Dynamic Native Optimization of Interpreters

Gregory T. Sullivan
gregs@ai.mit.edu
Artifical Intelligence Lab

Derek L. Bruening
iye@lcs.mit.edu
Laboratory for Computer Science

Iris Baron
iris@lcs.mit.edu
Laboratory for Computer Science

Timothy Garnett
timothyg@lcs.mit.edu
Laboratory for Computer Science

Saman Amarasinghe
saman@lcs.mit.edu
Laboratory for Computer Science

Massachusetts Institute of Technology
Cambridge, MA

Abstract

For domain specific languages, "scripting languages", dynamic languages, and for virtual machine-based languages, the most straightforward implementation strategy is to write an interpreter. A simple interpreter consists of a loop that fetches the next bytecode, dispatches to the routine handling that bytecode, then loops. There are many ways to improve upon this simple mechanism, but as long as the execution of the program is driven by a representation of the program other than as a stream of native instructions, there will be some "interpretive overhead".

There is a long history of approaches to removing interpretive overhead from programming language implementations. In practice, what often happens is that, once an interpreted language becomes popular, pressure builds to improve performance until eventually a project is undertaken to implement a native *Just In Time* (JIT) compiler for the language. Implementing a JIT is usually a large effort, affects a significant part of the existing language implementation, and adds a significant amount of code and complexity to the overall code base.

In this paper, we present an innovative approach that dynamically removes much of the interpreted overhead from language implementations, with minimal instrumentation of the original interpreter. While it does not give the performance improvements of hand-crafted native compilers, our system provides an appealing point on the language implementation spectrum.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.4 [**Programming Languages**]: Processors—*Interpreters*

1 Introduction

Certain kinds of programming languages lend themselves to an interpreted implementation: languages with a high degree of dynamism (e.g., dynamic OO languages), or with a premium on fast startup and smaller programs (e.g., scripting languages), or where simplicity of implementation is important. Indeed, it can be argued that nearly all sufficiently complex applications have elements of interpretation throughout[1].

Typically, the language front end will transform the surface syntax of a program to an intermediate representation of the program (e.g., bytecodes) that is then interpreted. A simple interpreter consists of a loop that fetches the next bytecode, dispatches to the routine handling that bytecode, then loops. There are many ways to improve upon this simple mechanism, but as long as the execution of the program is driven by a representation of the program other than as a stream of native instructions, there will be some "interpretive overhead".

We take an approach that is a combination of native *Just In Time* (JIT) compiler and *partial evaluation* techniques. We start with a dynamic optimization system called *DynamoRIO*, jointly developed at HP Labs and MIT. We give a quick overview of the DynamoRIO system in Section 2.

DynamoRIO records sequences of native instructions, which are subsequently partially evaluated with respect to the in-memory representation of the program being interpreted.

Suppose the application program is represented as an immutable vector of bytecodes, and we have a long trace of native instructions. If we look carefully at the trace, there will typically be a value (we call it the "Logical PC") used as an index into the bytecode vector, and there will be sequences of instructions as follows:

[1]Greenspun's Tenth Rule of Programming: "Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp."

1. Fetch the next bytecode, using the logical PC as an index into the vector;
2. Possibly dereference other nearby bytecodes, to be used as arguments to the handler for this bytecode;
3. A certain amount of conditional control flow may take place based on the value of the bytecode (e.g., a `switch` statement);
4. Jump to bytecode-specific instructions;
5. Increment the logical PC by a fixed amount;
6. Repeat.

The key insight of dynamic native partial evaluation is that if we know the starting logical PC at the start of the above sequence, then:

1. Dereferences from the bytecode array, indexed by the logical PC, can be statically folded to constants,
2. Conditional branches based on now constant values can be removed entirely, and
3. Direct increments to the logical PC can be identified and tracked, thus enabling continued partial evaluation.

2 The DynamoRIO Dynamic Optimization Framework

Our optimization infrastructure is built on a dynamic optimizer called DynamoRIO. DynamoRIO is based on 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. 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, 31].

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.

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

3 Interpreters Confound DynamoRIO

An important heuristic upon which DynamoRIO's trace identification and collection strategy is based is that popular targets of jumps are good candidates for *trace heads*. Furthermore, it is assumed that execution will proceed most or all of the way through a trace most of the time. In other words, recording long instruction streams starting at common branch targets will result in execution spending most of its time in recorded traces.

Unfortunately, typical interpreter implementations foil DynamoRIO's trace collection heuristic. In the case of a simple read-switch-jump loop, the interpreter's hottest loop head is in fact a poor candidate for a trace head, as its body follows a different path for each byte code type.

Threaded interpreters pose a related problem for DynamoRIO. The pervasive use of indirect jumps for control transfer foils DynamoRIO's trace head identification heuristics, and, even if a trace head were to be identified, the address of, e.g., the `CALL` bytecode does not uniquely identify a commonly occurring long sequence of native instructions.

3.1 Logical PC's

The goal is to record a long series of native instructions that:

1. will often be invoked, and
2. will usually run most or all of the way to the end once started.

An important realization for applying dynamic optimization techniques to interpreters is that we need a pair, (*native PC, logical PC*) to uniquely identify the current overall computation point. By *logical PC*, we mean some unique identifier into the control flow structure of the interpreted application. Neither the logical nor native PC is sufficient. For example, the native PC corresponding to the start of the handler for a `CALL` bytecode would be executed for each call site encountered in an interpreted application. The logical PC, on the other hand, might stay constant over a significant amount of interpreter execution (consider the interpretation of the `invokevirtual` JVM instruction). We call the (*logical PC, native PC*) pair the *abstract PC*.

4 Instrumenting the Interpreter

There are two types of information the interpreter writer must supply to DynamoRIO: the identification of logical control flow actions and the location of the immutable program representation used to drive interpretation.

4.1 Logical Control Flow API

Every time that either the native PC changes (either by sequential execution or a branch or jump) or the logical PC changes (by a

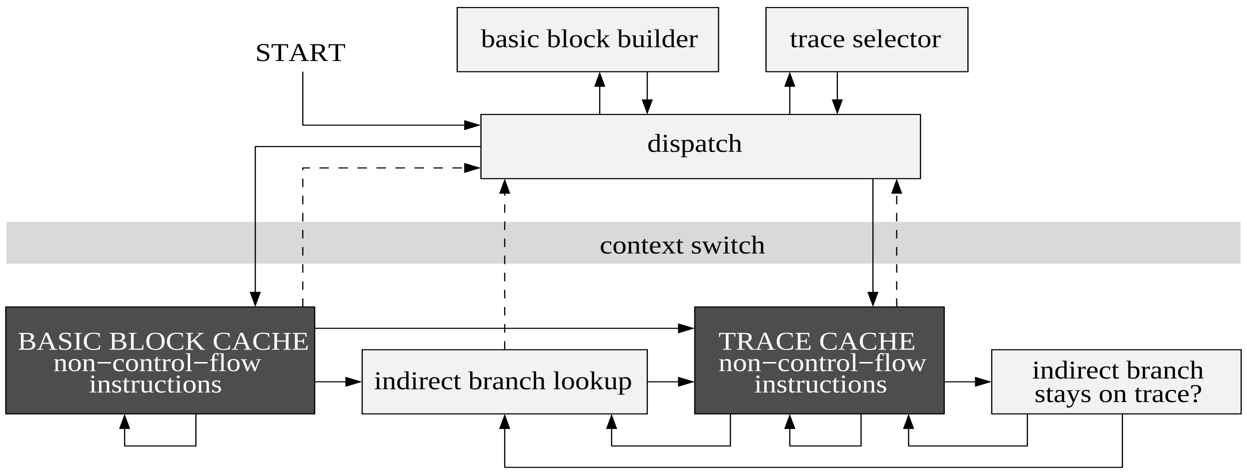

Figure 1. Flow chart of the DynamoRIO System. Dark shading indicates application code.

change in the controlling state of the interpreter), the abstract PC has changed. DynamoRIO provides the interpreter writer with a simple API for identifying relevant changes in the control state of the interpreter. Calls to these API functions enable DynamoRIO to identify potential *trace heads* and also to link traces together. Each of the three API functions correspond to typical native control transfer operations:

`logical_direct_jump(new_logpc)` corresponds to a direct native jump. By calling logical_direct_jump(x) at a particular point (native PC) in the interpreter, call it *n*, the interpreter writer promises that if the interpreter ever reaches *n* with the logical PC equal to its current value, say *y*, it will always make the logical control transfer to the next interpreter instruction (call it $n+1$) and logical PC *x*.

`logical_indirect_jump(new_logpc)` corresponds to an indirect native jump. As you might guess, this means that the interpreter cannot make the guarantee required for `logical_direct_jump`. A classic case of an indirect logical jump is the implementation of a `RETURN` bytecode, where the actual target is based on runtime data (e.g., a value on the stack) rather than a compile time constant.

`logical_relative_jump(offset)` corresponds to sequential native execution. Calling this function has the same guarantee as `logical_direct_jump` (that is, the current (native PC, logical PC) will always advance to (native PC + 1, logical PC + offset)) but also tells DynamoRIO that this transfer is "uninteresting". This corresponds to the fact that sequential execution through native code also involves regular changes to the native PC, but these are uninteresting control transfers.

Each of the three `logical_*_jump` functions identifies, when called, an abstract PC. The interpreter writer provides the logical PC value, and DynamoRIO provides the native PC value, for any given call. The native PC value for a logical control transfer is the address of the instruction following the call to the logical PC API function.

4.2 Identifying Immutable Program Data

In order for constant propagation to be able to fold dereferences of the program representation to constant values, the interpreter writer must identify regions of memory that hold immutable representations of the application, as well as identifying other memory locations that will be constant for given *(logical PC, abstract PC)* pairs. There are three API functions for providing this information:

`set_region_immutable(start, end)` identifies the region of memory delimited by `start` and `end` (inclusive) as immutable. `set_region_immutable` may be called for any number of regions. Once called for a given *start - end* region, that region of memory must not be written to for the duration of the program run.

`add_trace_constant_address(addr)` identifies an address whose value (i.e. when dereferenced) is guaranteed to be the same whenever control reaches the abstract PC of the call. For example, suppose a call `add_trace_constant_address(&pc)` is made when the logical PC is currently *lpc*, the native PC of the call site is *npc*, and the value of `pc` is 42. If DynamoRIO makes a trace headed by abstract PC *(lpc, npc)*, then any dereferences of `pc` can be folded to the constant 42. Furthermore, if this constant is then used to dereference values in immutable memory (e.g. `bc = byte_codes[pc]`) that dereference can also be folded to a constant.

`set_trace_constant_stack_address(addr, val)` identifies a stack-allocated variable (at address `addr` on the stack) that currently has the value `val`. The meaning of a call `set_trace_constant_stack_address(&pc, pc)` is just like a call to `add_trace_constant_address(&pc)` described above, except that the current value (`pc`) is made explicit in the call so that Dynamo can note the stack offset calculation used to dereference the stack-allocated variable. DynamoRIO will only fold dereferences of `addr` to a constant when control is within the stack frame of the call to `set_trace_constant_stack`.

If an address is identified as containing a trace constant, that address must not be *aliased*. That is, if `pc` is identified as a trace constant, changes to the value stored in `pc` must be apparent in the instruction sequence, rather than indirectly through another memory location.

For example,

```
ByteCode* pc = ...;
ByteCode* alias = pc;
add_trace_constant_address(&pc);
*alias = some_bytecode;
```

would violate the anti-aliasing requirement.

5 Example: TinyVM

In this section, we present a simple VM called `TinyVM`, written in C, to demonstrate the instrumentation by a programmer and the run-time optimization performed by DynamoRIO[2].

We start by adding calls to `dynamorio_app_init` and `dynamorio_app_start` to the startup code of TinyVM. Similarly, we arrange to call `dynamorio_app_stop` and `dynamorio_app_exit` when TinyVM exits. This is standard procedure for using DynamoRIO, as explained in the DynamoRIO documentation.

The main dispatch loop of TinyVM deconstructs the element of the bytecode vector `instrs` pointed to the logical PC `pc`, then finds the matching case in a `switch` statment:

```
loop:
  op = instrs[pc].op;
  arg = instrs[pc].arg;
  switch (op) {
    ...
```

Calls to functions are statically dispatched, and the target (bytecode offset) of the call is embedded in the bytecode stream. Thus, a call is a direct jump (i.e., from a given callsite, control will always transfer to the same target), and we instrument the handling of the `CALL` opcode as follows:

```
case CALLOP:
  ... setup new call frame ...
  pc = arg;     /* go to start of function body */
  dynamorio_logical_direct_jump(pc);
  goto loop;
```

As mentioned earlier, a `RETURN` bytecode is a case of an indirect logical control transfer. We instrument the `RETURN` case as follows:

```
case RETOP:
  ... clean up stack ...
  pc = pop_raw();  /* pop the return PC */
  dynamorio_logical_indirect_jump(pc);
  *(++sp) = val;   /* put return value back on stack */
  goto loop;
```

The case for bytecodes implementing conditional branches is more interesting. You might think that the logical branches would be indirect, but in fact they are often direct. In the following code, implementing the `BEQ` bytecode, the true branch is a direct logical jump (recall that the `arg` value came from the bytecode vector), and the false branch is a direct relative branch:

```
case BEQOP:
  n = pop_int();     /* top must be an int */
  if (n) {           /* not equal 0 */
    pc++;            /* just continue */
    dynamorio_logical_relative_jump(1);
  } else {           /* equal 0 */
    pc = arg;        /* do the branch */
    dynamorio_logical_direct_jump(pc);
  }
  goto loop;
```

The reason the logical transfers are direct is that the native conditional branch has already distinguished between the true and false cases. For each call to `dynamorio_logical_*`, the combination of the current native PC and the current logical PC uniquely identifies the target (native PC, target PC) abstract PC.

Finally, we inform DynamoRIO of immutable regions of memory and of any logical PC pointers:

```
set_region_immutable(instrs,
      (instrs + num_instrs*sizeof(ByteCode)−1));
dynamorio_add_trace_constant_address(&pc);
```

6 Collecting Logical Traces

The interpreter writer supplies the target logical PC (or offset) to the `dynamorio_logical_*` calls. When first processing the basic block, DynamoRIO adds the global DynamoRIO context, the native PC, and some “from context” data to the call also, so the actual functions receive four arguments. The translation of a call `logical_direct_jump(x)` at native PC *n* is given in Figure 2.

```
  push x              ; logical pc
  call set_log_pc     ; saves logpc (x) in a global
  pop 1
  jmp exit_stub
...
exit_stub:
  push dcontext       ; add arg. for logical_jump call
  push n+1            ; add native PC arg.
  push from_data      ; for linking traces
  call logical_direct_jump
  test next_logical_trace   ; if logical jmp to trace
  jne *next_logical_trace   ; go there
  next_addr = n+1     ; otherwise
  jmp dynamo_dispatch ; go to dynamorio with native PC = n+1
```

Figure 2. Translation of `logical_direct_jump(x)` at instruction *n*

Given the target logical and native PC’s, DynamoRIO proceeds as

[2]TinyVM will be available in the released version of DynamoRIO as an example.

outlined by the following pseudo-code:

```
logical_jump(dcontext, native_pc, logical_pc, from) {
  if (currently building a trace) {
    set flag telling DynamoRIO to finish trace;
  } else { /* if not currently building a trace */
    /* get logical basic block for (logical, native) abstract PC */
    lbb = lookup(native_pc, logical_pc),
          adding new entry if necessary;
    lbb->count++;   /* count hits */
    if ((native PC, logical PC) entry corresponds
          to a trace) {
      if (we are coming from a trace &&
           this is a direct logical jump) {
        link the two traces together;
      }
      next_logical_trace = lbb->trace;
    } else { /* lbb not a trace */
      /* should we build a trace? */
      if (lbb->count > hot_threshold) {
        set flags to start DynamoRIO tracing;
      }
    }
  }
}
```

When `logical_jump` is called during trace building mode with a direct or indirect logical jump, we signal DynamoRIO to finish building the trace. Thus each trace is headed by the target of a logical control transfer and ends with another logical control transfer. When DynamoRIO installs the trace, the trace is associated with the (*logical PC, native PC*) pair in the logical basic block table.

The first time in `logical_jump` for a given (*logical PC, native PC*) abstract PC, the pair will not be found in the “logical basic block table”, and an entry will be added.

After the same abstract PC has been seen `hot_threshold` times, from direct or indirect logical jumps, we set flags to signal DynamoRIO’s top level loop that we should start recording a trace, keyed by the (*logical PC, native PC*) pair. While trace building is active, DynamoRIO collects each basic block encountered into a contiguous list of instructions (this is regular DynamoRIO trace collection).

The next time `logical_jump` encounters an abstract PC for which a trace exists, it sets `next_logical_trace` (a field in the thread-local `dcontext` structure), and when control returns to the exit stub, the exit stub jumps directly to that target logical trace, without returning to DynamoRIO’s dispatch loop.

Finally, if a logical basic block entry has a corresponding trace, *and* the `from-data` supplied by DynamoRIO indicates that we are following a direct logical control transfer from another trace, we *link* the two traces.

Linking two traces involves replacing the `jmp exit_stub` instruction in Figure 2 with a jump directly to the target logical trace fragment.

7 Trace Optimization

By organizing traces by the logical and native PC’s, instead of the native PC’s that regular DynamoRIO uses, we improve performance so that the DynamoRIO system has little overhead compared to running without DynamoRIO. In some cases, we even improve performance over native (not under DynamoRIO) execution.

We then apply relatively simple but aggressive constant propagation to the recorded traces.

7.1 Constant Propagation

We apply standard constant propagation, made more challenging by the fact that we are doing it for the X86 instruction set. Crucial to the success of constant propagation is being able to fold references into immutable memory to constants. As mentioned earlier, we use `set_region_immutable` calls from the interpreter to establish constant memory regions. Furthermore, the `*trace_constant*` calls register addresses whose contents we can rely on at the start of any given trace.

7.2 Call-Return Matching

DynamoRIO performs expensive checking when a return is encountered, because a return is basically an indirect branch. If static analysis of the trace reveals that the return matches a call site in the trace, then the return overhead can be elimintated. By removing some jumps, call-return matching enables other optimizations.

7.3 Dead Code Elimination

Constant propagation produces dead code, such as from storing no longer needed temporary values or from statically dispatching conditionals, and DynamoRIO collects dead code during its optimization.

8 Experimental Results

Figure 3 shows the running times of six benchmarks, normalized to the execution time on TinyVM without DynamoRIO.

We see that running under regular Dynamo slows down the application significantly (by almost a factor of two for some of the benchmarks). While adding logical tracing recovers some performance (fib-iter, sieve), it is clear that we need both logical tracing and optimization.

Fragment	Size	Time	Exits
1	1201	30.5%	S1: 66.2% S7: 6.8% S10: 27.0%
2	1849	6.1%	S15: 100.0%
3	1909	6.1%	S7: 2.4% S16: 97.6%
4	366	5.9%	S1: 18.2% S2: 14.9% S3: 66.9%

Figure 4. Fragment Size and Exit Statistices, Sieve Benchmark, regular DynamoRIO

The primary reason that regular DynamoRIO loses so much performance can be gleaned from the results of DynamoRIO’s profiling tools. Figure 4 shows the top four traces for a run of a Sieve of Eratosthenes finding the first 30,000 prime numbers. Not only are the traces relatively short (about 1KB), but the hottest trace exits from its first exit (a conditional, taking the non-trace branch) 66% of the time.

Figure 5 shows the same profiling information for the application running under DynamoRIO with logical PC tracking. We see much longer traces, and the hottest trace exits at the end 97% of the time.

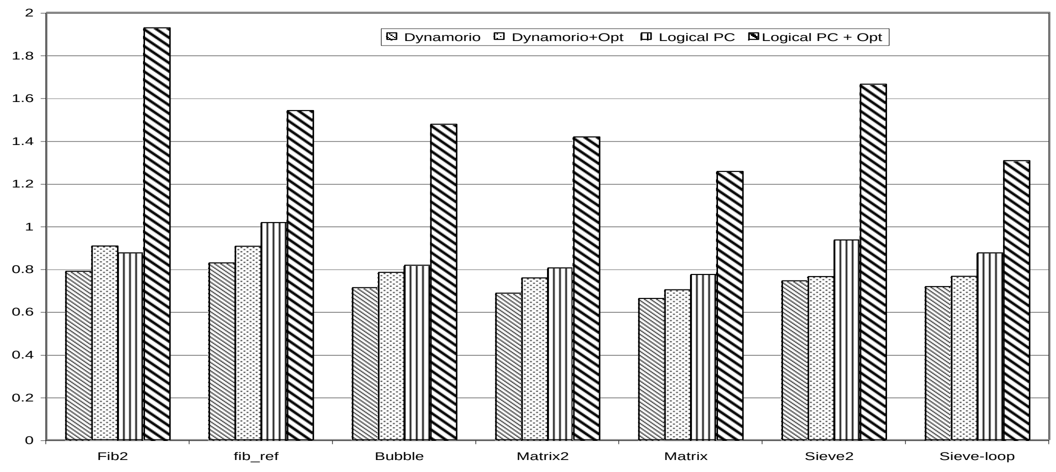

Figure 3. Running Times in Four Configurations, Normalized against Native (Without DynamoRIO)

Fragment	Size	Time	Exits
1	25255	48.1%	S34: 2.4% S242: 97.6%
2	36301	29.0%	S347: 100.0%
3	18834	15.0%	S161: 10.8% S178: 89.2%
4	10119	7.1%	S97: 100.0%

Figure 5. Fragment Size and Exit Statistices, Sieve Benchmark, DynamoRIO and Logical PC Tracing

In the fib-recurse benchmark, we actually see a slowdown when running with logical tracing instead of regular DynamoRIO. This is because of a large number of returns, which cause a lookup in our logical basic block table.

Figure 6 compares the contributions of the different optimizations we apply. It is surprising that constant propagation by itself does not contribute much. We get the single biggest improvement from call-return matching.

9 Related Work

Moore and Leach [27] describe writing threaded interpreters. Piumarta and Riccardi [29] go further with dynamically generated bytecode sequences.

Dynamic compilation has proven essential for efficient implementation of high-level languages [13, 1]. Some just-in-time compilers perform profiling to identify which methods to spend more optimization time on [20]. The Jalapeño Java virtual machine [3, 23] 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 reoptimization of frequently executed code at higher levels. Self [19] 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, 17, 24, 25, 16]. 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.

API-less dynamic optimization systems include Dynamo [4] for PA-RISC; Wiggins/Redstone [12], which employs program counter sampling to form traces that are 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 [21] 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 [31] 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, 15] and binary compatibility [8, 22]. Recent dynamic translation systems such as UQDBT [33] and Dynamite [30] separate the source and target architectures to create extensible systems that can be re-targeted.

Dynamic instrumentation can be used to build runtime code analyzers and, to some degree, runtime code modifiers. Both Dyninst [6] and Vulcan [32] can insert code into running processes. It is based on dynamic instrumentation technology [18] developed as part of the Paradyn Parallel Performance Tools project [26].

Other related fields include link-time optimization [28, 10] and low-overhead profiling [2, 14].

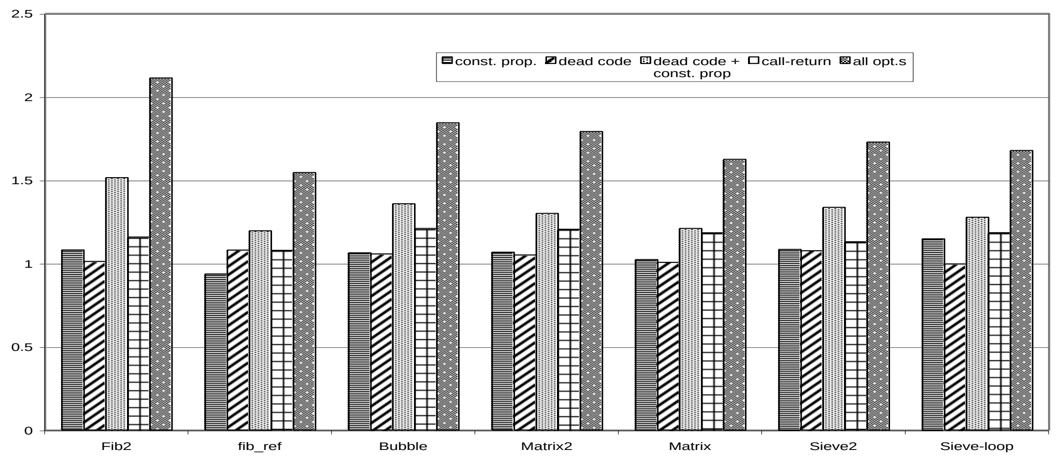

Figure 6. Contributions of Separate Optimizations, Normalized to DynamoRIO with Logical Tracing

10 Future Work

The research presented in this paper is a proof-of-concept for the idea of applying general purpose dynamic optimization techniques and native dynamic partial evaluation to interpreters.

We are currently working on applying this version of DynamoRIO to "real" interpreters, including OCAML and Kaffe. OCAML is a well-implemented threaded interpreter for a variant of the ML language. Kaffe is a very slow implementation, using recursive tree walking, of Java; Kaffe's interpreter can afford to be extremely slow, because the implementation also includes a reasonable JIT compiler. We expect very good results from applying our technology to OCAML. For Kaffe, we are generalizing the DynamoRIO-interpreter API to allow for multiple "logical PC" like values and to allow those values to be on the stack (to handle recursive interpretation).

There are also a number of straightforward engineering improvements that should improve the performance of the system.

We do not handle long loops within single bytecode implementations – they are unrolled in place. Currently we simply specify a maximum number of blocks per trace, and truncate the trace if we hit the maximum. We would like to avoid this sort of loop unrolling in the first place, and translate back jumps as jumps.

We are investigating adding more sophisticated alias analysis in order to enable constant propagation to track constant values through memory.

11 Conclusion

There will always be interpreted languages, and only some implementation efforts can afford the time and money to produce a native compiler.

We present a novel approach to improving the performance of interpreters. The interpreters do not have to be written in a particular style; they need only annotatations identifying a logical PC and regions of immutable memory (particularly the program representation). DynamoRIO then performs trace recording and aggressive partial evaluation of the X86 code to produce substantially optimized traces of the running interpreter. We have applied these techniques to simple interpreters and shown that this approach can provide substantial speedups with minimal instrumentation from the interpreter writer. We are currently working on applying these techniques to more sophisticated interpreters.

12 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] 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.

[14] 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.

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

[16] E. Feigin. *A Case for Automatic Run-Time Code Optimization*. Senior thesis, Harvard College, Division of Engineering and Applied Sciences, Apr. 1999.

[17] 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.

[18] 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.

[19] U. Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford University, 1994.

[20] The Java HotSpot performance engine architecture.

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

[22] A. Klaiber. The technology behind Crusoe processors. Transmeta Corporation, Jan. 2000.

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

[24] 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.

[25] 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.

[26] 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.

[27] C. H. Moore and G. C. Leach. Forth – a language for interactive computing. Technical report, Mohasco Industries, Inc., Amsterdam, NY, 1970.

[28] 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.

[29] I. Piumarta and F. Riccardi. Optimizing direct-threaded code by selective inlining. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[30] A. Robinson. Why dynamic translation? Transitive Technologies Ltd., May 2001.

[31] 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.

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

[33] 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.