

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
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 arithmetic simplification performed on it. Benchmark results are shown; the optimizations show substantial improvement on a certain category of benchmarks.

Thesis Supervisor: Larry Rudolph
Title: Research Scientist

Acknowledgments

A number of people were instrumental in helping me pull together my thesis. Larry Rudolph's and Saman Amarasinghe's advice and guidance was immeasurable. Derek Bruening's work in creating RIO is stupendous; he deserves more thanks than a few words (and MIT's stipend) can give him.

My peers here at MIT's Lab for Computer Science have been very helpful in helping me with my work: Shalini Agarwal, Todd Amicon, Amay Champaneria, Prabhat Jain, Vinson Lee, and Ed Suh. Thanks to Barbara Cutler for making that nifty flowchart that helps to elucidate RIO's operation.

A bunch of other people have been instrumental in keeping me sane while working on this 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 online fantasy baseball game, Trey Gunn, Tony Levin, Frank Zappa, and Trent Reznor.

Contents

1	Introduction	9
2	Related Work	13
2.1	Runtime Systems	13
2.2	Dynamic Hardware	14
2.3	Languages for runtime specialization	14
2.4	Runtime systems for runtime specialization	15
2.5	Feedback Directed Optimization	15
3	RIO Operation	17
3.1	Startup	17
3.2	Code Fragment Creation	18
3.3	Trace Creation	19
3.4	RIO Performance	20
4	Redundant load removal on traces	23
4.1	Algorithm overview	23
4.1.1	Correctness	25
4.1.2	Comparison with previous schemes	26
4.2	Performance	27
5	Eliminating cuntime constants	29
5.1	Algorithm overview	29
5.2	Runtime Constant Identification	30

5.2.1	Static trace analysis	30
5.2.2	Value profiling	31
5.2.3	Analyzing value profiles	32
5.3	Specialization and Optimization	33
5.3.1	Structure of an optimized trace	34
5.3.2	Replacing memory references with constants	34
5.3.3	Constant Propagation and Arithmetic simplification	38
5.4	Performance	39
5.5	Additional Work	41
5.5.1	Identifying runtime constants	41
5.5.2	Changing specialized values over time	42
6	Conclusion	43
	Bibliography	45

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 the compile-time optimization. Runtime optimization systems can monitor an application’s typical control flow pattern. This information can be used to modify the application’s code layout to increase the performance of its most frequent operations. Traditional compilers generating static code do not have information about an application’s execution, the compiler can only make guesses about the application’s 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 treats dynamically linked libraries identically to static-code, and such optimizations can be safely performed. 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 profiling information when run. This 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 then 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 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]. The RIO system 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 presents the potential for unique optimizations which cannot be performed on static binaries.

The first optimization scheme that this thesis presents is a redundant load removal optimization. This scheme builds on Fernande'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 which are runtime constants, and specializing traces for these values.

The primary contribution of this thesis is that we show 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 otherwise written with special annotations to facilitate runtime optimization. Our work shows that execution performance of native executables can be improved by runtime optimization. We demonstrate our system by improving the performance of unmodified applications from the SPEC2000 CPU benchmark suite.

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 described 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 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 additional code optimizations upon the traces.

Microsoft's Mojo [8] is a dynamic optimization system for IA32 applications run-

ning 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, code is generated that is specialized 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 which 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 with 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 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 having a separate thread to perform optimizations using idle processing cycles has been implemented within RIO.

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 runs as a system-wide process, rather than each application having its own optimizer. Both Kistler and Franz's work and Jalapeno operate on 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 run in a profiling mode, which collects data about variable values, and identify 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, they integrated a control flow profiler 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 by the end user as an IA32 instruction set interpreter which itself runs natively on the IA32 platform. RIO runs application binaries by executing copies of their original code out of its code cache. 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 copied into the code cache and 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.

3.1 Startup

The RIO *injector* is used to run an arbitrary application under RIO. The injector begins to execute an application via a call to its `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 `LD_PRELOAD` mechanism is used

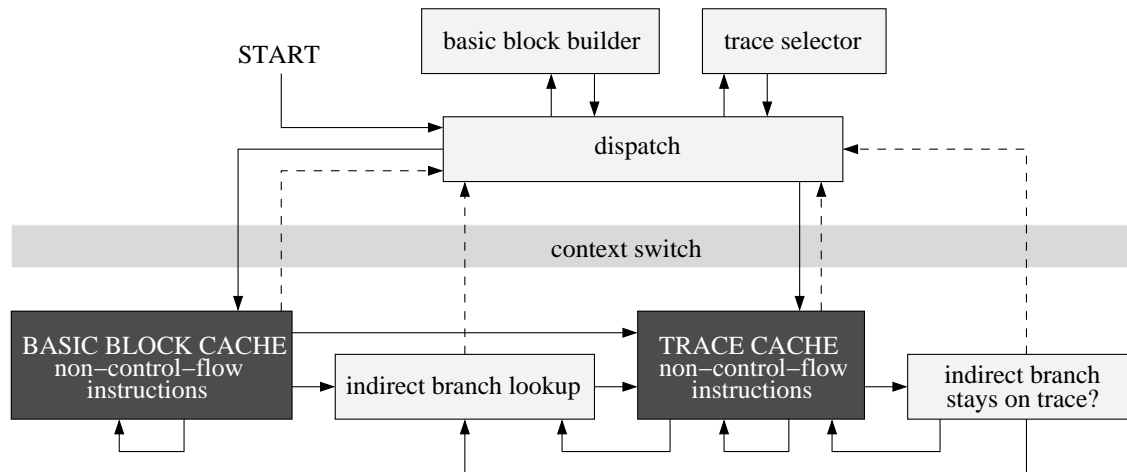


Figure 3-1: Flowchart of RIO Execution

to execute `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 `dynamorio_app_start` routine.

3.2 Code Fragment Creation

RIO begins executing an application by copying the application’s first basic block into its basic block 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 code cache. If there is no fragment corresponding to the destination PC in the fragment 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 because RIO must translate the desired target to execute code out of the code cache rather than code out of the original application. To handle this situation, an indirect branch is replaced with a jump to the indirect branch lookup routine. This routine attempts to locate the basic block or trace in the cache that corresponds to the desired application PC. If a code cache entry does not already exist for the desired application PC, a new basic block cache entry is created. Either way, after the indirect branch lookup routine, execution continues at a code cache entry that corresponds to the target instruction of the indirect jump.

3.3 Trace Creation

The creation of traces gives RIO the potential to achieve better performance than native execution. Traces 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 new trace in the trace cache. Conditional branches between basic blocks in the trace are changed so that if execution follows the same path as the trace, the jump 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 `cmp` instruction 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 flags after the `cmp`. 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 application's 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 executing the application code directly. This function is performed by the indirect branch lookup routine. This 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

4.1 Algorithm overview

Despite a compiler's best attempts at register allocation, application binaries often load and then re-load the same values. The redundant load removal (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 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:

```
load eax, (a1)
... ;code which uses but doesn't overwrite eax
load ebx, (a1)
```

After RLR:

```
load eax, (a1)
... ;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. If there is an available register, the 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 that contained the redundant load. Here is an example of this code transformation:

Before RLR:

```
load eax, (a1)
mov ebx, $0
... ;code that reads and writes eax and ebx
load ebx, (a1)
mov edx, $0
... ;code that requires (a1) 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 location between the initial load and the subsequent redundant load to the same address. If such a write occurs, then the second load can not 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 through `EBP`—the base pointer). Within a trace, stack memory references can be unambiguously distinguished by looking at their displacements off of the base pointer. A write to a displacement off of `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 `EBP` will also cause the redundant load optimizer to abort and leave the redundant load untouched.

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 IA32 platform that RIO runs on, the

register `AX` is an alias to the lower 16 bits of the 32 bit register `EAX`. In performing dependancy analysis, a write to `AX` will affect the value that is stored in `EAX`. Similarly, `AL` and `AH` alias to the lowest and second lowest bytes respectively in `EAX`. Similar sub-registers exist for most of the registers on the 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 [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 this scheme. Because their work directly modified the application, issues arose where loads in a given basic block could be 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's work. 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.

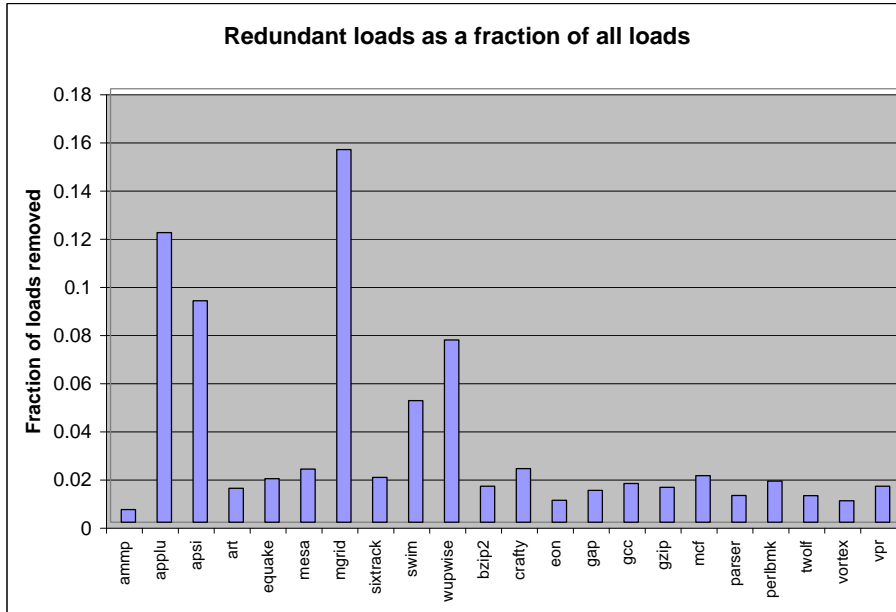


Figure 4-1: Redundant loads removed as a fraction of all loads seen in the application.

4.2 Performance

The performance improvement that the redundant load removal optimization achieves varies depending on the particular application that is being run. The Spec2000 CPU benchmarks are used to analyze the performance of our system.

Figure 4-1 shows the fraction of total loads in the application that were redundant and could be successfully removed. Typically, around 2% of the total loads in each benchmark could be removed. Some benchmarks, however, contain a much higher percentage of redundant loads. The benchmarks that had a larger fraction of their removed tend to be the ones that achieve the greatest performance improvement under this 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; the execution frequency of the trace containing each removed load is ignored in this

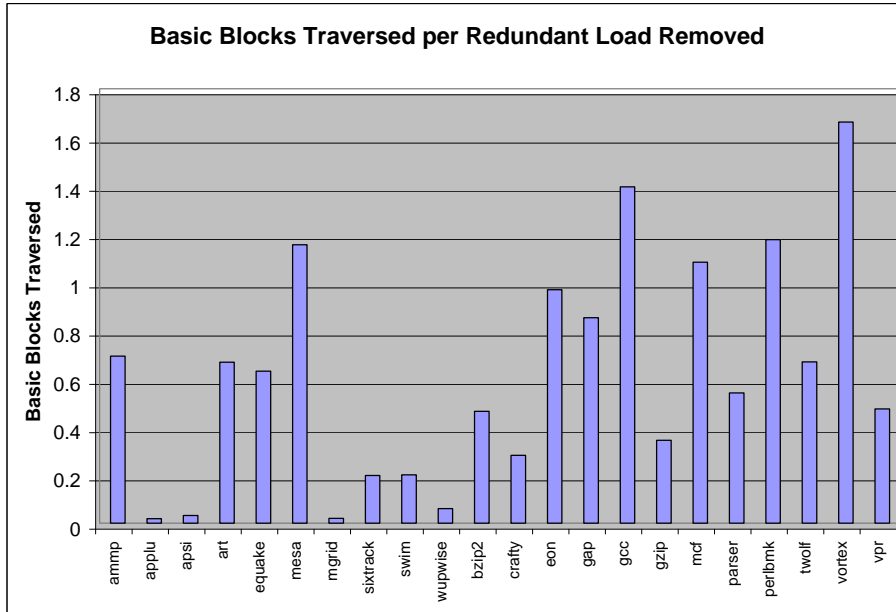


Figure 4-2: Average number of basic blocks traversed per redundant load removed.

graph. Nonetheless, the percent of loads that were able to be removed is a decent indicator towards overall application speedup with the redundant load optimization.

Figure 4-2 shows the average number of basic blocks that are traversed for every redundant load that is removed. This data is interesting because removing a redundant load spanning a basic block is a difficult optimization to make at compile-time. Integer-based applications (the benchmarks 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. *JOSH: add more about performance here*

Insert graph showing RLR performance vs. native execution, and also RLR performance vs. unoptimized dynamo.

Chapter 5

Eliminating runtime constants

5.1 Algorithm overview

The load to constant optimization (LTC) was inspired by the observation that many applications repeatedly read the same memory addresses without writing to them. If these memory addresses have relatively few values through over the course of the application's execution, they can be considered *runtime constants*. Specializing the trace for the value of the runtime constants in that trace allows a number of code optimizations to be performed on that trace. 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` their dependant 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) which specify the variables which 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. A system capable of automatically finding and optimizing runtime constants would be more flexible and useful. Because this approach requires that an application be written in a special way, it is only helpful for improving the performance of new applications. 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. The load to constant optimizer that we describe works by analyzing each trace to see if it contains memory references which are read but not written by that trace, this is a preliminary test of whether a memory address is a runtime constant. If this address is not written to, then code is inserted to log the value of this address each time the trace is executed. Once a sufficient number of samples are collected, the values profile is analyzed to determine if the memory value can be considered a runtime constant. If the trace contains any runtime constants, a new version of the trace is created which is specialized for the values of these runtime constants. Constant propagation and arithmetic simplification optimizations are performed on the specialized trace. When the trace is executed, a check is done to see if each of the runtime constants have the expected value. If the runtime constants have the value they were specialized for, then the specialized trace can be executed safely, otherwise an unmodified copy of the original trace is executed. The system is able to improve performance 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 Runtime Constant Identification

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.

Memory addresses which are read but not written by the trace are the runtime constant candidates. Our system only considers traces which contain a self-loop, or at least 5 instructions which read a runtime constant candidate. Traces which do not loop and only contain a few references to runtime constant candidates are unlikely to achieve enough of a performance gain from specialization to offset the overhead of choosing between the specialized and “safe” traces. When a register indirect memory reference is encountered, the memory reference can only be 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. Our value profiling scheme is simple compared with previous schemes[7, 22]. A more complex and efficient 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 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's values. If no runtime constants are present in that trace, then the trace will not be modified substantially, but it does need to be modified 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 the exit from this exit stub, 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 issues surrounding 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 at the top of the trace is needed to determine the specialized version of the trace can be safely executed. If the runtime constant rarely has the value that was found during the profiling, then the specialized version of the code will rarely be executed. Without the speedup from executing the specialized trace to offset it, the overhead of the additional `cmp` and `jz` instructions could become substantial. To achieve best performance, trace

specialization should only take place on those traces where the performance boost from executing the specialized trace is greater overhead of any additional instructions necessary to check the runtime constant's values.

The simplest (and safest) scheme for identifying runtime constants is to only specialize traces for runtime constants that value profiling only found a single value for. This should keep incorrectly identified runtime constants to a minimum, but it may also miss out on specialization opportunities. If a variable has the same value for 95% of the time, it is probably a good choice for specialization, yet this scheme would leave this value unspecialized. 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 optimizations 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. 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 it 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, to ensure execution correctness optimizations must err on the side of caution. For example, the arithmetic simplification pass cannot remove `add` instructions if the possibility exists that a subsequent block of code might depend on the `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 executed. The compares test the value of the specialized memory addresses. If all the addresses are identical to the values that they were specialized for, 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 `cmp` instruction(s) write to the processor's 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 of the original trace where our optimizer makes all attempts to change reads to runtime constants into immediate constants hardcoded directly into the instruction. The simplest optimization made is changing loads (`mov` instructions with a memory source) into immediate stores (`mov`'s of immediate constants into the same destination as the original instruction(register or 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.¹ For example, if the address a is replaced by the constant 0, then this instruction:

¹Many IA32 instructions, like `add`, can operate on register and memory operands, but also can reference immediate constants. However, a number of instructions including `div` and `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.

	Address	Instruction	
1	0x4020044b	cmp	0x8077030 \$0x080bf038
2	0x40200455	jnz	\$0x40200482
3	0x4020045b	mov	0x4(%ecx) -> %edx
4	0x4020045e	mov	%ecx -> 0x80bf038(,%edx,4)
5	0x40200465	mov	0x4(%ecx) -> %edx
6	0x40200468	mov	0x80bf038(,%edx,4) -> %eax
7	0x4020046f	mov	%di -> 0x20(%eax)
8	0x40200473	mov	(%ecx) -> %ecx
9	0x40200475	test	%ecx %ecx
10	0x40200477	jnz	\$0x4020045b
11	0x4020047d	jmp	\$0x402004ab
12	0x40200482	mov	0x4(%ecx) -> %edx
13	0x40200485	mov	0x8077030 -> %eax
14	0x4020048a	mov	%ecx -> (%eax,%edx,4)
15	0x4020048d	mov	0x4(%ecx) -> %edx
16	0x40200490	mov	0x8077030 -> %eax
17	0x40200495	mov	(%eax,%edx,4) -> %eax
18	0x40200498	mov	%di -> 0x20(%eax)
19	0x4020049c	mov	(%ecx) -> %ecx
20	0x4020049e	test	%ecx %ecx
21	0x402004a0	jnz	\$0x40200482
22	0x402004a6	jmp	\$0x402004ba

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 if located on lines 12 to 22. The beginning of the trace (lines 1 and 2) checks that the runtime constant has the value that it is specialized for, 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 line the specialized trace begins execution (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 their target so that they point the top of their sub-trace, so they avoid trace-choosing code on each loop iteration.

```
add a, eax -> eax
```

can be safely transformed to:

```
add 0, eax -> eax
```

A simple arithmetic simplification pass is then performed on the new instruction. In this example, adding 0 to `eax` leaves `eax` unchanged. This `add` is effectively a `nop` and can be safely removed. The only potential issue with removing the instruction 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 which 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 conditional jumps) is frequently used in conjunction with a `cmp`, but this instruction is omitted from the above table because

<code>jle</code> (jump if less than or equal)	→	<code>jge</code> (jump if greater than or equal)
<code>jge</code> (jump if greater than or equal)	→	<code>jle</code> (jump if less than or equal)
<code>jl</code> (jump if less than)	→	<code>jg</code> (jump if greater than)
<code>jb</code> (jump if below)	→	<code>ja</code> (jump if above)
<code>jae</code> (jump if above or equal)	→	<code>jbe</code> (jump if below or equal)
<code>jbe</code> (jump if below or equal)	→	<code>jae</code> (jump if above or equal)
<code>ja</code> (jump if above)	→	<code>jb</code> (jump if below)

Figure 5-2: When the operands of a `cmp` are transposed, conditional jumps which 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.

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 above allow code to be modified to work correctly with an altered `cmp`, but in some situations a jump targets code in a different fragment which reads the `cmp`'s write to the `eflags` register. This 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. In addition to being an invalid IA-32 instruction, the result of a compare of two constants can be determined when specialization occurs. When such a situation is encountered, the `cmp` can be removed and all conditional jumps which 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 which 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

When 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    $0xffffe -> 0x4(%eax)
7  mov    $0x00000010 -> %eax
8  inc    %edx -> %edx
9  inc    %eax -> %eax
10 cmp    %edx %eax
11 jle    $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    $0xffffe -> 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

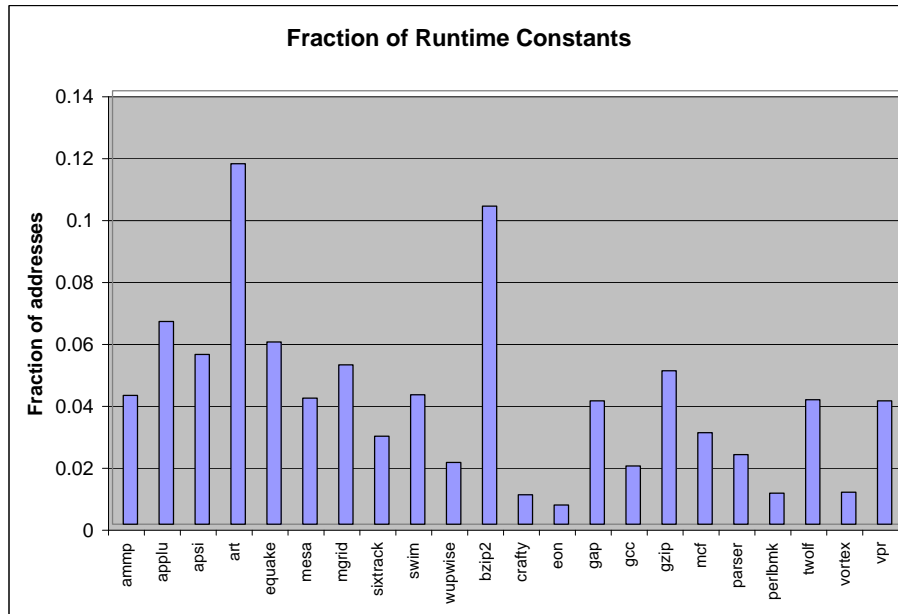


Figure 5-3: Fraction of runtime constants found among all memory references.

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 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

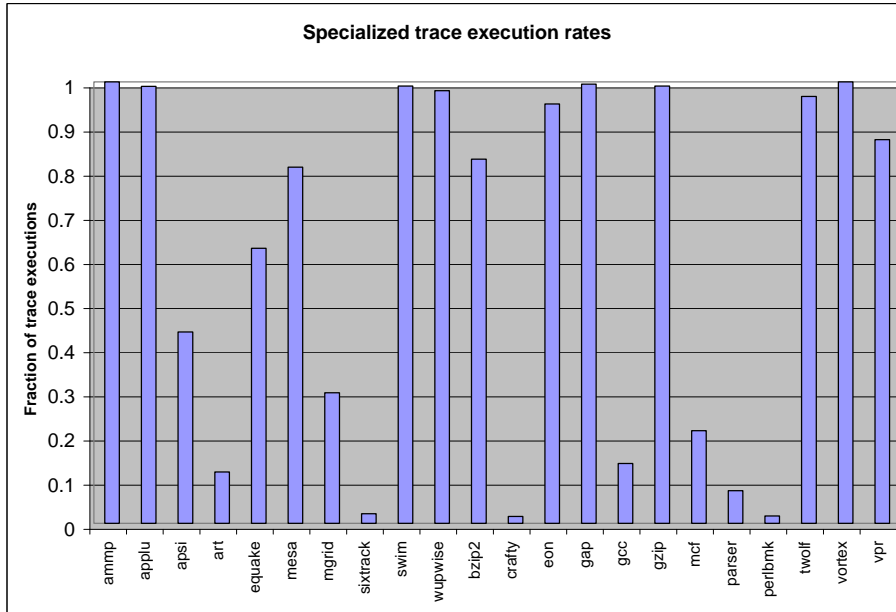


Figure 5-4: Specialized trace “hit-rate”..

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 phase change occurs 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 5-4 illustrates statistics about how often a trace that references a runtime constant can execute the specialized version of its trace. Runtime constant value profiling only occurs during the application’s startup period, so when a phase change occurs, the runtime constant’s value often changes. Changes in the value of a runtime constant slow execution because it prevents 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 the specialized traces. An ideal implementation of a runtime constant optimizer would maintain a high specialized trace “hit-rate” for all applications by periodically re-analyzing the values of all runtime constants. See Section 5.5.2 for more information.

Graphs still to include: LTC RIO execution time vs. native regular RIO execution time vs. native

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 runtime constant candidate in a trace is instrumented to save a fixed number of its values (currently the number of collected values is fixed set at 40). Once these values are collected, the address is considered a runtime-constant if every value collected is identical. 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 would be productive to specialize the trace.

The ideal implementation of a runtime constant identification system is that a trace should be specialized 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 constants, and also different combinations of values for different runtime constants. A runtime constant should also be identified even if the same value is not found in every sample—when a certain value occurs 95% of the time, the trace should be specialized for it. Determining the correct threshold for this value

is difficult, but certainly the value is some percentage below the 100% threshold that is currently used.

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 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 runtime constant. If the value profiling shows that the memory reference no longer appears as a runtime constant, a non-specialized trace can be executed instead—this would at least remove the overhead of checking the runtime constant's value each time the trace is executed.

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 that the optimizations we describe can be automatically performed on unmodified IA-32 binaries. Previous systems are capable of similar optimizations, but they typically require the programmer to implement the application in a special way. Our system performs its optimizations automatically, this saves development time for the programmer. 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 where 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 across basic blocks in hot traces and fixing up code that contains excessive register spills within basic blocks. Although the loads the optimization removes are likely accesses to values that are still in the processor's L1 cache, performance is still improved because loads from the L1 cache take longer than register-to-register `mov` instructions. This optimization's performance on the SPEC2000 CPU benchmarks show substantial improvement and demonstrate the usefulness of such a technique. The optimization's performance for removing redun-

dant loads within basic blocks is evidence that compilers could achieve substantial performance improvement by improving their register allocators 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 value profiling runs and additional recompilations, or else specialized dynamic code generating compilers 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 biggest change that needs to be made is to have the optimization adapt its the specialized traces along with runtime constants may change their values as the program enters different phases. Even without this functionality implemented, the underlying system shows that runtime code specialization is a viable technique even when applications are developed using traditional tools. Hopefully future work will build upon the structure described in this thesis to allow runtime code specialization become a viable tool for everyday computing.

Bibliography

- [1] IA-32 Intel architecture software developer's manual (order number 245470). Volume 1: Basic architecture. <http://www.intel.com/design/pentium4/manuals/245470.htm>.
- [2] Intel Pentium 4 and Intel Xeon processor optimization reference manual. (order number 248966-04).
- [3] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeno JVM. In *Conference on Object-Oriented Programming Languages*, pages 47–65, 2000.
- [4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2000.
- [5] Aart Bik, Milind Girkar, Paul Grey, and Xinmin Tian. Efficient exploitation of parallelism on Pentium III and Pentium 4 processor-based systems, Q1 2001.
- [6] Derek Bruening, Evelyn Duesterwald, and Sanjeev Banerjia. A transparent runtime optimization system. In *Proceedings of the ACM SIGPLAN Conferences on Programming Language Design and Implementation (PLDI '00)*, June 2000.
- [7] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *International Symposium on Microarchitecture*, pages 259–269, 1997.
- [8] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M. Gillies. Mojo: A

- dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, December 2000.
- [9] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. FX!32 a profile-directed binary translator. In *IEEE Micro*, March-April 1998.
- [10] Manel Fernandez, Roger Espasa, and Saumya K. Debray. Load redundancy elimination on executable code. In *European Conference on Parallel Processing*, pages 221–229, 2001.
- [11] Brian Grant, Matthai Philipose Markus Mock, Craig Chambers, and Susan J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Conference on Programming Language Design and Implementation*, May 1999.
- [12] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, Inc., San Francisco, California, 1996.
- [13] T. Kistler and M. Franz. Perpetual adaptation of software to hardware: An extensible architecture for providing code optimization as a central system service, 1999.
- [14] Mikko H. Lipasti, Christopher B. Wilkerson, and John P. Shen. Value locality and load value prediction. In *Architectural Support for Programming Languages and Operating Systems*, pages 138–147, 1996.
- [15] Robert Muth, Scott A. Watterson, and Saumya K. Debray. Code specialization based on value profiles. In *Static Analysis Symposium*, pages 340–359, 2000.
- [16] Francois Noel, Luke Hornof, Charles Consel, and Julia L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study.
- [17] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *SIGPLAN*

Conference on Programming Language Design and Implementation, pages 109–121, 1997.

- [18] Eric Rotenberg, Steve Bennett, and James E. Smith. A trace cache microarchitecture and evaluation. *IEEE Transactions on Computers*, 48(2):111–120, 1999.
- [19] Mike Smith. Overcoming the challenges to feedback-directed optimization. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo '00)*, 2000.
- [20] SPEC CPU2000 benchmark suite. Standard Performance Evaluation Corporation. <http://www.spec.org/osg/cpu2000/>.
- [21] Bjorn De Sutter and et al. Whole-program optimization of binary executables.
- [22] Scott A. Watterson and Saumya K. Debray. Goal-directed value profiling. In *Computational Complexity*, pages 319–333, 2001.