Improving Memory Performance through Runtime Optimization

by

Joshua Jacobs

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master's of Engineering in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2002

© Joshua Jacobs, MMII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author

Department of Electrical Engineering and Computer Science

May 24, 2002

Certified by...

Larry Rudolph

Principle Research Scientist

Thesis Supervisor

Accepted by.....

Arthur C. Smith

Chairman, Department Committee on Graduate Students
Improving Memory Performance through Runtime Optimization

by

Joshua Jacobs

Submitted to the Department of Electrical Engineering and Computer Science on May 24, 2002, in partial fulfillment of the requirements for the degree of Master’s of Engineering in Computer Science

Abstract

Runtime optimization creates the opportunity for a number of optimizations which are not possible with conventional static compilers. This thesis presents two new optimizations for improving application memory performance within the RIO runtime optimization system. Optimizations in RIO are performed at the level of traces. Traces are frequently executed sequences of code that may span traditional basic blocks. The first optimization we describe modifies traces to remove redundant loads. The second optimization locates memory addresses that are runtime constants and specializes the traces for their values. The specialized trace then has constant propagation and arithmatic simplification performed on it. Benchmarks results are shown; the optimizations achieved speed-ups of up to 25%.

Thesis Supervisor: Larry Rudolph
Title: Principle Research Scientist
Acknowledgments

This research was performed as a part of the Malleable Caches group at the MIT Laboratory for Computer Science. It was funded in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310.

My thesis advisor Larry Rudolph has been instrumental in helping me with this work. Without his advice and help I would not have been able to complete this work. I’m also especially grateful for his flexibility in supporting me as I searched for a project that fit my interests. I’ve also worked extensively with Professor Srinivas Devadas in my years here at MIT’s Lab for Computer Science, and I’m also very grateful for all his help and advice.

I’m very grateful to Professor Saman Amarasinghe for introducing me to RIO and assisting me while I worked with his group. Derek Bruening’s work implementing the RIO system is tremendous; without his work developing the system and helping me use it, there is no way I could have performed the work that I did.

My peers here at MIT’s Lab for Computer Science have been very helpful in helping me with my work: Todd Amicon has been here keeping me sane with his wit, advice, jokes and friendship for the past two years—for all that, I’m truly appreciative. Shalini Agarwal, Amay Champaneria, and Jorge Ortiz have been immensely helpful to me, they’ve been great friends and have helped me keep my head above water throughout the course of my work.

I’m also thankful to my other colleagues David Chen, Prabhat Jain, Tom Kotwal, Vinson Lee, Enoch Peserico, and Ed Suh for their advice and for making our lab a fun place to work. Thanks to Barbara Cutler for making that nifty flowchart that helps to elucidate RIO’s operation.

My mom, Susan Jacobs, was extremely helpful to me in proof-reading this thesis (even though she is a grammar-weenie.) My dad and brother, Michael and Brian Jacobs, have also been very supportive. A bunch of my other friends have been instrumental in maintaining my mental health while working on my thesis: Ricky Debnath,
Karsci Frank, Chris Gibaldi, Aaron Griswold, Erika Kitzmiller, Vinnie Lauria, Randy Punter, Onur Usmen, the programmers at yahoo.com responsible for their fantasy baseball game, the New Orleans Police Department (keep those streets clean!), Trey Gunn, Tony Levin, Frank Zappa, and Trent Reznor.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.1</td>
<td>Static trace analysis</td>
<td>29</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Value profiling</td>
<td>29</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Analyzing value profiles</td>
<td>30</td>
</tr>
<tr>
<td>5.3</td>
<td>Specialization and optimization</td>
<td>31</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Structure of an optimized trace</td>
<td>31</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Replacing memory references with constants</td>
<td>32</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Constant propagation and arithmetic simplification</td>
<td>36</td>
</tr>
<tr>
<td>5.4</td>
<td>Performance</td>
<td>37</td>
</tr>
<tr>
<td>5.5</td>
<td>Additional work</td>
<td>38</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Identifying runtime constants</td>
<td>39</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Changing specialized values over time</td>
<td>40</td>
</tr>
</tbody>
</table>

6 Conclusion

Bibliography
Chapter 1

Introduction

This thesis presents a system for optimizing application memory performance at runtime. The information available at runtime about an application’s execution facilitates a number of unique optimization opportunities. The work described in this thesis uses runtime execution information to deal with one of the biggest issues involved in creating high performance computer architectures: the increasing disparity between processor and memory speed. For many applications, overall performance is constrained by memory latency and bandwidth, not processor speed. This issue has become increasingly critical in recent years as the performance gap between memory and processor has steadily grown [12]. To deal with this growing performance gap, a number of schemes have been developed to decrease the latency of some operations according to the runtime behavior of the system:

- Memory caches store frequently used data memory addresses in a separate, faster memory [12].
- Trace caches store instructions frequently executed consecutively in a single cache line for fast access [18].
- Value prediction minimizes stalls while waiting for loads by speculating on load values [14].
Branch prediction uses an application's previous control flow behavior to speculate on the behavior of future jumps [12].

Most existing runtime optimization schemes are hardware-based. This thesis studies a software approach for dynamic optimization.

Software runtime optimization systems offer the potential for a wider range of optimizations than hardware schemes. Software optimization allows the generation of completely new code. Hardware optimization schemes, by contrast, cannot directly alter the executing instruction stream. Although creating a software dynamic optimization system is a more complex task than creating a hardware-based system, the increased flexibility such a system offers makes it attractive.

Runtime optimization has a number of distinct advantages over compile-time optimization. Runtime optimization systems can monitor an application's typical runtime control flow patterns. This information can be used to modify the application's code layout to increase the performance of its most frequent operations. Traditional compiler-time optimizations must do their best to optimize the application without any knowledge of its runtime behavior.

An additional benefit of runtime optimization is its ability to optimize code that interfaces with dynamically loaded libraries. Although new applications increasingly make use of code in dynamically linked libraries [21], these libraries pose substantial problems for traditional optimization techniques. When compiling an application that calls external libraries, the compiler cannot safely optimize code that interfaces with these libraries, because the code contained in the library may be different at runtime than it is at compile-time. However, runtime optimization views dynamically linked libraries identically to static-code, so it has no problem optimizing them. Runtime optimizers are also capable of performing optimization according to the particular data values the application encounters at runtime. If the runtime optimizer notices that the application has a tendency to use certain values, new code can be created to speed computation with these values.

Feedback directed optimization (FDO) allows static compilers to use information gathered at runtime to optimize the performance of code they create. FDO schemes
typically compile the application in a special mode so it outputs an execution profile when run. The application is then run one or more times on sample data to create an execution profile to feed back to the compiler. The application is compiled again with the compiler using the data in the execution profile to create a new binary that is optimized for runtime performance [19]. FDO-based compilers allow statically built applications to gain some of the performance benefits of runtime optimization, but runtime optimization still holds some distinct advantages. The resulting binary of an application compiled with a FDO compiler is still a static binary. This binary is unable to adapt to different phases of execution within an application, and is unable to adapt to runtime data that may be substantially different from the training data. A correctly engineered runtime optimization system suffers from none of these limitations.

This thesis presents two different schemes for increasing application memory performance via runtime optimization. These schemes function within the RIO [6] (Runtime Introspection and Optimization) runtime optimization system for the Intel x86 architecture (IA-32)[1]. The RIO system is an IA32 implementation of the original PA-RISC based Dynamo project[4]. RIO seeks to improve application performance by reorganizing the frequently executed parts of application’s code into traces. Traces are sequences of basic blocks that are frequently executed consecutively. Traces may span basic blocks, but are constrained to have a single entry point. Trace creation can improve performance by facilitating good instruction cache and branch prediction performance. An additional advantage of trace creation is that traces’ single-entry control flow offers the potential for unique optimizations that cannot be performed on static binaries.

The first optimization scheme that this thesis presents is a redundant load removal optimization. This scheme builds on Fernandez’s work to operate on traces [10]. A trace’s single entry-point control flow allows the system to perform instruction-level code optimizations that are impossible otherwise. The second optimization scheme that this thesis presents is a tool for locating runtime constants within traces. A
scheme is presented for detecting memory addresses that are runtime constants, and specializing traces for these values.

The primary contribution of this thesis is evidence that runtime optimization can be performed efficiently on unmodified binaries. Existing runtime optimization schemes require that the application be written in an interpreted language, or with special annotations to facilitate runtime optimization. The work we present shows that execution performance of native executables can be improved by runtime optimization. Benchmark performance data is presented that shows that our redundant load removal scheme substantially improves performance for a number of SPEC CPU2000 [20] benchmarks. The performance improvement of our runtime constant optimization is smaller, but it serves as a good starting point for further research into runtime constant optimization.
Chapter 2

Related work

The body of existing research in dynamic optimization is typically concerned with using information available only at runtime to perform optimizations that increase application performance. This section describes the existing dynamic optimization research that is related to the work described in this thesis. The work in this thesis combines some of the most attractive features of the optimization systems described below.

2.1 Runtime systems

Our work builds upon the Dynamo project at Hewlett-Packard labs [4], a runtime optimization system for HP’s PA-RISC architecture running on HP/UX. The biggest difference between the two systems is that Dynamo interprets much of the application’s code in software; only traces are executed on the hardware directly. By contrast, the RIO system executes the entire application on the underlying processor directly. The PA-RISC instruction set is relatively simple, so the overhead of interpreting the full instruction set does not present the performance problems that it does on the CISC IA32 platform. This project showed substantial speedups from its trace generation mechanism, even without performing any code optimizations on traces.

Microsoft’s Mojo [8] is a dynamic optimization system for IA32 applications running on Microsoft’s Windows. The published description of Mojo focused on the issue
of optimizing applications at runtime on the Windows platform and maintaining correct execution, rather than the issue of creating a high performance optimization system. Overall application performance under Mojo is typically worse than native application execution.

2.2 Dynamic hardware

In addition to Dynamo-style runtime optimizers, there are a number of other systems that take different routes to perform the same type of runtime-optimizations. Superscalar processors [12] essentially perform runtime optimization of the application instruction sequence. These processors can keep many different instructions “in-flight.” In essence, this lets the processor dynamically re-order the instruction stream (even across branches) for increased performance. Branch prediction allows the processor to dynamically adjust to the branching pattern of the application.

Trace caches [18] are another hardware scheme for adapting to an application’s execution pattern at runtime. Trace caches keep instructions that are frequently executed consecutively in the same cache line. This allows the system to perform well on applications whose execution pattern involves branches to distant pieces of code.

2.3 Languages for runtime specialization

A different way of creating a system for optimizing applications at runtime is to write the application in a language with special features to facilitate runtime optimization. The DyC [11] system allows the user to specify which variables should be considered runtime constants. For each area of code containing runtime constants, the static compiler generates a specialized dynamic code generator. Upon entry into this region at runtime, specialized code is generated for the current value of the runtime constants. Performance is reasonable because the static code generated for the dynamic compiler is specialized according to the particular code that it is to generate.

Other systems like Tempo [16] and tcc [17] are also designed for performing run-
time specialization. These systems, like DyC, also require that the programmer manually specify the variables that should be considered runtime constants.

2.4 Runtime systems for runtime specialization

Manually selecting variables to be considered as runtime constants is a burden for the programmer. A number of systems exist that are capable of locating the runtime constants in an application and optimizing execution accordingly. Java bytecode is designed to be interpreted; its structure facilitates runtime modification. Runtime modification of native binaries is more difficult. Work has been done inside the Jalapeno JVM to dynamically adapt the compiled code to the workload [3]. Frequent control flow paths are automatically inlined. Methods are specialized when parameters are runtime constants. The Jalapeno system also introduced the idea of performing code optimization in a separate thread to run on a different processor (or idle cycles on the main processor). The idea of using a separate thread to perform optimizations with idle processing cycles has been implemented within RIO (although it is not described in this thesis).

Kistler and Franz’s work involves a similar system for dynamically monitoring an application’s execution for optimization purposes [13]. This work proposes that the optimizer run as a system-wide process, rather than each application having its own optimizer. Both Kistler and Franz’s work and Jalapeno operate on high-level object-oriented languages. This high-level language interface facilitates fast and correct replacement of the old routines with optimized routines.

2.5 Feedback-directed optimization

Feedback-directed optimization schemes perform runtime optimization in two phases. Typically an application is first run in a profiling mode. The profiling mode collects data about variable values, and identifies frequent control flow paths. This data is
used by a link-time optimizer or compiler to create a new executable binary optimized according to the application’s runtime behavior in the training run.

Calder’s work involved finding efficient ways to collect value profiles without excessive slowdown [7]. Once an application’s value profile is collected, runtime constants can be determined and the code can be optimized accordingly. Muth, Watterson, et al. built upon Calder’s work and used Compaq’s Alto link-time editor to specialize code for runtime constants as determined from the profiling runs [15].

Another class of feedback-directed optimization schemes focus on optimizing using feedback from training runs to optimize the frequently executed control flow paths of the application. The FX32 IA32 emulator is a tool for running Microsoft Windows applications in a Unix environment on the Alpha platform [9]. One of the biggest hurdles in creating an emulator that performs well is dealing with control flow instructions. To deal with this problem, a control flow profiler was integrated into their emulator. The results of this profiling are used to optimize the way that the interpreter translates the frequently executed control flow paths into native code. Feedback-directed optimization schemes can also be used with more conventional systems. The Intel C compiler allows programs to be compiled so that they generate feedback about their runtime performance [5]. This data can be used by a subsequent compilation to improve performance through more accurate branch prediction and more efficient code layout.

Fernandez et al. used link-time editors with code path profile data in order to efficiently remove redundant loads [10]. Their system used information about hot code paths to remove the loads in the most frequently executed basic blocks at the expense of adding register moves to less frequently executed blocks. The register moves are used to copy the value into a dead register so that the value is present in a CPU register when the value is needed again. The primary contribution of this system is the idea that basic block execution frequency can be used to increase the efficiency of register allocation. The algorithm used by Fernandez et al. is similar to the redundant load removal optimization described in the next chapter of this thesis.
Chapter 3

RIO operation

The RIO system can be viewed as an IA32 (Intel x86) instruction set interpreter which itself runs natively on the IA32 platform. RIO runs Linux and Windows NT/2000 applications. Rather than executing an application directly, RIO copies, and perhaps modifies, parts of the application binary into its code cache. The code in the cache is then executed directly. Whenever a block of code is copied into the cache, the target addresses of any branches are modified to return control to RIO, rather than jumping to original application code. RIO contains instrumentation to monitor the application's execution. When "hot" traces are encountered, the code is transformed into a single, contiguous, straight-line piece of code. Traces are a sequence of frequently executed basic blocks with a single-entry point (it may have multiple exit points.) Traces' unique control flow structure can increase performance by allowing a processor's instruction decoder and branch predictor to work more efficiently. The other benefit of traces is that their single-entry, multiple-exit control flow creates opportunities for optimizations that are ordinarily not possible.

The rest of this chapter explains RIO's operation in more detail. First RIO's method for executing an application under its control is described. We then explain how application code is initially copied into the code cache. Then we discuss RIO's trace creation scheme. The chapter ends by explaining some of the issues relating to application performance under RIO.
3.1 Executing an application under RIO

The RIO injector is used to run an arbitrary application under RIO. The injector begins to execute an application via a call to its \texttt{dynamorio\_app\_start} routine. This routine is called after the application’s code is loaded into memory, but before the application’s code begins to execute. In Linux, the \texttt{LD\_PRELOAD} mechanism is used to execute \texttt{dynamorio\_app\_start}. In Windows, the injector creates an instance of the application which begins in a suspended state. The injector then modifies that application’s code to load the RIO shared library and call the \texttt{dynamorio\_app\_start} routine.

3.2 Code fragment creation

RIO begins executing an application by copying the application’s first basic block into its code cache. RIO extends basic blocks to continue across direct branches and calls; they end at the first conditional or indirect branch they reach. The terminating jump in a basic block is changed so that it no longer points back to the original application. If a basic block corresponding to the destination PC is already present in the basic block cache, the target of that jump is changed to point to the top of the corresponding basic block in the cache. If there is no entry corresponding to the destination PC in the cache, then the jump’s target is changed to return control to RIO. RIO’s basic block builder is responsible for emitting a new basic block into its cache when no cache entry corresponds to the desired application instruction address. When the new basic block is copied into the cache, the original jump is changed so that it links to this new basic block directly.

Indirect branches require special treatment in the system. As described above, direct branches are handled by having their desired target changed to point to the correct destination in the code cache, rather than the original application. Indirect branches must be handled differently. RIO translates the desired target to the appropriate code cache entry. Each indirect branch is replaced with a jump to the indirect
branch lookup routine. This routine attempts to locate the cache entry that corresponds to the desired application PC. If a code cache entry does not already exist for the desired application PC, a new cache entry is created for that application PC. Either way, after the indirect branch lookup routine, execution continues in the code cache.

### 3.3 Trace creation

Traces offer RIO the potential to achieve higher performance than native execution since they organize a sequence of commonly sequentially executed basic blocks into a single contiguous piece of code. Organizing the code into this form allows the trace to use the processor’s instruction cache and branch prediction unit more efficiently, leading toward increased performance.

Traces are identified by first locating trace heads. Trace heads are basic blocks that are the target of a backwards branch (an indication they could be the top of a loop), or the target of branches that exit existing traces. Each trace head has a counter associated with it; whenever this counter exceeds a fixed threshold, RIO enters trace creation mode.

In trace creation mode, the trace head and subsequent basic blocks are copied into a trace in the code cache. Conditional branches between basic blocks in the trace are
changed so that if execution follows the same path as the trace, the branch will not be taken and execution will proceed to the next instruction in the trace. The targets of indirect branches are copied directly into the trace. A compare instruction ( cmp ) is inserted to check that the destination of the branch is the same at runtime as it was when the trace was created. If the cmp shows that the runtime target is different from the target when the trace was created, then the code branches to the indirect branch lookup routine.

3.4 RIO performance

As a result of the way that RIO rearranges the application’s code, indirect branches are one of the primary factors impacting application performance under RIO. Within a trace, overhead is incurred by the extra cmp instruction to check the destination of the indirect branch. This cmp instruction writes to the CPU’s eflags register, so if the application depends on the flags being preserved, additional code must be inserted in order to save and restore the eflags after the cmp. Even if execution stays on trace, the overhead of the cmp and code to save and restore the flags can be substantial. On applications that have a relatively large number of indirect branches, the execution time of these additional instructions can severely impact performance.

Indirect branches pose an additional performance problem for applications run under RIO. An indirect branch cannot be executed directly because it is necessary to locate the code cache element corresponding to the desired target, rather than branching to the application code directly. The indirect branch lookup routine is called whenever a basic block in the cache ends with an indirect branch, or whenever an indirect branch in a trace targets a different instruction address from the one which the trace assumed. The execution time of the indirect branch lookup routine is substantially longer than that of the single instruction it replaces. Depending on the frequency of indirect branches in the application, the overhead of executing this routine can account for a significant slowdown.

Typically, the most substantial slowdown when running an application under RIO
occurs when new code is copied into the code cache. For applications with relatively small amounts of code, this slowdown is limited to a short startup period. Once all of the application’s code is in the cache, slowdown due to code cache insertion is no longer an issue. Applications with large amounts of code can encounter new code over the entire course of the application’s execution. Each time new code is encountered, execution slows because the new code must be copied into the code cache. Despite attempts to minimize RIO’s overhead, code cache insertion overhead causes applications with large amounts of code to typically be poor candidates for good performance when run under RIO.
Chapter 4

Redundant load removal on traces

This chapter describes the redundant load removal (RLR) optimization. This optimization analyzes traces and removes unnecessary loads. This optimization was inspired by examining typical application hot traces by hand. Many of these traces contained loads that could be removed without affecting application correctness. In many cases, the redundant loads are due to inefficient compiler generated code. This chapter describes the algorithm that our system uses to locate and remove these redundant loads. The performance of this optimization is then analyzed using the SPEC CPU2000 benchmark suite.

4.1 Algorithm overview

Despite a compiler’s best attempts at register allocation, application binaries often load and then re-load the same values. These redundant loads can be removed without affecting correctness. This RLR optimization analyzes traces to locate situations where values that were recently spilled to memory are subsequently re-loaded into a register. If the memory location has not been modified before the second load, the RLR optimizer attempts to eliminate the load by finding a dead register in which to save the value.

The simplest example of a redundant load is the case when the value that is to
be loaded is still present in a CPU register when it is re-loaded from memory. In this case, instead of re-loading the value from memory, the second load can be transformed into a move from the old register to the new register:

Before RLR:

```asm
load eax, (at)
... ;code which uses but doesn't overwrite eax
load ebx, (at)
```

After RLR:

```asm
load eax, (at)
... ;code which uses but doesn't overwrite eax
mov ebx, eax
```

A register-to-register move usually executes faster than a load, so when this optimization can be made, performance typically increases. On the Intel IA32 architecture, a register-to-register `mov` typically executes in a single cycle (or even a half cycle on a Pentium 4), whereas a load takes at least 3 cycles even if the data of interest is in the L1 data cache [2].

The optimization opportunity described above is only applicable when the register used to save the value has not been overwritten by any intermediate computation. However, it is possible to modify the above scheme to work when the original register is overwritten, whenever there is an available register. The value can be copied to that register to save the value without resorting to reloading the value from memory. (A register is considered “available” at a point in the program if no subsequent instructions read its value, and if the next write to that register is after the second, redundant, load.) This modification of the scheme involves inserting an additional register-to-register move instruction; however performance is typically still greater than the original code containing the redundant load. Here is an example of this code transformation:
Before RLR:
load eax, (at)
mov ebx, $0
... ;code that reads and writes eax and ebx
load ebx, (al)
mov edx, $0
... ;code that requires (at) in ebx

After RLR:
load eax, (a1)
mov edx, eax
mov ebx, $0
... ;code that reads and writes eax and ebx
mov ebx, edx
mov edx, $0 ;edx is overwritten, so it is available
... ;code that requires (a1) in ebx

In this example, the register edx is available, so an extra register-to-register move is inserted by the original load to save the value in edx.

4.1.1 Correctness

One issue that must be addressed when optimizing application binaries is that any changes made to the code must guarantee correct execution. For the redundant load removal scheme to be safe, the value used to replace the redundant load must be identical to the value that would normally be loaded.

The most critical issue as far as ensuring correct execution is ensuring that no writes occur to the memory address between the initial load and the subsequent redundant load to the same address. If such a write occurs, then the second load cannot be removed (and should not be considered redundant!) To ensure correctness, this optimization limits itself to removing loads from the stack (as recognized through loads indirect from EBP—the base pointer). Within a trace, stack memory references can be unambiguously distinguished by looking at their displacements from the base pointer. A write to a displacement from EBP that has the same address as a memory access that is a candidate for redundant load removal will cause the optimization to abort. It is also possible for a register indirect store to overwrite a memory location on the stack if a register besides EBP is a pointer to a stack memory location. To
prevent this, any register indirect stores that use registers besides \texttt{EBP} will also cause the redundant load optimizer to abort and leave the redundant load intact.

The other issue that must be addressed to ensure that redundant loads are replaced in a safe manner is ensuring that 8 or 16 bit instructions not partially overwrite one of the registers storing the loaded value. On the \texttt{IA32} platform, the register \texttt{AX} is an alias to the lower 16 bits of the 32 bit register \texttt{EAX}. In performing dependency analysis, a write to \texttt{AX} will affect the value that is stored in \texttt{EAX}. Similarly, \texttt{AL} and \texttt{AH} alias to the lowest and second lowest bytes respectively in \texttt{EAX}. Similar sub-registers exist for most of the registers on the \texttt{IA32} architecture, and care must be taken to ensure that no writes to these registers modify the value being saved to replace the redundant load.

4.1.2 Comparison with previous schemes

Fernandez et al. studied redundant load removal schemes using a link-time editor \cite{10}. Their scheme involved using a link-time editor to optimize the application binary directly, rather than optimizing in-memory traces as is done in our system. Because their work directly modified the application, they encountered involving a load in a basic block being redundant across some paths to that basic block, but not others. Because of this, a significant portion of their work dealt with statistically analyzing the execution frequency of each basic block to ensure that any instructions inserted to remove loads actually improved performance. Performance could be hurt if a register-to-register move instruction was inserted in a basic block that was executed more often than the basic block that would have the load removed.

Our use of instruction traces has the effect of re-arranging the code in a different way than the link-time editors used by Fernandez. A trace duplicates code so that there may be multiple copies of the same basic block of code depending on the path taken to reach it. Each trace represents a single most frequently executed path through a series of basic blocks, so when a redundant load is found and removed in a trace, it is likely that the removal will increase performance. There is typically a
single hot path through a trace, so our scheme rarely hurts performance by inserting instruction in the wrong places.

### 4.2 Performance

The performance improvement that the redundant load removal optimization achieves varies depending on the particular application that is being run. The Spec CPU2000 benchmarks are used to analyze the performance of our system. Figure 4-1 shows the runtime performance of the optimization. The figure shows the slowdown of RIO with RLR and also unoptimized RIO compared to native execution time.

As seen in Figure 4-1, RIO performs better with the RLR optimization on a number of benchmarks than RIO without any optimizations. RIO with the RLR optimization achieves a 26% speedup vs. native execution time on the mgrid benchmark; other applications that were sped-up include apsi, applu and wupwise. Additionally, the
RLR optimization improves performance on several other benchmarks as well. Some benchmarks have their performance hurt slightly by the RLR optimization, this could be attributable to the RLR optimization inserting additional mov instructions that get executed more frequently than the redundant load that was removed.

The performance of our system compares favorably with Fernandez's work [10]. The average speedup that our system achieves is about 7.5% (with a maximum speedup of 26% for mgrid), Fernandez achieved speedups of between 3% and 8%. These numbers should not be compared in detail because the manner in which Fernandez's work was performance tested was very different from the performance testing done for RIO with RLR. Nonetheless, the general trend in these numbers indicate that the performance of RIO with the RLR optimization achieves better speedups than the load removal scheme that Fernandez describes.

Figure 4-2 shows the fraction of total loads in each application that are redundant. Typically, around 2% of the total loads in each benchmark are ripe for removal. Some
benchmarks, however, contain a much higher percentage of redundant loads. The benchmarks that had the largest fraction of their loads removed tend to be the ones that achieve the greatest performance improvement from the RLR optimization, but this is not true in all cases. The most important factor in increasing performance is removing loads in the most frequently executed traces; trace execution frequency is ignored in this graph. Nonetheless, the percent of loads that are redundant appears to be a good indicator of overall application speedup from redundant load optimization.

Figure 4-3 shows the average number of basic blocks that are traversed for every redundant load that is removed. This data is interesting because removing a load that is redundant across basic blocks is a difficult optimization to make at compile-time. Integer-based applications (the benchmarks ranging from bzip2 to vpr on the right half of the graph) tend to have more basic block traversals among their removed loads. The integer benchmarks tend to have shorter basic blocks than the floating point benchmarks, this leads toward more redundant loads spanning basic blocks.
Chapter 5

Eliminating runtime constants

This chapter describes the load to constant (LTC) optimization. This optimization was inspired by the observation that, in many situations, the same memory address is read without an intervening update. If such memory addresses are found to have relatively few values over the course of the application’s execution, they can be considered runtime constants. Specializing a trace for the runtime constants the it accesses allows a number of code optimizations to be performed. This value specialization allows loads to be removed from the trace; the entire trace gains the performance advantages of constant propagation and arithmetic simplification. Additionally, depending on the particular way that the trace uses these values, specialization often allows the removal of cmps and their dependent conditional jump instructions.

As described in Section 2.3, a number of other system for performing dynamic optimization have addressed the issue of runtime constants. Typically these systems involve the use of special compilers that generate binaries capable of generating value specialized code at runtime. Applications are developed with special languages (or annotations on top of existing languages) that specify the variables that should be considered runtime constants. This approach is burdensome for the programmer, inflexible, and potentially inaccurate. Performance relies on the programmer remembering to annotate runtime constants, if the programmer doesn’t annotate a runtime constant, performance is not improved. However, the added complexity of these systems allows substantial speedup: DyC [11] achieves speedups of up to 4.6x for some
micro-benchmarks. Because this approach requires that an application be written in a special way, these schemes only useful for improving the performance of new applications (or at least applications where source code is available). It cannot be easily applied to existing applications.

The LTC optimization we describe is capable of detecting runtime constants without any special effort on behalf of the programmer. It can optimize applications written in any language because it is built within the RIO system. This chapter explains this system's method for identifying runtime constants. Then the creation of specialized traces is discussed. Finally, the performance of our scheme is analyzed.

5.1 Algorithm overview

The load to constant optimization analyzes each trace to see if it contains memory references that are read but not written. This is a preliminary test of whether a memory address is a runtime constant. If it passes this test, code is inserted to profile this value each time the trace is executed. Once a sufficient number of samples are collected, the value profile is analyzed to determine if the memory value can be considered a runtime constant. If so, a new version of the trace is created; this new trace is specialized for the values of any runtime constants accessed by that trace. Constant propagation and arithmetic simplification optimizations are performed on the specialized trace. When the trace is executed, code checks that each of the runtime constants have the expected value. If not, an unmodified copy of the original trace is executed. Overall performance increases when the speedup from executing the specialized trace is greater than any overhead of determining if the specialized trace can be safely executed.
5.2 Identifying runtime constants

5.2.1 Static trace analysis

The first step in identifying runtime constants is a static analysis of the instructions in a trace. The system analyzes every memory address that is read inside a trace. The load to constant optimization only considers traces that contain either a self-loop or at least 5 instructions that read a runtime constant candidate. When a memory address is register indirect, the memory reference is only considered a runtime constant if the effective address of the memory reference does not change while the trace is executing. To ensure that this address is constant, the registers in a register indirect memory reference cannot be written to by any code in the trace. Most register indirect memory references are stack references that are specified by a constant displacement off the base pointer (EBP). Writes to the base pointer are relatively rare, so this problem of a trace modifying the effective address of a register indirect memory access is infrequently encountered.

5.2.2 Value profiling

After statically analyzing the trace to determine which, if any, memory references are runtime constant candidates, the trace is instrumented to gather value profiles of each runtime constant candidate. The value profiling scheme is simple compared with previous schemes[7, 22], but a more complex system is unnecessary for our purposes. RIO’s ability to dynamically generate and modify code allows the value profiling code to be removed once sufficient value samples have been collected. Traditional systems leave the value profiling code in the application throughout its entire duration, so for these systems efficient value profiling code is essential to maintain reasonable performance.

A trace is instrumented for value profiling by inserting a call to the value profiling function at the top of the trace. The value profiling function takes the values of the CPU’s registers as arguments so that it can determine the effective address of any
register indirect memory references. The value of all the runtime constant candidates in that trace are recorded and saved. Typically, after the memory values have been collected, the value profiling function returns and the trace’s execution continues as normal. Any self-loops in the trace have their target changed so that any self-loops to the top of the trace instead jump to the instruction right after the call to the profiling function. This is important because a trace that contains a self-loop would quickly fill up the value profile buffer with identical values. The purpose of this value profiling function is to collect the values of runtime constants across many different calls to the trace.

The value profiling code keeps a counter of the number of values that have been collected. Once the value profile is complete, the system is ready to analyze the values to locate runtime constants. If one or more runtime constant was found in the trace’s value profile, the trace should be replaced with a trace specialized for the runtime constants’ values. If no runtime constants are present in that trace, then the trace will not be modified substantially, but it does to have its call to the value profiler removed. When value profiling is complete, the value profiling function returns true—it normally returns false. Special code in the trace catches this return value and returns control to RIO. A special hook in RIO catches this special exit, analyzes the collected values, and modifies the trace accordingly. Depending on the result of the value profiling, RIO replaces the trace with a special trace that contains code specialized for the runtime constants, or else the original trace (without the call to the value profiler).

5.2.3 Analyzing value profiles
This section discusses the issue of analyzing value profiles and deciding which memory addresses should be considered runtime constants. For every runtime constant in a trace, an extra cmp and jz instruction is needed at the top of the trace to determine if the specialized version of the trace can be safely executed. If the runtime constant rarely has the value for which it was specialized, the specialized version of the trace
will rarely be executed. Without the speedup from executing the specialized trace, the overhead of the additional \texttt{cmp} and \texttt{jz} instructions could become substantial.

The system considers runtime constants to be memory addresses that are found to have the same value for at least 80% of the value profile. See Further Work (section 5.5.1) for more details on other schemes for identifying runtime constants from their value profiles.

### 5.3 Specialization and optimization

One of the biggest challenges of implementing a binary runtime optimization system is that instruction level optimizations must be performed carefully to preserve correct execution. Most existing runtime optimization systems perform at least some of their code analyses at the level of source code. Source-level analyses allow optimizations to make more assumptions about the application’s control flow and data usage than is possible with instruction-level analysis [11, 16]. When performing instruction-level optimization, any changes made to the original application’s code must be done so that the changes do not modify any machine state in a way that could affect any other instruction’s execution. Because analyzing raw application instructions (without access to source code) gives minimal data about invariants within an application, optimizations must err on the side of caution to ensure execution correctness. For example, the arithmetic simplification pass cannot remove \texttt{add} instructions if the possibility exists that a subsequent block of code might depend on the \texttt{add}’s write to the processor’s carry flag.

#### 5.3.1 Structure of an optimized trace

When trace specialization takes place, the original application trace is converted to a new piece of code that contains an identical copy of the original trace (the “safe” trace), a specialized version of the original trace, and a sequence of compares to decide which trace is to be executed. If all the addresses are identical to the values for which they were specialized, then execution proceeds to the specialized version.
of the original code. If any of the memory addresses do not match their specialized values, then to ensure correct execution, the processor jumps to the safe trace. The \texttt{cmp} instruction(s) write to the processor’s \texttt{eflags} registers, so if the trace depends on the previous values of these, it is necessary to save and restore the flag’s values before and after the compares. Figure 5-1 shows an example trace with these different components labeled.

5.3.2 Replacing memory references with constants

The specialized trace is a modified copy of the original trace where our optimizer makes all attempts to change reads to runtime constants into immediate constants hard-coded directly into the instruction. The simplest optimization made is changing loads (\texttt{mov} instructions with a memory source) into immediate stores (\texttt{mov’s} of immediate constants into the same destination as the original instruction(\texttt{register} or \texttt{memory}). The optimization of other instructions can be more complicated.

IA-32 allows a wide variety of different instructions to read and write data memory. The optimization attempts to replace any instruction that reads a runtime constant with an equivalent instruction that instead references an immediate constant. The system’s ability to perform such replacements is often limited by the valid instructions in the IA-32 instruction set.\footnote{Many IA32 instructions, like \texttt{add}, can operate on register and memory operands, but also can reference immediate constants. However, a number of instructions including \texttt{div} and \texttt{mul} do not have valid encodings that refer to immediate constants. Instructions that do not have encodings for referencing immediate constants cannot have their memory reference removed. They are unchanged by our optimization.} For example, if the address $a$ is replaced by the constant 0, then this instruction:

\begin{verbatim}
add a, eax -> eax
\end{verbatim}

can be safely transformed to:

\begin{verbatim}
add 0, eax -> eax
\end{verbatim}

A simple arithmetic simplification pass is then performed on the new instruction. In this example, adding 0 to \texttt{eax} leaves \texttt{eax} unchanged. This \texttt{add} is effectively a \texttt{nop} and can be safely removed. The only potential issue with removing the instruction
Figure 5-1: This shows an example of the structure of a trace that contains a version of the trace specialized for a runtime constant. This trace is specialized for the runtime constant at address 0x8077030 to have the value $0x080bf038. The specialized version of the trace is located on lines 3 to 11; a copy of the original (unspecialized) trace is located on lines 12 to 22. The beginning of the trace (lines 1 and 2) checks that the runtime constant has the value for which it is specialized. If it has a different value, then it jumps to the original trace at 0x40200482 (line 12). If the runtime constant has the expected value, then the specialized trace begins executing (line 3). The conditional jumps on lines 10 and 21 are self-loops that should target the top of the trace. Notice how the system modifies the targets of any self-loops in the specialized or original trace so that these self-loops point the top of their sub-trace. The trace-choosing code is not executed on each loop iteration.
Figure 5-2: When the operands of a cmp are transposed, conditional jumps that depend on the cmp’s result must be changed appropriately. The left column of this table shows the original instruction; the right column shows what instruction it must be changed to if the cmp it depends on has its operands transposed.

<table>
<thead>
<tr>
<th>Original Instruction</th>
<th>Transposed Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>jle (jump if less than or equal)</td>
<td>jge (jump if greater than or equal)</td>
</tr>
<tr>
<td>jge (jump if greater than or equal)</td>
<td>jle (jump if less than or equal)</td>
</tr>
<tr>
<td>jl (jump if less than)</td>
<td>jg (jump if greater than)</td>
</tr>
<tr>
<td>jb (jump if below)</td>
<td>ja (jump if above)</td>
</tr>
<tr>
<td>jae (jump if above or equal)</td>
<td>jbe (jump if below or equal)</td>
</tr>
<tr>
<td>jbe (jump if below or equal)</td>
<td>jae (jump if above or equal)</td>
</tr>
<tr>
<td>ja (jump if above)</td>
<td>jb (jump if below)</td>
</tr>
</tbody>
</table>

is that it is possible that subsequent instructions depend on this instruction’s write to the eflags register. The instruction can only be removed if the side-effects of the instruction are not necessary for correct execution. However, even if an eflags dependency prevents this instruction from being removed, performance is still increased because the instruction no longer needs to complete a read from memory.

**Replacing memory references in cmp’s**

Replacing memory references with immediates in cmp instructions is more complicated than with other instructions. The IA-32 instruction set defines the cmp instruction so that if it is comparing a register or memory value with an immediate, the immediate must be the second operand. Many situations arise where the memory reference that is to be replaced by an immediate is the cmp’s first operand. The only way to correctly encode a cmp instruction that uses this immediate is to swap the order of the operands so that the immediate becomes the second operand.

Swapping the order of a cmp’s operands cannot always be done safely. Because a cmp’s result depends on the order of its operands, when a cmp’s operands are switched, instructions that use this result must be changed appropriately. The cmp’s result is stored in the eflags register, so it is necessary to examine all instructions that read the eflags. The most common instructions that read the eflags after a cmp write are conditional jumps. Conditional jumps can be altered to accommodate a cmp’s transposed operands as shown in Figure 5-2.
The jz instruction (and several other types of conditional jumps) is frequently used in conjunction with a cmp, but this instruction is omitted from the above table because it does not depend on the ordering of the operands in the cmp and therefore does not need to be changed when an cmp’s operands are transposed.

A cmp can only have its operand ordering changed if we are able to modify all code that could be affected by the change. The transformations in Figure 5-2 allow code to be modified to work correctly with an altered cmp, but in some situations a branch targets code in a different fragment which could read the cmp’s write to the eflags register. The instructions in this trace cannot be safely modified to account for the cmp’s swapped operands, so the optimization cannot be safely performed.

An additional complication arises if the original cmp compares a runtime constant with an immediate. If the runtime constant were replaced with an immediate, then the cmp would have two immediate operands. The result of such a compare can be determined when specialization occurs by analyzing the values of the constants. When such a situation is encountered, the cmp can be removed and all conditional jumps that depend on the cmp can be either removed or made into unconditional jumps. Once again, this optimization can only be performed if all instructions that read the cmp’s eflags write can be examined. If the application jumps to a piece of code that depends on the cmp’s write to the eflags register, and this instruction cannot be modified, then the cmp cannot be removed safely.
5.3.3 Constant propagation and arithmetic simplification

After a trace is specialized for its runtime constants, the resulting trace often contains
an excess of immediate-to-register mov instructions. These instructions can often be
removed or combined with other instructions by a constant propagation and arithmetic
simplification pass over the specialized traces. Here is an example instruction
sequence from the SPEC2000 CPU twolf benchmark[20]. This trace has already had
a number of runtime constants replaced by immediate stores, so it contains a number
of immediate stores:

1. mov $0x0819b368 -> %eax
2. mov (%eax,%edx,4) -> %eax
3. mov $0xfc18 -> 0x2(%eax)
4. mov $0x0819b368 -> %eax
5. mov (%eax,%edx,4) -> %eax
6. mov $0xfffe -> 0x4(%eax)
7. mov $0x00000010 -> %eax
8. inc %edx -> %edx
9. inc %eax -> %eax
10. cmp %edx %eax
11. jie $0x804f940
12. jmp $0x804f967

Here is the same code after constant propagation and arithmetic simplification
have been performed. The constant 0x0819b368 is the memory address of a data
structure in memory. This value is determined at runtime so this specialization can
not be performed by a compiler.

13. mov 0x819b368(,%edx,4) -> %eax
14. mov $0xfc18 -> 0x2(%eax)
15. mov 0x819b368(,%edx,4) -> %eax
16. mov $0xfffe -> 0x4(%eax)
17. inc %edx -> %edx
18. mov $0x00000011 -> %eax
19. cmp %edx $0x00000011
20. jle $0x804f940
21. jmp $0x804f967

In this situation, three instructions were able to be removed from the trace: lines
1 and 2 were combined to create line 13, lines 4 and 5 were combined to form line
15, and lines 7 and 9 were merged to create line 18. (Additionally, the cmp in line 19 was able to use the constant 0x11 rather than relying on the register eax.) With modern superscalar processors, instruction removal does not guarantee performance improvement, but the removal of 25% of the instructions in this trace should lead to some sort of performance improvement. This example is only a small part of the twolf benchmark, but nonetheless the example shows the potential benefit of performing constant propagation and arithmetic simplification on traces specialized for runtime constants.

5.4 Performance

Figure 5-3 shows the slowdown of RIO with our LTC optimization and unoptimized RIO relative to native execution time. The performance of the LTC optimization varies substantially across different benchmarks: on the art and mcf benchmarks, RIO with LTC achieves a noticeable performance improvement over unoptimized RIO execution. On other benchmarks, the LTC optimization doesn’t perform as well, even incurring noticeable slowdown vs. unoptimized RIO execution on some benchmarks.

Figure 5-4 shows the fraction of runtime constants that our system found among all the memory references made by traces. Although this graph shows that some benchmarks (art and bzip2, particularly) exhibit over 10% of their memory references to runtime constants, the information in this graph should not be used by itself as an indicator of application performance. To accurately gauge the performance that comes from locating runtime constants, it is necessary to examine whether the runtime constants maintain their value over the entire length of the application’s execution. If an application changes phases, and a single runtime constant in that trace changes its value, then the specialized trace can no longer be executed and no speedup occurs.

Figure 5-5 shows how often the specialized version of traces can safely be executed (ie. how often the runtime constants have the expected value). Runtime constant value profiling only occurs during the application’s startup period; when a phase change occurs, the runtime constant’s value often changes. Changes in the value of
a runtime constant prevent the specialized trace from being executed. The data in
this graph shows that a number of the benchmarks maintain high “hit-rates” for their
specialized traces—indicating that runtime constants maintain their value. However,
the runtime constant optimization hurts performance when running applications that
infrequently execute the specialized traces. An ideal implementation of a runtime
constant optimizer would maintain a high specialized trace “hit-rate” for all appli-
cations by periodically re-analyzing the values of all runtime constants. See Section
5.5.2 for more information.

5.5 Additional work

The LTC optimization presented here is an early implementation, but it shows the
potential of the system. There are a number of ways that the base system itself could
be improved. Areas with potential for improvement in the system include: Improving
identification of runtime constants, and ensuring that traces are specialized for the correct values for the runtime constants.

5.5.1 Identifying runtime constants

Currently the system's scheme for identifying runtime constants is fairly simplistic: Every trace is instrumented to gather a value profile for any runtime constants in the trace (currently the number of collected values is fixed set at 40). Once these values are collected, the address is considered a runtime constant if the number of identical values in the profile exceeds a certain threshold (currently this threshold is set at 80% of the collected values.) If the value is indeed a runtime constant, then the trace is specialized for the collected value. This scheme is relatively simple and can miss a number of situations where it could be productive to specialize the trace.

The ideal implementation of a runtime constant identification system would spe-
Specialize a trace for any values of the runtime constant that would produce a speedup. Such an implementation would allow a single trace to be specialized for multiple values of a runtime constant, and also allow different combinations of values for different runtime constants. Also, the fixed threshold that the system uses for determining runtime constants from value profiles is overly simplistic. The correct threshold for when a variable should be considered a runtime constant should depend on the potential speedup that is possible from specializing the trace for that value. If the potential speedup is large, then the system should be more aggressive about specializing the trace.

5.5.2 Changing specialized values over time

In the current implementation, once a runtime constant is found, the value that it is specialized for is unchanged throughout the application’s entire execution. It is
possible that a runtime constant actually has a few values that change over the course of the program's execution (perhaps as the application enters different phases). A better implementation could monitor the accuracy of each specialized value (the LTC optimization infrastructure already supports counters for whether the specialized or safe version of each trace is executed). Whenever the specialized "hit rate" falls below a certain threshold (perhaps indicating a phase change in the application) the system could go back into value profiling mode. Once in value profiling mode, a new value can be found for the same memory address. If the value profiling shows that the memory reference is no longer a runtime constant, the specialized trace should be replaced by the original trace.
Chapter 6

Conclusion

This thesis describes a series of optimizations within the RIO runtime optimization system for improving application memory performance. The primary contribution of this work is the description and implementation of optimizations that can be automatically performed on unmodified IA-32 binaries. Existing systems are capable of similar optimizations, but they typically require the programmer to implement the application with special tools. The system described in this thesis performs optimizations automatically; this can save substantial development time. Because our system operates at runtime on native binaries, it puts no limitations on the manner in which an application can be developed. The system can optimize applications when source code is not available. An additional benefit of optimizing at runtime is that our system can optimize dynamically loaded libraries that an application may access; this is impossible with conventional compile-time optimization.

The redundant load optimization (see Chapter 4) improves application performance by finding and removing redundant loads in traces. The optimization also removes excessive register spills within basic blocks due to inefficient compiler-generated code. Although the loads that the optimization removes typically are accesses to addresses still in the processor’s L1 cache, performance is improved because executing a load from the L1 cache take longer than register-to-register mov instructions. This optimization’s performance on the SPEC CPU2000 benchmarks show substantial improvement and demonstrate the usefulness of such a technique. The optimization’s
performance for removing redundant loads within basic blocks is evidence that compilers could achieve substantial performance improvement by improving their register allocation within basic blocks. Our results indicate that execution performance could be improved by running a load-removal algorithm on individual basic blocks at compile time.

The runtime constant optimization described in this thesis (see Chapter 5) allows the automatic generation of specialized code for the values of runtime constants. There are no existing tools that allow this optimization to be performed on unmodified binaries. Existing schemes require either value profiling runs and additional recompilations, or special compilers that generate dynamic code from with special annotated source code. We showed examples of code where this optimization can achieve substantial speedups. However, the code is still fairly immature and a number of hurdles need to be overcome in order to show performance improvement on large applications. The major shortcoming of our implementation is that our system does not adapt its specialized traces to runtime constants that change their values as the program enters different phases. However, even with this shortcoming, our implementation demonstrates that runtime code specialization is a viable technique for optimizing application performance.

One of the biggest difficulties encountered in this work is the difficulty of optimizing application binaries while maintaining execution correctness. One of the advantages of schemes that specify runtime constants through source-level annotations is that specialization can be facilitated by special compiler-generated code. Our system only locates runtime constants at the level of traces, with some sort of compiler support for runtime constant identification, any access to a runtime constants could easily be identified. Compiler-supported dynamic code generation (like in DyC [11]) is able to eliminate much of the complex instruction-level modification that our system must perform. Compilers have much more knowledge about application control flow than can be determined from instruction-level analysis. This information can be put to use when performing runtime constant specialization. Compiler support simplifies the engineering of the system and could lead to additional opportunities.
for specialization. A system that supports runtime constant specialization at the source-code level can be much more aggressive about finding and removing runtime constants. Instruction-level specialization must be conservative, any register indirect write (except for displacements from EBP) could change the value of a runtime constant. Source-level analysis allows all writes to memory to be disambiguated, and the value of runtime constants can be known with greater accuracy. Perhaps the ideal use of runtime specialization is a system such as RIO along with some compiler help for getting additional information about application control flow.
Bibliography


