intraprocedural block-reordering strategy with only minor modifications to apply it in the context of dynamic recompilation. Samples [26] explores a similar intraprocedural algorithm that reduces instruction cache miss rates by up to 50%. Pettis and Hansen [24] also describe an interprocedural algorithm that places procedures in memory such that those that execute close together in time will also be close together in memory. Since determining the optimal ordering is NP-complete, they use a greedy "closest is best" strategy.

Several recent research projects have focused on lightweight run-time code generation [8, 15, 22, 20, 25], with code generation costs on the order of five to 100 instructions executed for each instruction generated. Although the overhead inherent in our model is greater, the potential benefits are greater as well.

Little attention has previously been paid to profile-driven dynamic optimizations, with the notable exception of work on Self [9, 18]. This work uses a specialized form of profiling to guide generation of special-purpose code to avoid generic dispatch overhead. Our infrastructure incorporates a more general profiling strategy that is not targeted to any particular optimization technique, although we have so far applied it only to the limited problem of block reordering.

Keppel has investigated the application of run-time code generation to value-specific optimizations, in which code is special-cased to particular input values [19]. Although we have so far focused on profile-driven optimizations, we believe that our infrastructure is well suited to value-specific optimizations as well.

6 Conclusions

We have described an efficient infrastructure for dynamic recompilation that incorporates a low-overhead edge-count profiling strategy supporting first-class continuations and reinstrumentation of active procedures. We have shown how the profile data can be used to improve performance by reordering basic blocks and optimizing counter placement. In addition, we have explained how the profile data can be associated with the original source to provide graphical feedback to the programmer.

The mechanisms described in this paper have all been implemented and incorporated into *Chez Scheme*. The recompiler can profile and regenerate all code in the system, including itself, as it runs. Performance results show that the average run-time profiling overhead is initially 50% and decreases significantly with recompilation. The average compile-time profiling overhead is 11%, and the average recompile time relative to the base compile time is 12% without profiling instrumentation and 15–16% with profiling instrumentation. The block-reordering algorithm is fast and effective at reducing the number of mispredicted branches and unconditional branches.

The techniques and algorithms are directly applicable to garbage-collected languages such as Java, ML, Scheme, and Smalltalk, and can be adapted to other languages as described in Section 1.

Dynamic recompilation need not be limited to low-level optimizations such as block reordering. A promising area of future work involves associating the profile data with earlier passes of the compiler so that higher-level optimizations can take advantage of the information. For example, register allocation, flow analysis, and inlining could benefit from profile data. An unoptimized active code segment with no analogue in the optimized version of a program, such as code for an active procedure that has been inlined at its call sites, can be retained by the system until no longer needed. This will, in fact, occur naturally in garbage-collected systems.

Another area of future work involves a generalization of edge-count profiling to measure other dynamic program characteristics such as the number of procedure calls, variable references, and so forth. For example, profile data can be used to measure how many times a procedure is called versus how many times it is created, and this ratio could be used to guide lambda lifting [12]. Combined with an estimate of the cost of generating code for the procedure, this ratio could also help determine when run-time code generation [22] would be profitable. Our system can also be extended to recompile (specialize) a closure based on the run-time values of its free variables.

References

[1] J. Michael Ashley and R. Kent Dybvig. An efficient implementation of multiple return values in Scheme. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 140–149, June 1994.

[2] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 59–70, January 1992.

[3] Anders Bondorf. *Similix Manual, System Version 5.0*. DIKU, University of Copenhagen, Denmark, 1993.

[4] Bhaskar Bose. DDD—A transformation system for Digital Design Derivation. Technical Report 331, Indiana University, Computer Science Department, May 1991.

[5] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 99–107, May 1996.

[6] Robert G. Burger. The Scheme Machine. Technical Report 413, Indiana University, Computer Science Department, August 1994.

[7] Robert G. Burger. *Efficient Compilation and Profile-Driven Dynamic Recompilation in Scheme*. PhD thesis, Indiana University, March 1997.

[8] Chaig Chambers, Susan J. Eggers, Joel Auslander, Matthai Philipose, Markus Mock, and Przemyslaw Pardyak. Automatic dynamic compilation support for event dispatching in extensible systems. In *WCSSS'96 Workshop on Compiler Support for System Software*, February 1996.

[9] Craig Chambers. *The Design and Implementation of the* SELF *Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, April 1992.

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

[11] William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July–September 1991.

[12] William D. Clinger and Lars Thomas Hansen. Lambda, the ultimate label, or a simple optimizing compiler for Scheme. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 128–139, 1994.

[13] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk–80 system. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages*, pages 297–302, January 1984.

[14] R. Kent Dybvig, David Eby, and Carl Bruggeman. Don't stop the BIBOP: Flexible and efficient storage management for dynamically typed languages. Technical Report 400, Indiana University, Computer Science Department, March 1994.

[15] Dawson R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, May 1996.

[16] Dawson R. Engler, Deborah Wallach, and M. Frans Kaashoek. Efficient, safe, application-specific message processing. Technical Memorandum MIT/LCS/TM533, MIT Laboratory for Computer Science, March 1995.

[17] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 66–77, June 1990.

[18] Urs Hölze and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 326–335, June 1994.

[19] David Keppel, Susan J. Eggers, and Robert R. Henry. Evaluating runtime-compiled value-specific optimizations. Technical Report 93-11-02, Department of Computer Science and Engineering, University of Washington, November 1993.

[20] Mark Leone. *A Principled and Practical Approach to Run-Time Code Generation*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997. In preparation.

[21] Mark Leone and R. Kent Dybvig. Dynamo: A staged compiler architecture for dynamic program optimization. Technical Report 490, Indiana University, Computer Science Department, September 1997.

[22] Mark Leone and Peter Lee. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 137–148, May 1996.

[23] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Department of Computer Science, Columbia University, 1992.

[24] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[25] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. `tcc`: A template-based compiler for 'C. In *WCSSS'96 Workshop on Compiler Support for System Software*, pages 1–7, February 1996.

[26] A. Dain Samples. Profile-driven compilation. Technical Report 627, University of California, Berkeley, 1991.

[27] Robert Sedgewick. *Algorithms*, chapter 31. Addison-Wesley Publishing Company, second edition, 1988.

[28] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[29] Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 250–262, 1994.