An Infrastructure for Adaptive Dynamic Optimization

Derek Bruening, Timothy Garnett, and Saman Amarasinghe
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{iye,timothyg,saman}@lcs.mit.edu

Abstract

Dynamic optimization is emerging as a promising approach to overcome many of the obstacles of traditional static compilation. But while there are a number of compiler infrastructures for developing static optimizations, there are very few for developing dynamic optimizations. We present a framework for implementing dynamic analyses and optimizations. We provide an interface for building external modules, or clients*, for the DynamoRIO dynamic code modification system. This interface abstracts away many low-level details of the DynamoRIO runtime system while exposing a simple and powerful, yet efficient and lightweight, API. This is achieved by restricting optimization units to linear streams of code and using adaptive levels of detail for representing instructions. The interface is not restricted to optimization and can be used for instrumentation, profiling, dynamic translation, etc.*

To demonstrate the usefulness and effectiveness of our framework, we implemented several optimizations. These improve the performance of some applications by as much as 40% relative to native execution. The average speedup relative to base DynamoRIO performance is 12%.

1 Introduction

The power and reach of static analysis is diminishing for modern software, which heavily utilizes dynamic class loading, shared libraries, and runtime binding. Not only is it difficult or impossible for a static compiler to analyze the whole program, but static optimization is limited by the accuracy of its predictions of runtime program behavior. Using profile information improves the predictions but still falls short for programs whose behavior changes dynamically. Additionally, many software vendors are hesitant to ship binaries that are compiled with high levels of static optimization because they are hard to debug.

Shifting optimizations to runtime solves these problems. Dynamic optimization allows the user to improve the performance of binaries without relying on how they were compiled. Furthermore, several types of optimizations are best suited to a dynamic optimization framework. These include adaptive, architecture-specific, and inter-module optimizations.

Adaptive optimizations require instant responses to changes in program behavior. When performed statically, a single profiling run is taken to be representative of the program's behavior. Within a dynamic optimization system, ongoing profiling identifies which code is currently hot, allowing optimizations to focus only where they will be most effective.

Architecture-specific code transformations may be done statically if the resulting executable is only targeting a single processor, or by using dynamic dispatch to select among several transformations prepared for different processors. The first option unduly restricts the executable while the second bloats the executable size. Performing the optimization dynamically solves the problem by allowing the executable to remain generic and specialize itself to the processor it happens to be running on.

Inter-module optimizations cannot be done statically in the presence of shared and dynamically loaded libraries. But all code is available to a dynamic optimizer, presenting optimizations with a view of the code that cuts across the static units used by the compiler for optimization.

Dynamic optimizations have one significant disadvantage versus static optimizations. The overhead of performing the optimization must be amortized before any improvement is seen. This limits the scope of optimizations that can be done online, and makes the efficiency of the optimization infrastructure extremely critical. For this reason, while there are numerous flexible and general compiler infrastructures for developing static optimizations, there are very few for the development of dynamic optimizations.

Another important contrast with static compilation is transparency. Unlike a static compiler optimization, a dynamic optimization cannot use the same memory allocation routines or input/output buffering as the application, because the optimization's operations are interleaved with those of the application.

The main contribution of this paper is a framework for implementing dynamic analyses and optimizations. The framework is based on the DynamoRIO dynamic code modification system. We export an interface for building external modules, or *clients*, for DynamoRIO. With this API, custom runtime code transformations are simple to develop. Efficiency is achieved through two key principles: restricting optimization units to linear streams of code and using adaptive levels of detail for representing instructions. Our interface provides direct support for customizable traces and adaptive optimization of traces, while maintaining transparency with respect to the application. It is general enough to be used for non-optimization purposes, including instrumentation, profiling, and security [23]. The system is available to the public in binary form [16].

The paper is organized as follows. We first present in Section 2 a description of the DynamoRIO system. DynamoRIO operates on unmodified native binaries and requires no special hardware or operating system support. It is implemented for both IA-32 Windows [5] and Linux, and is capable of running large desktop applications. Section 3 describes our optimization interface in detail, and Section 4 gives some examples of dynamic optimizations written using the interface. We give experimental results in Section 5. We discuss two other dynamic code modification interfaces along with other related work in Section 6.

2 DynamoRIO

Our optimization infrastructure is built on a dynamic optimizer called DynamoRIO. DynamoRIO is the IA-32 version [5] of Dynamo [4]. It is implemented for both IA-32 Windows and Linux, and is capable of running large desktop applications.

The goal of DynamoRIO is to observe and potentially manipulate every single application instruction prior to its execution. The simplest way to do this is with an interpretation engine. However, interpretation via emulation is slow, especially on an architecture like IA-32 with a complex instruction set, as shown in Table 1. DynamoRIO uses a typical trick to avoid emulation overhead: it caches translations of frequently executed code so they can be directly executed in the future.

DynamoRIO copies *basic blocks* (sequences of instructions ending with a single control transfer instruction) into a code cache and executes them natively. At the end of each block the application's machine state must be saved and control returned to DynamoRIO (a *context switch*) to copy the next basic block. If a target basic block is already present in the code cache, and is targeted via a direct branch, DynamoRIO *links* the two blocks together with a direct jump. This avoids the cost of a subsequent context switch.

Indirect branches cannot be linked in the same way because their targets may vary. To maintain transparency, original program addresses must be used wherever the application stores indirect branch targets (for example, return addresses for function calls). These addresses must be translated into their corresponding code cache addresses in order to jump to the target code. This translation is performed as a fast hashtable lookup.

To improve the efficiency of indirect branches, and to achieve better code layout, basic blocks that are frequently executed in sequence are stitched together into a unit called a *trace*. When connecting beyond a basic block that ends in an indirect branch, a check is inserted to ensure that the actual target of the branch will keep execution on the trace. This check is much faster than the hashtable lookup, but if the check fails the full lookup must be performed. The superior code layout of traces goes a long way toward amortizing the overhead of creating them and often speeds up the program [4, 32].

A flow chart showing the operation of DynamoRIO is presented in Figure 1. The figure concentrates on the flow of control in and out of the code cache, which is the bottom portion of the figure. The copied application code looks just like the original code with the exception of its control transfer instructions, which are shown with arrows in the figure.

Table 1 shows the typical performance improvement of each enhancement to the basic interpreter design. Caching is a dramatic performance improvement, and adding direct links is nearly as dramatic. The final steps of adding a fast in-cache lookup for indirect branches and building traces improve the performance significantly as well.

The Windows operating system directly invokes application code or changes the program counter for callbacks, exceptions, asynchronous procedure calls, `setjmp`, and the `SetThreadContext` API routine. These types of control flow are intercepted in order to ensure that all application code is executed under DynamoRIO [5]. Signals on Linux must be similarly intercepted.

DynamoRIO maintains thread-private code caches, each separated into a basic block cache and a trace cache. It was found that, in most multi-threaded applications, very little

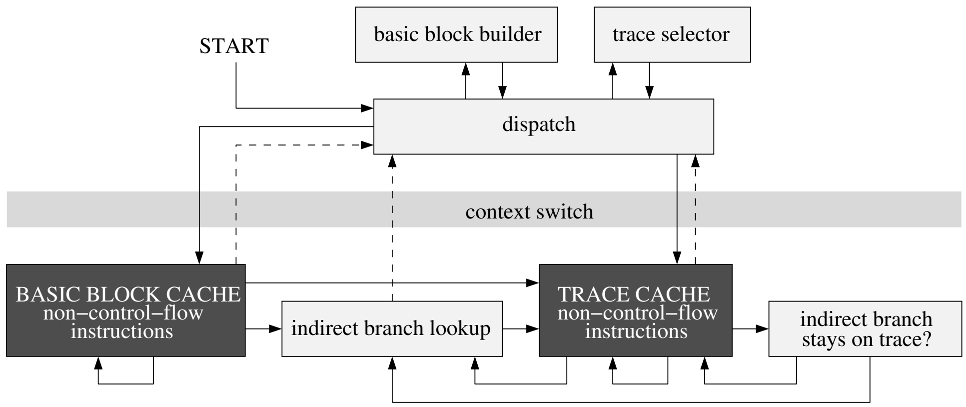

Figure 1. Flow chart of the DynamoRIO system infrastructure. Dark shading indicates application code. Note that the context switch is simply between the code cache and DynamoRIO; application code and DynamoRIO code all runs in the same process and address space. Dotted lines indicate the performance-critical cases where control must leave the code cache and return to DynamoRIO.

System Type	Normalized Execution Time	
	crafty	vpr
Emulation	˜ 300.0	˜ 300.0
+ Basic block cache	26.1	26.0
+ Link direct branches	5.1	3.0
+ Link indirect branches	2.0	1.2
+ Traces	1.7	1.1

Table 1. Performance achieved when various features are added to an interpreter, measured on two of the SPEC2000 benchmarks [34], crafty and vpr. Pure emulation results in a slowdown factor of several hundred. Successively adding caching, linking, and traces brings the performance down dramatically.

code was shared between threads, so the cost of duplicating the small amount that was shared for each thread was far outweighed by the savings of not having to synchronize changes in the cache with all the running threads [5]. Additionally, thread-private code caches enable thread-specific optimizations.

In this paper we will use the term *fragment* to mean either a basic block or a trace in the code cache.

3 Client Interface

DynamoRIO exports a rich Application Programming Interface (API) to the user for building a *DynamoRIO client* [16]. A DynamoRIO client is coupled with DynamoRIO in order to jointly operate on an input program. In addition to using the API, the client supplies specific hook functions which are called by DynamoRIO.

The client interface supports the development of custom program transformations. It hides low-level details of DynamoRIO such as cache management, trace building, and context switching, focusing only on how the application code is modified when it is placed into the code cache. The interface also has explicit support for maintaining transparency with respect to the application.

To keep overheads down, the interface restricts all instruction sequences to have linear control flow and uses adaptive levels of detail in representing instructions.

3.1 Instruction Representation

DynamoRIO operates on two kinds of code sequences: basic blocks and traces. Both have linear control flow, with a single entrance and potentially multiple exits, but no internal join points (all transfers of control that originate inside must exit). Optimizations make use of the linear control flow present in traces. The single-entry multiple-exit format simplifies analysis algorithms, which reduces optimization overheads.

Since DynamoRIO deals only with linear streams of code, it represents a basic block or trace as a linked list of instructions called an `InstrList`. A single instruction, or a group of bundled un-decoded instructions, is represented in the list by an `Instr` data structure.

In any system designed to manipulate machine instructions, the instruction representation is key. Ease and flexibility of use have been traditional concerns for compiler writers. Dynamic frameworks add an additional concern for performance. Since the decoding and encoding of ma-

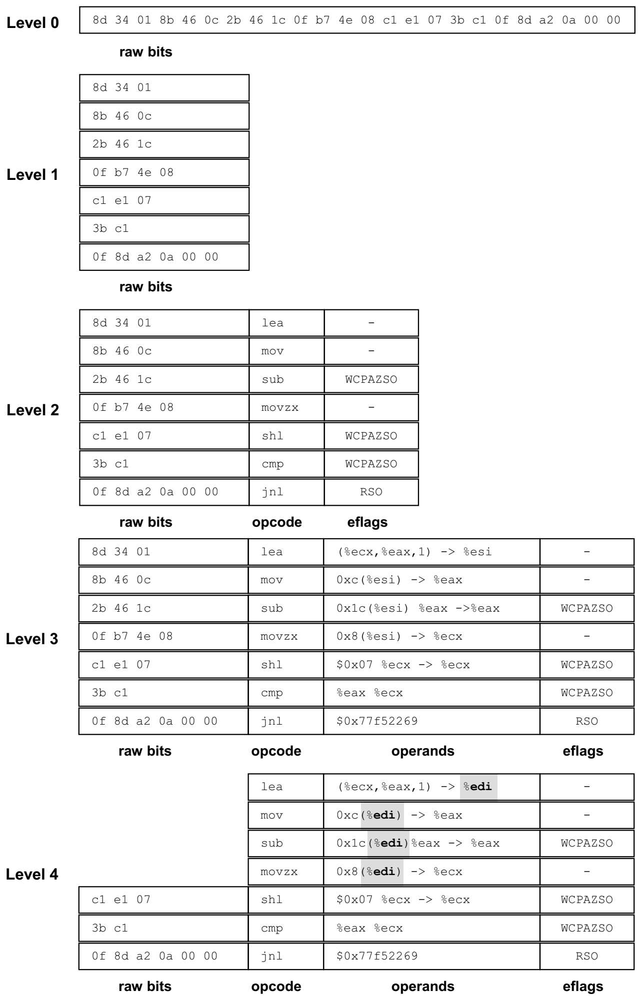

Figure 2. Example sequence of instructions at each of the five levels of representation.

chine instructions is performed at runtime under a dynamic framework, many dynamic systems resort to low-level, often difficult-to-use, representations in the interest of efficiency. The problem is especially pronounced in CISC instruction sets such as IA-32, where instructions vary greatly in length and complexity, and require significant overhead to fully decode. DynamoRIO addresses this issue by using an adaptive level-of-detail instruction representation with five different levels, which are illustrated in Figure 2:

Level 0 – At its lowest level of detail, an `Instr` holds the raw instruction bytes of a series of instructions and only records the final instruction boundary.

Level 1 – A Level 0 `Instr` is split such that an `Instr` is created for each machine instruction. Each `Instr` still holds only the un-decoded raw bits for the instruction it represents.

Level 2 – The instruction is decoded enough to determine its opcode and effect on the `eflags` register (which contains condition codes and status flags) for quick determination of whether the `eflags` need to be saved or restored around inserted instructions. Many IA-32 instructions modify the `eflags` register, making them an important factor to consider in any code transformation.

Level 3 – A fully-decoded instruction whose raw bits are valid. `Instr` has fields for opcode, prefixes, and `eflags` effects, plus two dynamically-allocated arrays of operands, one for sources and one for destinations. These arrays are dynamically-allocated because IA-32 instructions may contain between zero and eight sources and destinations. This level combines quick encoding (simply copy the raw bits) with high-level information.

Level 4 – A fully-decoded instruction that has been modified (or newly created) and does not have a valid copy of raw instruction bits. This is the only level at which instructions must be encoded (or re-encoded) to obtain the machine representation.

The initial level of an `Instr` is determined by which API routine is used to build the instruction. Later operations can change the level, either implicitly or explicitly. For example, modifying an operand will cause the raw bytes to become invalid, moving an instruction up to Level 4. This automatic adjustment makes it easy for an optimization to use the lowest cost representation possible. Switching incrementally between levels costs no more than a single switch spanning multiple levels.

To support the multiple `Instr` levels, multiple decoding strategies are employed. The lowest level simply finds instruction boundaries (even this is non-trivial for IA-32). Although the instruction boundaries need to be determined for both Level 0 and Level 1, the boundary information may not be needed later. Level 0 avoids storing that information, and further simplifies encoding by allowing a single memory copy rather than an iteration over multiple boundaries. Level 2 decodes just enough to determine the opcode and the instruction's effect on the `eflags`. Finally, for Level 3 and Level 4, a full decode determines all of the operands.

To encode an `Instr`, first the raw bit pointer is checked. If it is valid, the instruction is encoded by simply copying the raw bits. If the raw bits are invalid (Level 4), the instruction must be fully encoded from its operands. Encoding an IA-32 instruction is costly, as many instructions have special forms when the operands have certain values. The encoder must walk through every operand and find an instruction template that matches. Avoiding this by copying raw bits whenever possible is important.

Level	Time (μs)	Memory (bytes)
0	2.12	64.00
1	12.42	628.95
2	13.01	629.07
3	19.10	791.55
4	61.79	791.55

Table 2. Average time and memory used to decode and then encode the basic blocks of the SPEC2000 benchmarks [34].

As an example of the use of various levels of instruction representation, consider the creation of a basic block fragment. All that DynamoRIO needs to know about is control flow instruction terminating the block. Accordingly, the `InstrList` for a basic block might contain only two `Instr`s. The first `Instr` (at Level 0) simply points to the raw bits of an arbitrarily long sequence of non-control flow instructions, while the second `Instr` (at Level 3) holds the fully decoded state for the block-ending control flow instruction, ready for modification. When performing optimizations, DynamoRIO fully decodes all instructions in a trace's `InstrList`, but keeps their raw bit pointers valid (Level 3). All unmodified instructions can be quickly encoded by simply copying the bits.

For a quantitative evaluation of the different levels of instruction representation, we measured the time and memory used to decode and then encode basic blocks at each level of representation. Table 2 shows the average time and memory across all blocks for the SPEC2000 benchmarks [34].

3.2 DynamoRIO API

DynamoRIO exports a rich set of functions and data structures to manipulate IA-32 instructions, using the data structures discussed in Section 3.1. Instruction generation is simplified through a set of macros. A macro is provided for every IA-32 instruction. The macro takes as arguments only those operands that are explicit and automatically fills in the implicit operands (many IA-32 instructions have implicit operands). The IA-32 instruction set abstraction level can also be bypassed by specifying an opcode and complete list of operands.

To support transparency, DynamoRIO provides routines for input/output and memory allocation (global and thread-private) that do not interfere with the application. A client that instead uses the same buffers or memory allocation routines as the application has a good chance of affecting program correctness.

DynamoRIO uses special thread-local slots to spill registers. It exports an API routine that will save a register to one of these slots. Additionally, it provides a generic thread-local storage field for use by clients. DynamoRIO also provides a field in the `Instr` data structure that can be used by the client for annotations while it is processing instructions.

Frequently, an optimization will make an assumption in order to optimize a sequence of code. If the assumption is violated, some clean-up action is required. To maintain the linearity of traces, DynamoRIO provides a mechanism for implementing this kind of clean-up code in the form of *custom exit stubs*. Each exit from a trace or basic block has its own stub. When it is not linked to another trace, control goes to the stub, which records where the trace was exited and then performs the context switch back to DynamoRIO. The client can specify a list of instructions to be pre-pended to the stub corresponding to any exit from a trace or basic block fragment, and can specify that the exit should go through the stub even when linked. The body of the fragment is then optimized for the assumption. Conditional branches direct control flow to the custom stub if the assumption is violated. Without this direct support, a client would be forced to add branches targeting the middle of the trace, destroying the linear control flow which may be expected by other optimizations.

DynamoRIO also provides routines that identify features of the underlying processor, making it easy to perform architecture-specific optimizations.

3.3 DynamoRIO Client

A DynamoRIO client can implement several functions shown in Table 3 that will be called by DynamoRIO at appropriate moments. The two most important client-supplied hooks are those for basic block and trace creation, `dynamorio_basic_block` and `dynamorio_trace`. Through these hooks the client has the ability to inspect and transform any piece of code that is emitted into the code cache.

DynamoRIO calls `dynamorio_basic_block` each time a block is created. The basic block is passed as a pointer to an `InstrList`. This routine is used by clients that need to operate on every piece of application code.

DynamoRIO calls `dynamorio_trace` each time a trace is created, just before the trace is placed in the trace cache. The trace is passed as an `InstrList` that has already been completely processed by DynamoRIO. The client sees exactly the code that will execute in the code cache (with the exception of the exit stubs). Most client optimizations only operate on traces, restricting themselves to hot code.

`dynamorio_fragment_deleted` is called each time a fragment is deleted from the block or trace cache. Such information is needed if the client maintains its own data structures about emitted fragment code that must be kept consistent across fragment deletions.

`dynamorio_end_trace` is described in Section 3.5.

Client Routine	Description
`void dynamorio_init()`	Client initialization
`void dynamorio_exit()`	Client finalization
`void dynamorio_thread_init(void *context)`	Client per-thread initialization
`void dynamorio_thread_exit(void *context)`	Client per-thread finalization
`void dynamorio_basic_block(void *context, app_pc tag, InstrList *bb)`	Client processing of basic block
`void dynamorio_trace(void *context, app_pc tag, InstrList *trace)`	Client processing of trace
`void dynamorio_fragment_deleted(void *context, app_pc tag)`	Notifies client when a fragment is deleted from the code cache
`int dynamorio_end_trace(void *context, app_pc trace_tag, app_pc next_tag)`	Asks client whether to end the current trace

Table 3. Client routines imported by DynamoRIO. The client is not expected to inspect or modify the `context` parameter, which is an opaque pointer to the current thread context. The `tag` parameters serve to uniquely identify fragments by their original application origin.

3.4 Extensions for Adaptive Optimization

Two additional routines are exported by DynamoRIO to support adaptive optimization:

```
InstrList* dr_decode_fragment(
  void *context, app_pc tag);
bool dr_replace_fragment(void *context,
  app_pc tag, InstrList *il);
```

Clients may wish to re-optimize code after it is placed in the code cache. To do this, clients need to re-create the `InstrList` for a trace from the cache, modify it, and then replace the old version with the new. For example, consider a client that inserts profiling code into selected traces. Once a threshold is reached, the profiling code calls `dr_decode_fragment` and then rewrites the trace by modifying the `InstrList`. Once finished, `dr_replace_fragment` is called to install the new version of the trace.

DynamoRIO is able to perform this replacement while execution is still inside the old fragment, allowing a trace to generate a new version of itself. This is accomplished by delaying the removal of the old fragment until a safe point. All links targeting and originating from the old fragment are immediately modified to use the new fragment. This means that the current thread will continue to execute in the old fragment only until the next branch. Since there are no loops except in explicit links, the time spent in the old fragment is minimal, and all future executions use the new fragment.

Enabling optimizations to be performed in a separate thread requires surprisingly few additions to the adaptive optimization interface. To make fragment replacement possible from a separate thread, we simply prevent the optimizing thread and the application thread from both being in DynamoRIO code at the same time. If the application thread remains in the code cache until after the replacement is complete, no synchronization cost is incurred. We plan to investigate using a concurrent thread for "sideline optimization" using this low-overhead trace replacement.

3.5 Extensions for Custom Traces

A client can direct the building of traces through a combination of the client hook `dynamorio_end_trace` and this API routine:

```
void dr_mark_trace_head(void *context,
  app_pc tag);
```

The basic trace building mechanism is similar to the original Dynamo [4] traces. Certain basic blocks are considered *trace heads*. A counter associated with each trace head is incremented upon each execution of that basic block. Once the counter exceeds a threshold, DynamoRIO enters trace generation mode. Each subsequent basic block executed is added to the trace, until a termination point is reached.

Dynamo only considered targets of backward branches and exits of existing traces to be trace heads. Our interface allows a client to choose its own traces heads, marking them with `dr_mark_trace_head`. When DynamoRIO is in trace generation mode, it calls the client's `dynamorio_end_trace` routine before adding a basic block to the current trace. The client can direct DynamoRIO to either end the trace, not end the trace, or use its default test (which stops at a backward branch or upon reaching an existing trace) for whether to end the trace. For an example of using this interface, see Section 4.4.

4 Examples

We present four sample optimizations implemented with the DynamoRIO client interface. Section 5 shows the performance impact of these optimizations.

4.1 Redundant Load Removal

We took a traditional compiler optimization, redundant load removal, and implemented it dynamically. Because there are so few registers in IA-32, local variables are frequently loaded from and stored back to the stack. If a variable's value is already in a register, a subsequent load can be removed. The compiler should be able to eliminate redundant loads within basic blocks, but we found that `gcc` at its highest optimization level still emits a number of redundant loads within blocks. It also produces redundant loads across basic block boundaries, which are a little more difficult for the compiler to identify. This optimization shows that even code compiled at high optimization levels stands to benefit from dynamic application of traditional optimizations.

4.2 Strength Reduction

On the Pentium 4 the `inc` instruction is slower than `add 1` (and `dec` is slower than `sub 1`). The opposite is true on the Pentium 3, however. As the code in Figure 3 shows, all a DynamoRIO client needs to do to perform this strength-reduction optimization is walk the instructions in each basic block and look for `inc` instructions. A simple analysis needs to be done to determine if the `eflags` differences between `inc` and `add` are acceptable for this block. If so, the `inc` is replaced by `add 1`. In a similar manner, `dec` is replaced with `sub 1`.

This is a perfect example of an architecture-specific optimization that is best performed dynamically, tailoring the program to the actual processor it is running on. It would be awkward to try this at load time. The variable-length IA-32 instruction set makes it difficult or impossible to analyze binaries statically, because the internal module boundaries are not known. Furthermore, a loader would have to rewrite all code in all shared libraries, regardless of how little of that code is actually run, and would need to specially handle dynamically determined libraries (loaded using `dlopen` or `LoadLibrary`).

```
EXPORT void dynamorio_init() {
  enable = (proc_get_family() == FAMILY_PENTIUM_IV);
  num_examined = 0;
  num_converted = 0; }

EXPORT void dynamorio_exit() {
  if (enable) {
    dr_printf("converted %d out of %d\n",
      num_converted, num_examined); }
  else { dr_printf("kept original inc/dec\n"); } }

EXPORT void dynamorio_trace
  (void *context, app_pc tag, InstrList *trace) {
  Instr *instr, *next_instr;
  int opcode;
  if (!enable) return;
  for (instr = instrlist_first(bb); instr != NULL;
       instr = next_instr) {
    next_instr = instr_get_next(instr);
    opcode = instr_get_opcode(instr);
    if (opcode == OP_inc || opcode == OP_dec ) {
      num_examined++;
      if (inc2add(context, instr, trace))
        num_converted++; } } }

static bool inc2add
  (void *context, Instr *instr, InstrList *trace) {
  Instr *in;
  uint eflags;
  int opcode = instr_get_opcode(instr);
  bool ok_to_replace = false;
  /* add writes CF, inc does not, check ok! */
  for (in=instr; in != NULL; in=instr_get_next(in)) {
    eflags = instr_get_eflags(in);
    if ((eflags & EFLAGS_READ_CF) != 0) return false;
    /* if writes but doesn't read, we can replace */
    if ((eflags & EFLAGS_WRITE_CF) != 0) {
      ok_to_replace = true;
      break; }
    /* simplification: stop at first exit */
    if (instr_is_exit_cti(in)) return false; }
  if (!ok_to_replace) return false;
  if (opcode == OP_inc )
    in = INSTR_CREATE_add(context,
      instr_get_dst(instr,0),OPND_CREATE_INT8(1));
  else
    in = INSTR_CREATE_sub(context,
      instr_get_dst(instr,0),OPND_CREATE_INT8(1));
  instr_set_prefixes(in, instr_get_prefixes(instr));
  instrlist_replace(trace, instr, in);
  instr_destroy(context, instr);
  return true; }
```

Figure 3. Code for a client implementing an `inc` to `add 1` strength reduction optimization.

4.3 Indirect Branch Dispatch

As an example of adaptive optimization, we perform value profiling of indirect branch targets. DynamoRIO, like Embra [37] and Dynamo [4], inlines one target of an indirect branch when it builds a trace across the branch. However, whenever the indirect branch has a target other than the inlined target, a hashtable lookup is required. This lookup is the single greatest source of overhead in DynamoRIO. To mitigate the overhead, a series of compares and conditional direct branches for each frequent target are inserted prior to the hashtable lookup. This is similar to the "inline caching" of virtual call targets in Smalltalk [14] and Self [21], but applied to returns and indirect jumps as well as indirect calls.

The optimization works as follows: when an indirect branch inlined in a trace has a target different from that recorded when the trace was created, it usually transfers

```
call prof_routine
jmp hashtable_lookup
```

⇓

```
cmp real_target, hot_target_1
je hot_target_1
cmp real_target, hot_target_2
je hot_target_2
call prof_routine
jmp hashtable_lookup
```

Figure 4. Code transformation by our indirect branch dispatch optimization. A profiling routine rewrites its own trace to insert dispatches for the hottest targets among its samples, avoiding a hashtable lookup.

control to the hashtable lookup routine. The optimization diverts that control transfer to a code sequence at the bottom of the trace. This code sequence consists of a series of compare-plus-conditional-branch pairs followed by a call to a profiling routine, as shown in Figure 4. After the call is a jump to the hashtable lookup routine. Initially there are no compare-branch pairs and control immediately goes to the profiling call. The profiling routine records the target of the indirect branch each time it is called. Once a threshold is reached in the number of samples collected, the profiling routine rewrites the trace to add compare-branch pairs for the hottest targets. The profiling call is kept in the trace but is only reached if none of the hot targets are matched, adaptively replacing the hashtable lookup with a series of compares and direct branches.

No profiling is done to determine if the inserted targets remain hot; once a target is inserted, it is never removed. Improving this is an area of future work, requiring the development of always-on, low-overhead profiling techniques.

4.4 Custom Traces

As an example of our custom trace interface, we built a client that attempts to inline entire procedure calls into traces. The standard DynamoRIO traces focus on loops and often end up with a hot procedure call's return in a different trace from the call. This causes many hashtable lookups as the call is invoked from different call sites and the inlined return target keeps missing.

Our custom traces simply mark calls as trace heads and returns as end-of-trace conditions. A trace will be terminated if a maximum size is reached, to prevent too much unrolling of loops inside calls. Once a return is reached, the trace is ended after the next basic block. This inlines the return and nearly guarantees that the inlined target will match. Our implementation goes ahead and assumes that the calling convention holds, in which case the return can be removed entirely.

5 Experimental Results

This section shows the performance results of the optimizations from Section 4. All of the results in this section are for the SPEC2000 benchmarks [34] (excluding the FORTRAN 90 benchmarks) on Linux, compiled with full optimization (`gcc -O3`) and run with unlimited code cache space on a Pentium 4 2.2GHz Xeon. The best of four runs was used for each data point.

Figure 5 shows normalized execution time (the ratio of our time to native execution time, so smaller is better) for six data points. The first bar gives the performance of the base DynamoRIO infrastructure. DynamoRIO breaks even on many benchmarks, even though it is not performing any optimizations beyond efficient code layout when creating traces. For the benchmarks with slowdowns, most of the overhead comes from handling indirect branches and dealing with `eflags` changes caused by introduced code. DynamoRIO suffers from more costly indirect branch mispredictions than the native application, as it translates all indirect branches (including returns and indirect calls) into indirect jumps. The Pentium processors have return address predictors, but not indirect jump predictors, penalizing DynamoRIO, which cannot efficiently use the return address predictor (also, to do so would require storing code cache addresses on the stack, violating transparency).

Comparing DynamoRIO's base performance to that of Dynamo [4], the underlying architecture's treatment of indirect branches is the key difference, with CISC versus RISC secondary. Dynamo ran on PA-RISC, which does not have a return address predictor. Dynamo's sole goal was optimization, and it gave up (returning control to native execution) if it was not performing well. DynamoRIO is a platform for dynamic code modification, not just optimization, and as such it maintains control over the entire application run.

The second bar in Figure 5 gives the performance for our redundant load removal optimization. This optimization achieves a forty percent speedup for `mgrid` and also does well on a number of other floating-point benchmarks. Its effects on the integer benchmarks are less dramatic. The third bar shows the results for the `inc` to `add 1` transformation, which is able to speed up a number of benchmarks. The fourth bar gives the performance of the adaptive indirect branch target optimization. It does quite well on several of the integer benchmarks. The fifth bar shows the custom traces optimization of. It speeds up a number of the integer benchmarks. We have not finished tweaking the custom trace parameters, and we hope to find trace strategies that perform well even for benchmarks like `perlbmk` and `gcc`.

Our optimizations result in slight slowdowns relative to base DynamoRIO performance on a few benchmarks. The largest slowdowns are on `perlbmk` and `gcc`. Both of these consist of multiple short runs with little code re-use. It is

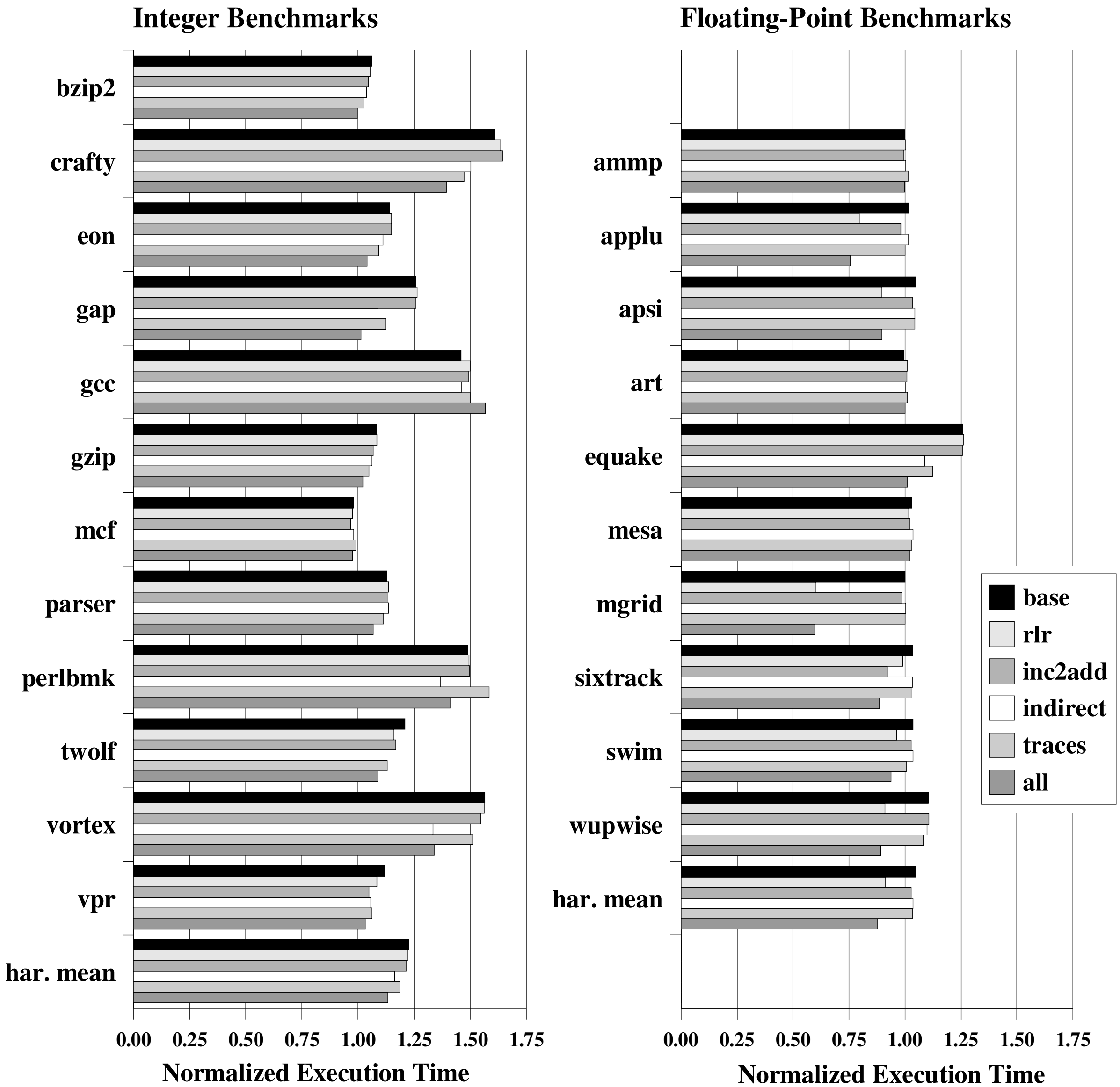

Figure 5. Normalized program execution time (the ratio of our execution time to native execution time) on the SPEC2000 benchmarks [34]. Six data points are shown: the base DynamoRIO performance, each of our four sample optimizations applied independently, and all of them applied in combination.

difficult to amortize overheads in such conditions. The time spent performing the optimizations outweighs any benefits for these benchmarks.

The final bar in Figure 5 gives the performance of running all four of our sample optimizations at once. The mean execution time for the floating-point benchmarks is a 12% improvement over native. Combining floating-point and integer, the mean performance exactly matches native, a 12% improvement over the base DynamoRIO performance. We hope to improve the efficiency of the DynamoRIO infrastructure itself in the future, to achieve better end results.

6 Related Work

There are two other systems we know of that export an API for the creation of custom dynamic optimizations, DELI [13] and Strata [33]. DELI has hooks for transforming traces as they are built, but has no mechanism for re-optimizing traces after they have been placed in the code cache. As with Dynamo [4], profiling is used only up front to build a trace: once built, a trace is no longer profiled. To adapt to changes in program behavior, the entire cache must be flushed, which is too coarse-grained for general adaptive optimizations. Furthermore, DELI's instruction representation has a single level of detail.

Strata [33] separates part of its system into a client interface which can be modified to build custom dynamic code modification tools. The interface includes hooks into Strata's fragment creation and emission routines, but has no support for re-optimizing fragments once they are in the cache. The instruction representation is not discussed.

API-less dynamic optimization systems include Dynamo [4] for PA-RISC; Wiggins/Redstone [12], which employs program counter sampling to form traces which are then specialized for a particular Alpha machine; and Mojo [7], which targets Windows NT running on IA-32, but has no available information beyond the basic infrastructure of the system. Kistler [24] proposes "continuous program optimization" that involves operating system re-design to support adaptive dynamic optimization.

Hardware dynamic optimization of the instruction stream is performed in superscalar processors. The Trace Cache [32] allows such optimizations to be performed off of the critical path.

Dynamic translation systems resemble dynamic optimizers in that they cache native translations of frequently executed code. Domains include instruction set emulation [9, 17] and binary compatibility [8, 25]. Recent dynamic translation systems such as UQDBT [36] and Dynamite [31] separate the source and target architectures to create extensible systems that can be re-targeted.

Dynamic compilation has proven essential for efficient implementation of high-level languages [14, 1]. Some just-in-time compilers perform profiling to identify which methods to spend more optimization time on [22]. The Jalapeño Java virtual machine [3, 26] utilizes idle processors in an SMP system to optimize code at runtime. Jalapeño optimizes all code at an initial low level of optimization, embedding profiling information that is used to trigger re-optimization of frequently executed code at higher levels. Self [21] uses a similar adaptive optimization scheme.

Staged dynamic compilers postpone a portion of compilation until runtime, when code can be specialized based on runtime values [11, 19, 27, 28, 18]. These systems usually focus on spending as little time as possible in the dynamic compiler, performing extensive offline pre-computations to avoid needing any intermediate representation at runtime.

Dynamic instrumentation can be used to build runtime code analyzers and, to some degree, runtime code modifiers. Both Dyninst [6] and Vulcan [35] can insert code into running processes. Dyninst is based on dynamic instrumentation technology [20] developed as part of the Paradyn Parallel Performance Tools project [29]. Because these tools modify the original code by inserting trampolines, extensive modification of the code is unwieldy.

Other related fields include link-time optimization [30, 10] and low-overhead profiling [2, 15].

7 Conclusions

This paper presents a flexible yet efficient infrastructure for the development of adaptive dynamic optimizations. Dynamic optimization has great potential to solve the problems of static compilation of modern, dynamic software. However, there are few dynamic optimization infrastructures, due to the engineering challenges in building the core system and strict requirements of efficiency and transparency for operating while the program is executing.

The key principles used by our infrastructure to maintain efficiency are restricting optimization units to linear streams of code (*traces*) and using adaptive levels of detail for representing instructions. Our interface provides direct support for building customizable traces and custom adaptive optimization of traces, while maintaining transparency with respect to the application.

We have demonstrated the usefulness and effectiveness of our framework with several example optimizations. We do not rely on hardware, operating system, or compiler support, and operate on unmodified binaries on both generic Linux and Windows IA-32 platforms.

Our infrastructure is general enough to be used for purposes other than optimization. Potential applications are numerous: instrumentation, profiling, statistics gathering, sandboxing, intrusion detection, on-the-fly code decompression or decryption, code streaming, dynamic translation. The benefits are vast for a dynamic code modification infrastructure that is general while maintaining efficiency.

References

[1] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *Proceedings of the SIGPLAN'98 Conference on Programming Language Design and Implementation*, June 1998.

[2] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *16th ACM Symposium on Operating System Principles (SOSP '97)*, Oct. 1997.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, Oct. 2000.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent runtime optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, June 2000.

[5] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework

for Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, Dec. 2001.

[6] B. R. Buck and J. Hollingsworth. An API for runtime code patching. *Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[7] W.-K. Chen, S. Lerner, R. Chaiken, and D. M. Gillies. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, Dec. 2000.

[8] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. FX!32: A profile-directed binary translator. *IEEE Micro*, 18(2), Mar. 1998.

[9] R. F. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *SIGMETRICS*, 1994.

[10] R. Cohn and P. G. Lowney. Hot cold optimization of large Windows/NT applications. In *29th Annual International Symposium on Microarchitecture (MICRO '96)*, Dec. 1996.

[11] C. Consel and F. Nöel. A general approach for run-time specialization and its application to C. In *ACM Symposium on Principles of Programming Languages (POPL '96)*, Jan. 1996.

[12] D. Deaver, R. Gorton, and N. Rubin. Wiggins/Restone: An on-line program specializer. In *Proceedings of Hot Chips 11*, Aug. 1999.

[13] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. DELI: A new run-time control point. In *35th Annual International Symposium on Microarchitecture (MICRO '02)*, Nov. 2002.

[14] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *ACM Symposium on Principles of Programming Languages (POPL '84)*, Jan. 1984.

[15] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00)*, Oct. 2000.

[16] DynamoRIO dynamic code modification system binary package release. MIT and Hewlett-Packard, June 2002. `http://www.cag.lcs.mit.edu/dynamorio/`.

[17] K. Ebcioglu and E. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *24th Annual International Symposium on Microarchitecture (ISCA '97)*, June 1997.

[18] E. Feigin. *A Case for Automatic Run-Time Code Optimization*. Senior thesis, Harvard College, Division of Engineering and Applied Sciences, Apr. 1999. `http://www.eecs.harvard.edu/hube/publications/feigin-thesis.pdf`.

[19] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*, May 1999.

[20] J. Hollingsworth, B. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Proceedings of the 1994 Scalable High-Performance Computing Conference*, May 1994.

[21] U. Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford University, 1994. `http://www.cs.ucsb.edu/oocsb/papers/urs-thesis.html`.

[22] The Java HotSpot performance engine architecture. `http://java.sun.com/products/hotspot/whitepaper.html`.

[23] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, Aug. 2002.

[24] T. Kistler and M. Franz. Continuous program optimization: Design and evaluation. *IEEE Transactions on Computers*, 50(6), June 2001.

[25] A. Klaiber. The technology behind Crusoe processors. Transmeta Corporation, Jan. 2000. `http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf`.

[26] C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8), Mar. 2001.

[27] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, May 1996.

[28] M. Leone and R. K. Dybvig. Dynamo: A staged compiler architecture for dynamic program optimization. Technical Report 490, Department of Computer Science, Indiana University, Sept. 1997.

[29] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, 28(11):37–46, Nov. 1995.

[30] R. Muth, S. Debray, S. Watterson, and K. D. Bosschere. alto : A link-time optimizer for the Compaq Alpha. *Software Practice and Experience*, 31:67–101, Jan. 2001.

[31] A. Robinson. Why dynamic translation? Transitive Technologies Ltd., May 2001. `http://www.transitive.com/documents/Why_Dynamic_Translation1.pdf`.

[32] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *29th Annual International Symposium on Microarchitecture (MICRO '96)*, Dec. 1996.

[33] K. Scott and J. Davidson. Strata: A software dynamic translation infrastructure. In *Proceedings of the IEEE 2001 Workshop on Binary Translation*, July 2001.

[34] SPEC CPU2000 benchmark suite. Standard Performance Evaluation Corporation. `http://www.spec.org/osg/cpu2000/`.

[35] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, Apr. 2001.

[36] D. Ung and C. Cifuentes. Machine-adaptable dynamic binary translation. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, Jan. 2000.

[37] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1996.