HEWLETT PACKARD

Transparent Dynamic Optimization: The Design and Implementation of Dynamo

Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia
HP Laboratories Cambridge
HPL-1999-78
June, 1999

E-mail: vas@hpl.hp.com

dynamic optimization, compiler, trace selection, binary translation

Dynamic optimization refers to the *runtime* optimization of a *native* program binary. This report describes the design and implementation of Dynamo, a prototype dynamic optimizer that is capable of optimizing a native program binary at runtime. Dynamo is a realistic implementation, not a simulation, that is written entirely in user-level software, and runs on a PA-RISC machine under the HPUX operating system. Dynamo does not depend on any special programming language, compiler, operating system or hardware support.

Contrary to intuition, we demonstrate that it is possible to use a piece of software to improve the performance of a native, statically optimized program binary, *while it is executing*. Dynamo not only speeds up real application programs, its performance improvement is often quite significant. For example, the performance of many +O2 optimized SPECint95 binaries running under Dynamo is comparable to the performance of their +O4 optimized version running without Dynamo.

© Copyright Hewlett-Packard Company 1999

Contents

1 INTRODUCTION 7

2 RELATED WORK 9

3 OVERVIEW 11

3.1 Design Goals 11
3.2 How Dynamo works 12
3.3 Innovative features 13
3.4 Bailing out to direct execution 14
3.5 System calls 14
3.6 Self-modifying code 15

4 MEMORY LAYOUT 16

4.1 Address space separation 18

5 CONTROL 20

5.1 Context switches 21
5.2 Trapping control 21
5.3 Signal handling 22

6 INTERPRETER 24

6.1 Plug-in interpreter interface 24
6.2 Interpretation via just-in-time translation 25

7 TRACE SELECTOR 27

7.1 Online profiling 27
7.2 Speculative trace selection (SPECL) 28
7.2.1 Basic block selection (BLOCK) 29
7.2.2 Static trace selection (STATIC) 30
7.2.3 Two-level hierarchical selection (2-LEVEL) 31
7.3 Trace selection rate 32
7.4 Bailing out 33
7.5 Evaluation 34
7.6 Trace quality metric 36
7.7 Selection threshold 38

8 FRAGMENT OPTIMIZER 40

8.1 Dynamic optimization opportunities 41
8.1.1 Exposed partial redundancies 41
8.1.2 Suboptimal register allocation 42
8.1.3 Shared library invocation 44
8.2 Intermediate form 44
8.3 Fragment formation 45
8.3.1 Direct branch Adjustments 46

8.3.2 Indirect branch Adjustments 46
8.3.3 Linker Stubs 46
8.4 Optimization 47
8.4.1 Optimization levels 47
8.4.2 Speculative optimizations 48
8.4.3 Multi-versioning optimization 49
8.4.4 Fragment-link-time optimization 49
8.5 Undesirable side-effects of optimization 51
8.6 Register allocation 52
8.7 Experimental evaluation 52

9 FRAGMENT LINKER 56
9.1 Emitting the fragment code 56
9.2 Pros and cons of branch linking 56
9.3 Link records and the fragment lookup table 57
9.4 Direct branch linking 58
9.5 Indirect branch linking 59
9.6 Linker stubs revisited 59
9.7 Branch unlinking 60
9.8 Performance benefit from linking 62

10 FRAGMENT CACHE MANAGER 63
10.1 Fragment Cache design 63
10.2 Working-set based Fragment Cache flushing 64
10.2.1 The case for Fragment Cache flushing 65
10.2.2 Selecting a time to flush 66
10.2.3 Performance: memory usage 68
10.2.4 Performance: speedups 71
10.3 Alternative placement strategy 74
10.3.1 Functionality and behavior 75
10.3.2 Performance 75

11 SIGNAL HANDLER 78

12 PERFORMANCE SUMMARY 80
12.1 Experimental framework 80
12.2 Experimental results 80
12.3 Dynamo code and memory footprint 82
12.4 Dynamo on highly optimized programs 83
12.5 Machine specific performance: the PA-8000 84
12.5.1 Branch prediction 85
12.5.2 Indirect branches 85
12.5.3 Virtual memory 85
12.6 Other programs and processors 85
12.6.1 Program traits 86
12.6.2 Machine traits 87

13 SYSTEM DEVELOPMENT TOOLS AND UTILITIES 89

13.1 The system profiler - sprof 89
13.1.1 Specifying what to profile: the profiling registry 89
13.1.2 Gathering profiling data: timer interrupts 89
13.1.3 Reporting profiling data: the back-end 90
13.1.4 Examples of profiling data..91
13.2 Memory allocators 93
13.2.1 The mempool allocator 93
13.2.2 The poolmalloc allocator 94
13.2.3 Memory usage in Dynamo..94
13.2.4 Retrospective...95

14 CONCLUSION... 96

15 ACKNOWLEDGEMENTS 98

16 REFERENCES 99

Figures

Figure 1: How Dynamo works 12
Figure 2: Memory layout of a program executing under Dynamo 17
Figure 3: Control flow between the components of the Dynamo system 20
Figure 4: Use of linker stubs to trap control on a Fragment Cache exit 22
Figure 5: Handshake between the Interpreter and Trace Selector 24
Figure 6: Interpretive overheads 26
Figure 7: Dynamo's trace selection scheme with trace heads shown in bold 29
Figure 8: 2-Level hierarchical selection (2-LEVEL) 31
Figure 9: Average trace selection rate 32
Figure 10: Trace selection rate for go and li 33
Figure 11: Overall Dynamo speedup based on SPECL trace selection without and with bail-out 34
Figure 12: Number of traces selected by each selection scheme 35
Figure 13: Accumulated fragment size in Kbytes for each selection scheme 36
Figure 14: 90%-Cover sets 37
Figure 15: Overall Dynamo speedup with varying selection schemes 37
Figure 16: Accumulated fragment size with varying threshold T 38
Figure 17: Overall Dynamo speedup with varying trace selection threshold for SPECL 38
Figure 18: Anatomy of a fragment 45
Figure 19: (i) Indirect branch with actual target 0x20, and (ii) its conversion into a direct branch 46
Figure 20: Example of guarded load optimization 49
Figure 21: Example of link-time optimization 50
Figure 22: Fragment epilog and prolog used for link-time optimization 51
Figure 23: Overall Dynamo speedup with optimization contribution 54
Figure 24: Optimization overhead 55
Figure 25: The linker stub template 59
Figure 26: Illustration of linking and unlinking 61
Figure 27: Performance slowdown when linking is disabled 62
Figure 28: Fragment Cache memory map with 2 fragments 63
Figure 29: Fragment creation rate pattern for m88ksim 66
Figure 30: State diagram for phase change detection logic 67
Figure 31: Fragment Cache memory usage with and without WS flushing 68
Figure 32: Dynamic changes in Fragment Cache memory usage with WS flushing in m88ksim .69
Figure 33: Memory usage in dynamic pools with and without WS flushing 70
Figure 34: Run time performance of WS flushing compared to no WS flushing 71
Figure 35: Number of flushes caused by WS flushing 72
Figure 36: Run time performance of interval flushing versus WS flushing 73
Figure 37: Fragment Cache page usage with and without WS flushing 74
Figure 38: Stub-based linking with and without WS flushing 74
Figure 39: Run time performance of the split cache 76
Figure 40: Fragment Cache page usage with and without the split cache 76
Figure 41: Stub-based linking with and without the split cache. 77
Figure 42: Dynamo performance (light/yellow bars indicate that Dynamo bailed-out) 81
Figure 43: Dynamo overhead (light/yellow bars indicate that Dynamo bailed-out) 81
Figure 44: Fragment Cache and Dynamo memory pool footprint 82
Figure 45: Breakdown of the Dynamo code footprint (underline indicates PA-dependent code) .83
Figure 46: Performance comparison of Dynamo against native execution 84
Figure 47: Fragment creation rate pattern for perl 86

Figure 48: Fragment creation rate pattern for ijpeg......87
Figure 49: Fragment Cache hit rate and breakdown of Dynamo overhead 91
Figure 50: Counter values from registered functions 92
Figure 51: Fragment profiling data 92

1 Introduction

As software and hardware technologies evolve in a rapidly changing marketplace, there are several forces pulling in opposite directions, making cost-effective performance delivery ever more challenging.

Consider the high-end server market for example. In the drive for greater performance, microprocessors now provide capabilities for compiler software to take on a greater role in performance delivery [Gwennap 1997]. In the meantime, however, trends in software technology are gradually increasing the obstacles to static compiler optimization. The widespread use of object-oriented programming languages and the trend towards shipping software binaries as collections of DLLs (dynamically linked libraries) rather than monolithic binaries, has resulted in a greater degree of run-time binding. Run-time binding offers the advantage of being able to ship software patches to the customer after the product sale has been made, and also eases the integration of third-party middleware into the product. Unfortunately, it also makes traditional compiler optimization, which operates on the statically bound scope of the program, more difficult. From a practical standpoint, the use of compiler optimization is also limited by the inability to effectively debug highly optimized code. For this reason, many ISVs (independent software vendors) are reluctant to ship software binaries that are compiled with high levels of optimization, in spite of their improved performance potential.

In the desktop PC market on the other hand, performance delivery is complicated by the prevalence of legacy binaries. Recompilation is clearly an impractical means for improving program performance, given the sheer volume of legacy software. As a result, the microprocessor hardware has taken on most of the performance burden. Unfortunately, when system prices drop rapidly but performance expectations don't, this strategy starts to become impractical.

Then there is the emerging Internet and mobile communications marketplace. In a networked device where code can be downloaded and linked in on the fly, the role of traditional performance delivery mechanisms like static compiler optimization is unclear. Furthermore, the tendency of advanced compiler optimizations to produce fat program binaries restricts their applicability in small memory-footprint mobile devices.

With all of these forces pulling in opposite directions, there is a need for new performance delivery technologies to bridge the widening gap. It is clear that the current model, where the performance delivery task is divided only between the microprocessor hardware and the compiler software is very limited for use in the newly emerging computing environment.

In this report we propose a radically different performance delivery technology. The application is optimized at the time the native bits are *executed* on the microprocessor, rather than when the native bits are *created* as in the case of a static compiler. We refer to this as *dynamic optimization*, and in principle, it is similar to what a superscalar reorder buffer does in a modern microprocessor: the *runtime optimization* of a *native* instruction stream. In contrast to a hardware dynamic optimizer however, our scheme works entirely in software, and yet retains the "transparent" nature of hardware dynamic optimization. One can view this as a "virtual microprocessor", where the dynamic optimization component is implemented in software while the CPU core is implemented in hardware.

We have implemented a prototype called **Dynamo** that demonstrates the feasibility of this idea in a very compelling way, and represents the culmination of three years of research conducted at Hewlett-Packard Labs [Bala and Freudenberger 1996]. Dynamo is a transparent dynamic optimizer that performs optimizations at runtime on a native binary. Dynamo does not rely on any annotations in the original program binary. It uses interpretation as a means of observing program behavior without having to instrument it. This allows Dynamo to operate on legacy binaries without recompilation or instrumentation. No information is written out during or after the program's execution for consultation during a later run, and no pre-runtime analysis is

done on the binary. Instead, the profiling information that is collected via interpretation is consumed on the fly to dynamically select hot instruction traces from the program. The hot traces are optimized using a low overhead optimizer and placed in a software code cache. Subsequent occurrences of these traces cause the cached version to be executed, giving a performance boost. The original loaded image of the program binary is never modified by Dynamo, allowing Dynamo to even run programs that modify their own text image. The operation of Dynamo is thus completely transparent to the end-user.

The Dynamo system is implemented entirely in software, and does not depend on any special compiler, operating system or hardware support. Dynamo has a compact code footprint, allowing it to comfortably fit within typical ROM constraints if necessary, and its data memory footprint is configurable for a variety of performance/memory tradeoffs. This allows the dynamic optimization technology to be deployed in a wide range of platforms, from large servers to embedded devices.

Dynamo is a fundamentally different performance delivery vehicle. Unlike a traditional compiler, the ISV is not responsible for enabling the optimization. In contrast to a JIT, there is no translation component: the input is a native instruction stream. And unlike a superscalar microprocessor, no special hardware is used for doing the optimization. Dynamo is a realistic implementation, and not a simulation. Our prototype currently runs on a PA-8000 workstation under the HPUX 10.20 (HP Unix) operating system.

2 Related Work

A lot of work has been done on dynamic translation as a technique for non-native system emulation [Bedichek 1995; Cmelik and Keppel 1993; Herold 1998; Hohensee et al. 1996; Insignia 1991; May 1987; Stears 1994, Witchel and Rosenblum 1996]. The basic idea is simple: interpretation is too expensive a way to emulate a long running program on a host machine. Caching the native code translations of frequently interpreted regions can lower the overhead. If sufficient time is spent executing the cached translations to offset the overhead of interpretation and translation, the caching strategy will generally outperform interpretation in performance. Some implementations even bypass the interpreter completely and translate to native code prior to executing every unit [Cramer et al. 1997; Griswold 1998; Ebcioglu and Altman 1997]. More recently, hybrid dynamic translation systems are emerging that use a combination of software and custom hardware [Kelly et al. 1998].

Dynamic *optimization* is different from dynamic translation: the input is a *native* program binary, so there is no "translation" step involved. While the goal of a dynamic translator is to beat the performance of pure interpretation, the goal of a dynamic optimizer is to beat the performance of the input program executing directly on the host machine.

Dynamic optimization is also different from dynamic *compilation* as generally defined in the literature. In dynamic compilation, the programmer either explicitly requests that a specific part of the program be compiled at runtime via program annotations [Auslander et al. 1996; Leone and Dybvig 1997 [1]; Consel and Noel 1996; Leone and Lee 1996] or uses a language that allows efficient runtime compilation [Engler 1996; Holzle 1994; Poletta et al. 1997]. The code generated at compile-time contains stubs to transfer control at runtime to a compiler, which then dynamically generates the native executable code. In contrast, we view dynamic *optimization* as a much more transparent process. No special programming language or compiler annotations should be necessary to trigger its activation. It should work on already compiled native code without the need for any additional processing, including instrumentation of the binary. In the case of dynamic compilation, the runtime compiler is a necessary step for the execution of the program, whereas in the case of dynamic optimization, it is not. At any moment, the dynamic optimizer has the option of "bailing out" and letting the input program execute directly on the underlying processor.

There are several implementations of *offline binary translators* that also perform native code optimization [Chernoff et al. 1998; Hookway and Herdeg 1997; Sites et al.]. These generate profile data during the initial run via emulation, and perform background translation together with optimization of hot spots based on the profile data. Again, this strategy is not transparent in the sense that the dynamic optimizer is. The benefit of the optimization is only available during subsequent runs of the program. In the case of a dynamic optimizer, on the other hand, profile data is generated and consumed in the very same run, and no data is written out for use offline or during a later run.

Hardware implementations of dynamic optimizers are now commonplace in modern microprocessors [Kumar 1996; Song et al. 1995; Papworth 1996; Keller 1996]. The optimization unit is a fixed size instruction window, with the optimization logic operating on the critical execution path. The Trace Cache is another hardware alternative that can be extended to do superscalar-like optimization off the critical path [Peleg and Weiser 1994; Rotenberg et al. 1996; Friendly et al. 1998]. But we are unaware of any attempts to implement a transparent dynamic optimizer entirely in software. This is perhaps not without good reason: conventional wisdom

[1] The Indiana project is also called "Dynamo". We regret the name collision. When our project was started in the winter of 1995, we were unaware of the other Dynamo project. We have since used the Dynamo name in numerous internal documents, memos, presentations and patent filings, making the task of renaming our project at this stage somewhat more difficult than one might imagine.

flies in the face of the suggestion that a software dynamic optimizer can improve the performance of a compiled native program binary while it executes on the host machine.

3 Overview

Dynamic optimization is not intended to replace static compiler optimization, rather the two are for the most part complementary. The static compiler optimizer performs its optimizations based on the static scope of the program, whereas the dynamic optimizer performs its optimizations based on the dynamic scope of the program, after all runtime objects have been bound. The dynamic optimizer can be viewed as performing a last minute correction of the most frequently executed instruction sequences in the running program. This allows it to perform optimizations across dynamically linked procedure call boundaries and virtual function calls, which would be difficult to do in a static compiler. It can also perform runtime path-specific optimizations without having to do extensive code duplication to separate out the path from the surrounding control flow. Dynamic optimization opportunities exist even in programs compiled at the highest static optimization levels.

3.1 Design Goals

The vision of a totally transparent software dynamic optimizer is nevertheless a particularly challenging one, given that it has to beat the performance of compiled native code handily in order to gain wide acceptance. The following were the principal design goals for the Dynamo project:

1. *Performance*: a program binary under dynamic optimization should run at least as fast as the same program binary running directly on the underlying processor. Input program binaries will be created using the native compiler using the default optimization level (-O). Performance comparison will be based on actual wall-clock time on a real machine, and not simulated cycles.
2. *Transparency*: the behavior of the program under dynamic optimization should be indistinguishable from that of the same program running by itself, except for any performance differential. This means guaranteeing reproducibility of behavior even in the presence of signals. It also means preserving any bugs in the original code. A consequence of this is that the original program binary must be loaded at the identical address in memory, and its loaded binary image cannot be modified by the dynamic optimizer.
3. *Compactness*: the system should have a compact footprint (no more than a few hundred KB), and its data memory usage should be tunable for different performance/memory tradeoff points. This would allow the system to be deployed in devices ranging from large servers to small handheld mobile gadgets.
4. *Instrumentability*: since this is intended to be a research infrastructure, the system should be thoroughly instrumented to study and understand its behavior. Performance should not be impacted when the instrumentation is shut off.

From the outset it was clear that every component of Dynamo would have to be very carefully engineered, because overhead of dynamic optimization is such a critical issue. In that sense, we couldn’t afford to merely build a proof-of-concept research prototype; the system had to be fairly robust in order to come even close to meeting the above design goals. Even so, these are difficult goals to achieve, especially since our implementation does not rely on any operating system or hardware support. The transparency goal was by far the most difficult one to meet. While we cannot claim that we have met every aspect of these goals in our prototype, we have managed to come a lot closer than we had imagined would be possible at the start.

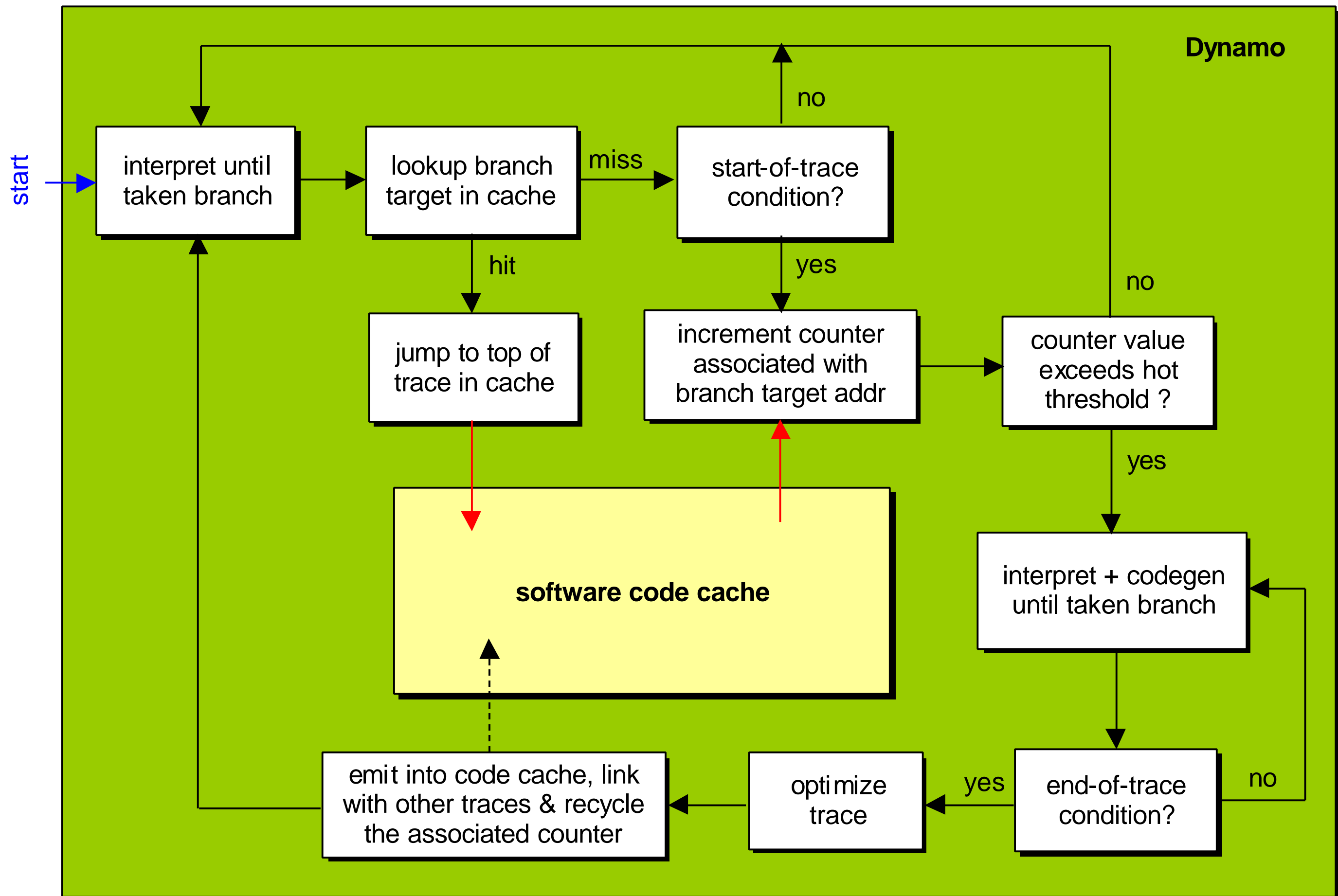


Figure 1: How Dynamo works

3.2 How Dynamo works

Dynamo is currently implemented for the PA-RISC platform (a PA-8000 based workstation running HPUX 10.20). It is written in a combination of C and assembler. Although the implementation takes advantage of some features of the PA instruction set architecture to improve the efficiency of certain functions, it does not rely on them for operation. Thus in principle, Dynamo can be ported to other processor architectures.

Figure 1 shows a high level outline of how Dynamo works. Dynamo starts out by interpreting the input program's instructions. The input program must be a user-mode native executable (supervisor-mode programs can create complications that our current prototype cannot yet handle reliably). It can be a legacy binary, and may invoke dynamically linked libraries (shared libraries) and system calls. It can also arm signal handlers. The "interpreter" here is a *native* instruction interpreter, implemented in software. Interpretation is primarily used as a means for observing program behavior without having to instrument the program binary. As a side effect of interpretation, Dynamo increments counters associated with certain program addresses that satisfy a "start-of-trace" condition (such as the interpretation of a backward taken branch). If a counter value exceeds a preset hot threshold, the interpreter toggles state and goes into "code generation" mode. When interpreting in this mode, the native instructions comprising the trace are emitted into a buffer. When an "end-of-trace" condition is reached, the contents of

this buffer are optimized by a fast, lightweight optimizer. The optimizer makes the trace instructions into an executable unit called a *fragment*, optimizes the fragment, and then hands it to a linker. The linker emits the fragment code into a software code cache, and links it to other fragments already in the cache. The fragment is tagged with the program address corresponding to the first instruction in the sequence (referred to as the "fragment entry point").

Every time the interpreter, when interpreting in its normal mode, interprets a taken branch instruction, it looks up the software cache to see if a fragment exists whose tag is equal to the branch target PC. If there is a hit, control immediately jumps to the top of the fragment, causing that fragment code (and others that may be linked to it) to execute directly on the underlying processor. At this point, the execution of Dynamo is effectively suspended, and the optimized program instructions in the software code cache are executing directly on the processor. The optimized fragments from the software cache gradually start populating the underlying processor's instruction cache, and the end-user perceives a performance boost.

Eventually, a branch exits the software code cache address space, causing control to trap to Dynamo. The execution of the optimized program instructions within the software code cache is now suspended, and Dynamo code is now executing on the underlying processor. A counter associated with the exit branch target is incremented, and if this counter exceeds another preset threshold, the interpreter is invoked in its code generation mode, causing it to produce a fresh trace. Otherwise, the interpreter is invoked in its normal mode of operation, completing the Dynamo execution loop. The rationale here is that if an exit from a hot trace itself gets hot, a new trace should be formed starting at the exit branch target. Gradually, as more and more hot traces are materialized in the software code cache, an increasingly greater time is spent executing within the software cache, and a correspondingly smaller proportion of time is spent in Dynamo.

3.3 Innovative features

At the abstract level of the above description, the working of Dynamo is not very different from that of recent dynamic translator implementations like Shade [Cmelik and Keppel 1994]. Yet, Dynamo's performance is quite startling in comparison to past dynamic translator implementations, especially given that it significantly speeds up *native*, *statically optimized* program binaries. And it manages to do so with a very compact code and data memory footprint (approximately 250 KB and 300 KB respectively in our current version). Three innovations make all of this possible.

The first, and probably the most important, is Dynamo's hot trace selection scheme. It is highly speculative in nature, and uses very minimal profiling. The essence of the idea is that when a program address gets hot, the sequence of addresses executed immediately following it are likely to be a hot trace. Thus, instead of using complicated path profiling, Dynamo only associates counters with certain special program addresses (such as targets of backward taken branches). When such an address gets hot, the interpreter emits a copy of the interpreted native instructions that immediately follow the one at the hot address. Not only does this provide a statistically likely "hot" trace, but it also allows the trace to be grown past indirect branches and dynamically linked calls. The appeal of this approach is its simplicity and elegance of implementation. In our extensive experiments with alternative region selection schemes, we could never come up with one that matched the performance/overhead ratio of this speculative scheme.

The second innovation is the use of a dynamic trace as the granularity of runtime optimization. The instructions comprising a dynamic trace are often non-contiguous in memory, and thus may not have been part of the static program scope that was available to the generator of the program binary. A dynamic trace is therefore likely to expose new optimization opportunities, even in a statically optimized program binary. Furthermore, Dynamo's traces tend to begin at the targets of backward taken branches, and very often, a cycle of traces is quickly captured within the software code cache by the trace selection scheme. The traces can only be entered at the top,

and they have no control join points within them. This simplicity of control flow allowed us to design a very fast, non-iterative, trace optimizer, that turned out to be highly effective. Dynamo only optimizes each trace once, at the time it is emitted into the software code cache, and does not use any form of "staged optimization". Yet it produces significant performance improvements, even on input programs that were created using high static optimization levels. In our current prototype, the optimizer does not do any instruction scheduling, since the input program already contains scheduled code. Instruction scheduling has the potential of increasing the runtime optimization overhead substantially, depending on the nature of the underlying processor. On the other hand, the time spent within the optimizer is a negligible fraction of the overall Dynamo overhead. If other components of the Dynamo code could be made more efficient, the extra headroom may be sufficient to offset the scheduling overhead.

The third key innovation was our software code cache design. All modern dynamic translators use a software code cache of some sort. However, to the best of our knowledge, all of these implementations size the code cache large enough to avoid the overhead of cache management due to capacity and conflict misses. Thus, multi-megabyte code caches are not uncommon in most dynamic translator implementation. Our approach is exactly the opposite. The Dynamo code cache is sized very small (300 KB in our current implementation), and the system tolerates a significant number of cache flushes. The trick is to do pre-emptive flushing of the software code cache, in anticipation of a working set change. This performs far better than a reactive policy that flushes the cache only on a capacity miss, because the reactive policy often ends up throwing away useful traces in the current working set along with the old traces. This in turn increases the overhead of re-creating the useful traces once again. The code cache design in Dynamo is the principle reason for its small data memory footprint. All internal trace-related data structures in Dynamo are tied to the code cache flush, so that a flush automatically resets all of these memory pools as a side effect.

3.4 Bailing out to direct execution

Since performance is an essential goal in our design, Dynamo must avoid situations where there is little likelihood of performance improvement. An advantage that the dynamic optimizer enjoys over other dynamic compilation or dynamic translation systems is the ability to "bail-out", and let the input (native) program execute directly on the underlying processor. Dynamo monitors the ratio of time spent between the software code cache and Dynamo code. When this ratio falls below a threshold, Dynamo bails-out and lets the input program execute directly. Dynamo will also bail out whenever it encounters tricky situations that confuse it.

The disadvantage of bailing out is that our prototype is currently incapable of resuming later on the same program. Once Dynamo bails out, the program continues to execute directly on the processor, and Dynamo never regains control. Since Dynamo has to guard against bailing out too early, it can cause a significant slowdown in very short running programs. In general, a program has to run at least 1.5 mins for our prototype to achieve a speedup or come close to break-even performance through bail-out.

3.5 System calls

Dynamo has to examine all system calls in the program code. Usually, this is necessary to avoid losing control over the program, in cases where the kernel might directly invoke a program routine, undercutting Dynamo. An example of this is system calls that install signal handlers. Dynamo intercepts all signal handler installation calls, and substitutes its own handler instead. The kernel will transfer control to Dynamo when the signal arrives, and the program's handler code is then executed under Dynamo control.

Sometimes however, the situation can get quite complicated, and Dynamo has to make a decision on whether it can continue to maintain control over the program without compromising

all the other design goals, such as transparency and performance. An example of this situation is when the program code contains a call to *fork* or *exec*. In principle, it is possible for Dynamo to inject itself into the executable image of the child process, similar to the way the current process was set up to run under Dynamo. Rather than deal with the additional complication this creates however, our prototype will bail out to direct program execution if a *fork* or *exec* is encountered in the program code.

System calls that manipulate the program's resource limits (such as *setrlimit*) present another tricky case. Our approach is to allow calls that increase the program's soft limits, but to bail out if the program code attempts to raise its hard limits.

3.6 Self-modifying code

Self-modifying code is code that writes into its own text segment. Self-modification creates a problem, because when a program modifies its own text segment at runtime, it can result in stale program code in Dynamo's software code cache. Fortunately, on the PA-RISC (as well as on most modern microprocessors), the program is expected to explicitly synchronize the processor's instruction cache with the memory image if it writes into its text segment. It does so by invalidating specific lines, or the entire I-cache, forcing it to be re-filled from memory. On the PA-RISC, the *flush* instruction allows invalidation on a line granularity. When Dynamo detects such an instruction in the program code, it immediately flushes the entire contents of the software code cache, so that it gets re-populated with new fragments. This allows Dynamo to even dynamically optimize programs that modify their own text image.

4 Memory Layout

An important aspect of our design is that the entire execution of the application program is done under Dynamo control. The original application program instructions never actually execute at their original program text address. Every program instruction is either interpreted by Dynamo or a copy is executed within the software code cache. The same is true of any dynamically linked libraries that may be invoked by the program.

Dynamo thus has to first gain control of the program before the program starts execution. Since transparency is an essential design goal, we have to accomplish this without perturbing the original program's executable image. Otherwise, it will be impossible to ensure precise exception delivery under dynamic optimization (an already difficult task to begin with). We must therefore load the program binary at the same virtual address, whether or not it is going to be dynamically optimized. There are several ways to achieve this, and each has its own pros and cons:

1. Modify the kernel loader. Dynamo can be compiled as a shared library that is automatically loaded by the kernel loader when it loads the application program image. The kernel loader then calls the Dynamo entry point instead of the program's main entry point. The advantage of this scheme is that it is truly transparent to the user. The disadvantage is that it requires operating system modification.
2. Use *ptrace* to attach to the application program. Ptrace is a mechanism that allows one process to control another, and is typically used by debuggers. Dynamo can be set up as a separate process that attaches to the application program via ptrace, and runs it until the point where *crt0* (the execution start up code at the top of the program's binary image) is about to call the program's entry point. Execution of the application program is then suspended, and Dynamo fetches the program instructions and executes them on its behalf. Like solution 1, this is also transparent, except for the creation of an additional process. The disadvantage of this scheme is its dependence on the ptrace interface. Many operating systems (like embedded RTOSs for example) do not support ptrace.
3. Extend the program's text segment in a separate copy of the program executable file. The application program executable file can be copied over to a temporary location, and the program text segment extended by adding the Dynamo text at the end. Then, the *start* symbol (the entry point that is called by *crt0*) is changed to the Dynamo entry point. This new executable file is then *exec*ed. The original program text is still loaded at the same virtual address as it would normally have, but Dynamo will gain control before the actual application program starts. The advantage of this solution is that it does not require modification of any kernel routines, nor does it rely on any special operating system features like ptrace. It is a complete user space solution. The disadvantage is the overhead of doing the file copy (recall that modification of the original program executable file is not an option as per our project design goals).
4. Use a special version of *crt0*. Crt0.o is the execution start up code (typically created from the assembly file crt0.s) that is linked in to the executable by the link editor **ld** at link-time. The kernel loader transfers control to the top of crt0 after it has loaded the entire executable image. The crt0 code is responsible for picking up the command line arguments, setting up the initial stack and data segment, and then making a call to the value of the *start* symbol (usually the main() function of the program). Prior to calling the program entry point, crt0 maps the dynamic link loader **dld**, which then loads any dynamically linked libraries (shared libraries) referenced by the application program. A custom version of crt0 can be used to additionally map the Dynamo code (itself compiled as a shared library), and call Dynamo's entry point instead of the one defined by the *start* symbol. The problem with this scheme is that it requires re-linking of the object files that constitute the application program. Solutions 1-3 on the other hand will work with legacy binaries without re-linking. One way to

overcome this problem is to use a special version of **dld**, which loads the Dynamo shared library in addition to any libraries invoked by the program code, and patches the crt0 code so that it jumps to Dynamo's entry point instead of the program's.

All of the above solutions preserve an essential requirement of the transparency goal: they do not modify the application program code. The program instructions generated by the compiler are loaded unmodified into memory. The only modification done is to the execution start up code in *crt0*; and solutions 1 and 2 do not even do this. Furthermore, no re-compilation of the program is necessary, only solution 4 requires re-linking, but using a modified version of **dld** can circumvent this. Note however, that if Dynamo were to be pre-installed in a new system, the system could be shipped with a custom version of crt0.o, preventing the need to re-link binaries created on that system. In this report, we assume solution 4 is used to start up Dynamo.

Figure 2 illustrates a typical memory layout when a program is running under Dynamo control. Dynamo is compiled as a position-independent shared library, allowing several processes running under Dynamo to use a single copy of the Dynamo code. The Dynamo text segment itself can be ROM resident. The Dynamo data segment is process-local, as is the case in any shared library.

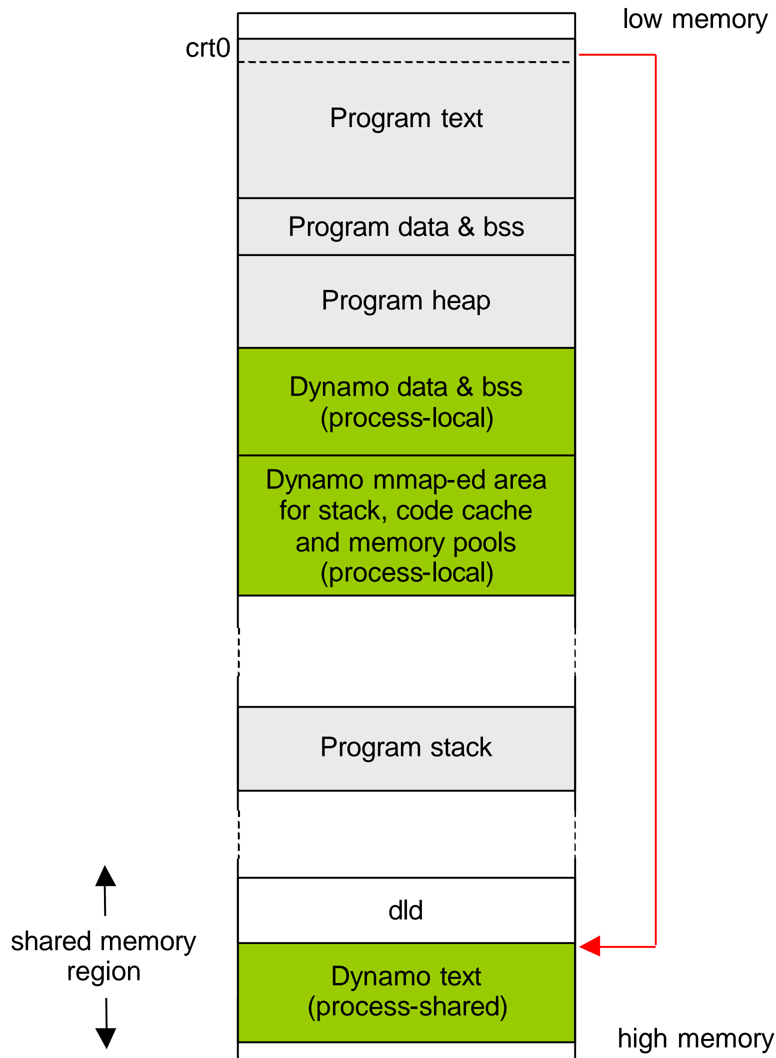


Figure 2: Memory layout of a program executing under Dynamo

In contrast to a regular shared library however, Dynamo does not use any part of the program's data segment, including the program's stack or heap. Instead, Dynamo *mmaps* a separate area of memory, which it then manages itself, to allocate its runtime stack and software code cache. No recursive routines are used within Dynamo, and the longest call-chain within Dynamo is known apriori. This allows Dynamo to allocate a fixed amount of room for its runtime stack. Dynamo does not use a malloc-style heap; the separate *mmap*-ed area is used to allocate its dynamic objects, using a custom memory allocator that is part of Dynamo. Thus, Dynamo's operation does not interfere with the application program's runtime stack or heap area in any way, a critical requirement for meeting our transparency goals.

On systems where *mmap* is unavailable (such as in an embedded RTOS), this space has to be statically allocated, for instance by declaring a large enough array to force the bss segment to a certain size. The drawback here is that page-level protection is not possible the way *mmap* allows.

4.1 Address space separation

An application program instruction is "executed" only through interpretation, or via direct execution of a copy of it in the software code cache managed by Dynamo. The entire execution of the application program is thus done under Dynamo control. An implication of this is that all application program instructions are actually executed within the same address space as Dynamo. This opens the opportunity for inadvertent or even malicious corruption of the Dynamo data segment (including the contents of the code cache) by the program being dynamically optimized.

The pages containing the Dynamo data segment and the additional *mmap*-ed memory area are marked writeable only when control is within Dynamo. Just before Dynamo transfers control into the software code cache, it sets the protection flags on these pages to be read-only. When control exits the software cache, the original page protections are restored. This prevents any instructions executed within the software code cache from writing into the Dynamo data segments. Dynamo also watches for any system calls in the program that can modify the protection bits associated with a page (such as *mmap*, *munmap*, and *mprotect*). These system calls are always interpreted by Dynamo, and the program code that invokes them is never generated into the software code cache. The interpreter first checks to see if the pages referred to by these calls are outside the Dynamo text and data space. If not, an exception is raised and the system call is not allowed to go through.

Dynamo prevents rogue branches in the program code from jumping into the Dynamo text segment by always trapping control when any branch exits the fragment cache address range[2]. If the exit branch's target address is not within the application program's text segment address range, an illegal address exception is raised.

It is possible that a program branch instruction may target a bad address that coincidentally happens to be within the software cache address space. In the original program its execution would have resulted in an illegal address trap, but when a copy of this instruction is executed within the fragment cache itself, it is legal. Fortunately, this is easily checked, since Dynamo controls the linking of every branch that crosses fragments. At branch linking time (when a fragment is connected to others within the code cache), if the branch target is within the Dynamo text or data segments, Dynamo realizes that this must be a bad address, and the branch instruction is interpreted rather than executed within the fragment cache. The Dynamo interpreter will raise the appropriate exception. This strategy works even for indirect branches whose targets are determined dynamically. For indirect branches, Dynamo generates a jump to a lookup code that is permanently resident in the software code cache. The same check can be performed prior to the lookup.

---

[2] Dynamo can do this because it controls the generation of all code into its fragment cache. A branch whose target is not in the software code cache is made to jump to a special stub that traps control back to Dynamo.

While we have made every attempt to bulletproof our prototype against inadvertent or malicious corruption of the dynamic optimizer system and its code cache, our implementation is not immune from such situations. Having to execute in the same user address space as the program being dynamically optimized creates many obstacles to building a truly robust and transparent system. One could envision Dynamo as eventually becoming a special low-level software layer that is either directly supported by the underlying processor, or is a special operating system service. Such an implementation would allow it to be engineered in a much more transparent and robust manner than was possible in our prototype.

5 Control

Figure 3 illustrates the various components of the Dynamo system, and the flow of control between them. The Init component is the main entry point into Dynamo. Control must enter here before any of the application program instructions have started executing. Init is responsible for initializing various internal data structures, and *mmap*-ing space for allocating the Dynamo runtime stack, the software code cache (henceforth referred to as the Fragment Cache), and memory pools for dynamically allocated objects. It then calls the Dispatch component. Dispatch expects to be passed a machine context completely specifying the application program's execution context in the underlying machine, with the value of the PC set to the next executable application program address. Dispatch is the central point of control for the entire Dynamo system. It manages transfer of control between Dynamo and the fragment cache, as well as any exceptions or interrupts that occur when executing within the code cache.

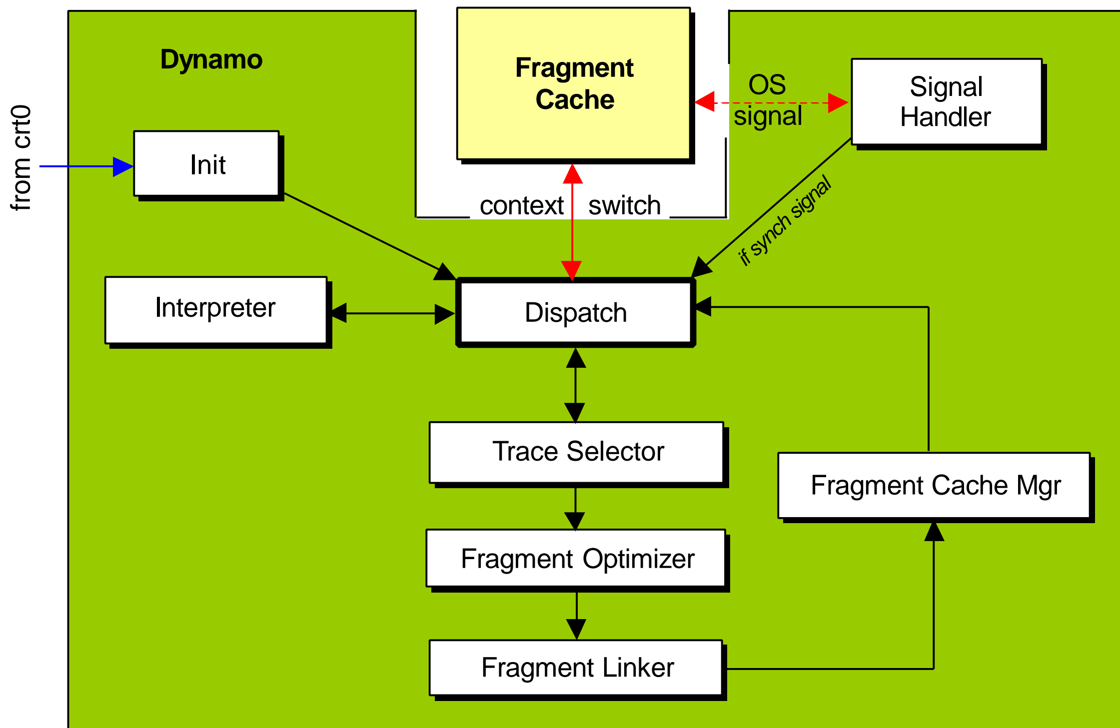


Figure 3: Control flow between the components of the Dynamo system

Dispatch also manages the handshake between the Interpreter and Trace Selector components. When Dispatch is invoked, it first looks up the Fragment Cache for a tag match on the next program PC. If the tag match fails, the Interpreter is invoked. As explained earlier, the Interpreter will interpret until a taken branch, and then return to Dispatch. If the branch target address matches a tag in the Fragment Cache, Dispatch will perform the context switch to transfer control to the cache, else it will invoke the Trace Selector. The Trace Selector is the real "driver" of the Interpreter, controlling the switch that toggles the interpreter's state. The Trace Selector is the place where the actual profiling itself is done. When invoked by Dispatch, the Trace Selector may return immediately after updating a counter associated with the next program PC. Dispatch then invokes the Interpreter again, passing it the return code from the Trace Selector. This Dispatch-Interpreter-Dispatch-Trace Selector-Dispatch loop continues until either (a) a Fragment Cache hit occurs in Dispatch, or (b) a program address gets hot, in which case the Trace Selector

invokes the Fragment Optimizer. The tag information for fragments in the Fragment Cache is maintained in a central data structure in Dynamo memory called the *fragment lookup table*. This is a hash table that is index by a program address, and maps certain program addresses to fragment entry points within the Fragment Cache. Entries are added to this table by the Fragment Linker, just prior to adding a new fragment into the Fragment Cache. The Fragment Cache Manager provides the allocation and deallocation interface to the Fragment Cache, and is responsible for managing the Fragment Cache storage.

5.1 Context switches

At any given moment during program execution under Dynamo, control can either be in the Fragment Cache or in the Dynamo code itself. Thus, two different machine contexts exist, one that corresponds to the execution of Dynamo, and the other to the execution of the dynamically generated code in the Fragment Cache. When control is in Dynamo, the underlying machine registers contain Dynamo's context, and the application program's context is kept in a context save area in the Dynamo data segment. When control is transferred to the Fragment Cache, the application program's context is loaded into the underlying machine registers, but Dynamo's context is discarded. Thus, Dynamo never "resumes" execution when control later exits the Fragment Cache. Control always enters Dynamo at the Dispatch entry point when a Fragment Cache exit occurs. Thus the only part of Dynamo's state that needs to be preserved across multiple Fragment Cache entrances is the values of its global data structures. The part of Dynamo's context that is in the machine registers at the time Dynamo is executing is discarded when control is transferred from Dynamo to the Fragment Cache. Thus, there is only *one* context save area maintained by Dynamo, and it is used only for saving the application program's machine state.

The saving and restoring of the program's context is done by Dynamo and not by the underlying operating system. This keeps the context switch fast. However, it can create a vexing problem on some machines. Since the input to Dynamo is a native program binary, we have to assume that all of the user-visible registers in the underlying machine are used by the program code. Unfortunately, on machines that do not have an absolute addressing mode (as is the case with most RISC instruction sets), a spare register has to be created via spilling to the context save area to implement the saving and restoring of the context.

On some systems, it may be possible to compile the Dynamo source program in a mode that restricts Dynamo code to using only a subset of the general-purpose machine registers. This would allow a more lightweight context save and restore sequence, since only those registers that can be modified by Dynamo code need to be saved in the application program's context save area. Another optimization to lower the context switch overhead is to avoid saving and restoring the floating point state of the machine. Dynamo code does not use any floating point, so saving and restoring of the floating-point registers is redundant.

5.2 Trapping control

Dynamo has to make sure that when a taken branch exits the Fragment Cache address space, control is not lost. This is accomplished by trapping to Dispatch whenever such a branch is executed. Again, it is inefficient to rely on the operating system to trap control (for instance by using the *break* instruction). For every branch that can exit the Fragment Cache, Dynamo generates a unique linker stub into the Fragment Cache [3]. The branch is modified so that it jumps to its associated linker stub instead of the original target. The code in the linker stub does a branch-and-link to a context save routine, which saves the program context to the context save

[3] The linker stubs associated with a fragment are treated as part of the fragment for purposes of storage allocation and de-allocation.

area and then invokes Dispatch. This strategy is similar to that employed in prior dynamic translator implementations, such as Shade [Cmelik and Keppel 1993] and Embra [Witchel and Rosenblum 1996]. Information about the exit branch and its original target address are recorded in a *link record* data structure maintained by the Fragment Linker. The address of this data structure is embedded directly below the branch-and-link instruction in the linker stub corresponding to the exit branch. This "instruction" can never be executed because there is no return corresponding to this branch-and-link instruction.

The location of this embedded link record pointer is automatically communicated to Dispatch via the link register used in the branch-and-link instruction. When Dispatch starts execution, it uses the value of this link register to extract the embedded pointer and retrieve information about the branch that just exited the Fragment Cache. This strategy of directly embedding the pointer in the linker stub avoids the need to implement expensive hashing to achieve the same result. Figure 4 illustrates the scheme used to trap control for branches that would originally have exited the Fragment Cache.

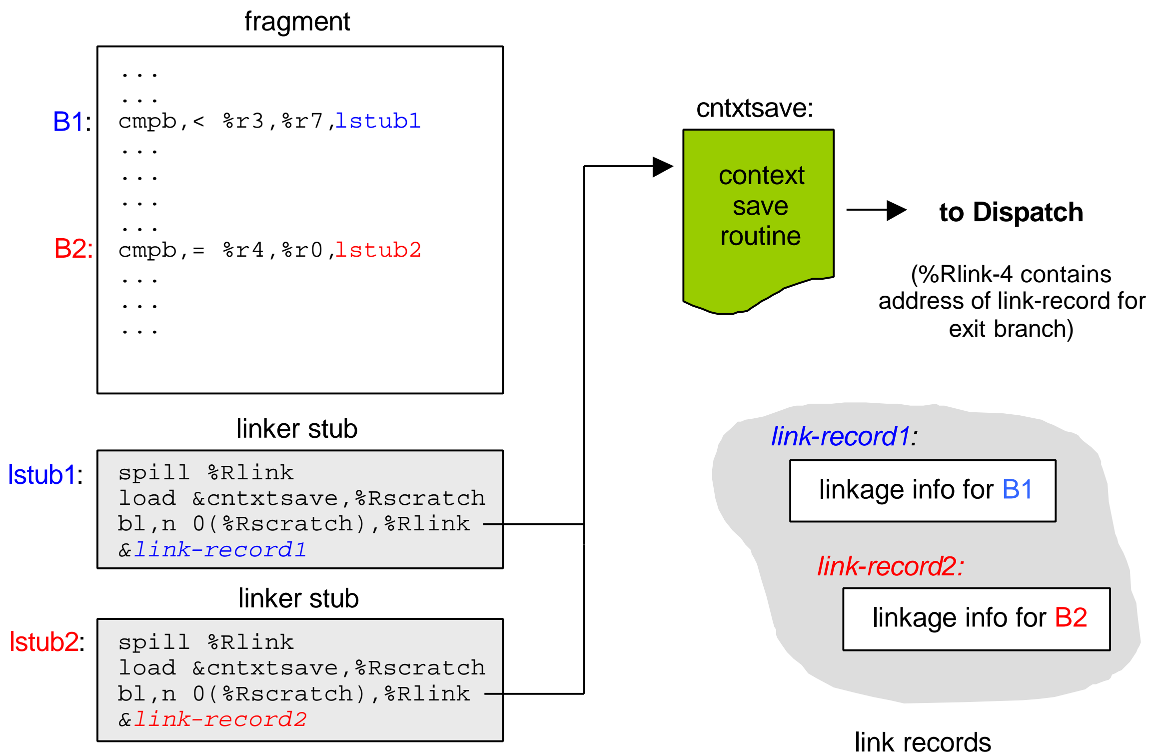


Figure 4: Use of linker stubs to trap control on a Fragment Cache exit

5.3 Signal handling

Once Dynamo gains control over the program, it has to maintain it throughout its execution. Signals create a special problem however. If the application program installs its own signal handler, the operating system will transfer control to that handler (located in the program's address space) when the corresponding signal arrives, causing Dynamo to lose control. To prevent this from happening, Dynamo registers its own signal handler (the Signal Handler component in Figure 3) in place of any program-specified signal handlers. The address of the program-specified handler is recorded in an internal table, but the program's signal handler

installation is suppressed. If the signal occurs during execution, control traps to the Dynamo Signal Handler component first, and the program's handler code is then executed under Dynamo control.

Dynamo itself does not arm any signal, so all signals are available to the program running under Dynamo. Nevertheless, our prototype does not do comprehensive signal handling. For instance, there is no guarantee that signals will not get lost, and that signals arriving while executing the program's signal handler under Dynamo control will be handled properly. Further details about signal handling are discussed in the Signal Handler section later in this report.

6 Interpreter

The Interpreter component in Dynamo is a *native* instruction interpreter by default. The Interpreter is unique among the Dynamo components because it is designed as a plug-in. The default implementation can be overridden with a different one at Dynamo install time, as long as the plug-in interpreter conforms to certain interface requirements. It can even be a non-native instruction interpreter, allowing Dynamo's dynamic optimization technology to be applied in a variety of situations, such as Java JITs or on-the-fly binary decompression in software. The default interpreter can also be replaced with a hardware native interpreter (i.e., the underlying processor) provided it is capable of supporting the necessary plug-in interface requirements.

6.1 Plug-in interpreter interface

The Interpreter's role is to assist Dynamo in profiling the application program for hot spots without having to instrument its loaded binary image, and in collecting the dynamic sequence of native instructions that make up a hot region. The Interpreter supplied to Dynamo must satisfy the following requirements in order to be a valid plug-in:

1. It should transfer control to the Trace Selector every time a taken branch interpreted, passing it the PC of the branch instruction and the PC of the taken target. This is the interpreter's "normal state" of operation.
2. It should provide a mode of operation in which the *native* form of each interpreted instruction is recorded in a buffer, together with the application program PC that it maps to [4]. When operating in this mode, the Interpreter should call the Trace Selector whenever a taken branch instruction is interpreted, passing it the contents of the buffer (or a pointer to it and its current extent). This is the interpreter's "code generation state".
3. Upon return from each of the above calls to the Trace Selector, a return code will specify the state in which the interpreter should subsequently operate.

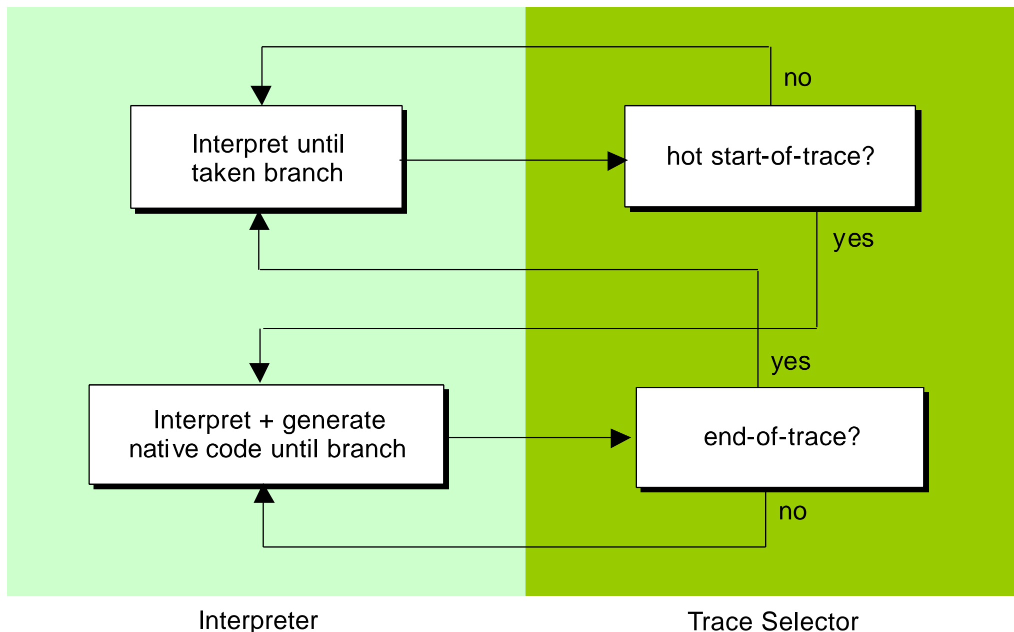


Figure 5: Handshake between the Interpreter and Trace Selector

[4] In the case of a non-native program binary, several native instructions may map to a single program PC.

The Interpreter initially starts up in the normal state. Thereafter, the Trace Selector is responsible for toggling the Interpreter's state via the return code sent back to the Interpreter. The actual profiling itself (i.e., the association of counters with program addresses and incrementing them) is done by the Trace Selector. Figure 5 illustrates the handshake between the Interpreter and the Trace Selector. This handshake is actually done through Dispatch, as discussed earlier in Section 5. Dispatch is only responsible for doing the Fragment Cache lookup upon each return from the Interpreter, and this is not shown in the figure.

The default Interpreter implementation in Dynamo is a straightforward *fetch-decode-eval* loop, where the *eval* part consists of a set of functions written in C to emulate each native instruction opcode. The execution context of the program is updated in Dynamo's context save memory area as a side-effect of interpretation.

6.2 Interpretation via just-in-time translation

An alternative implementation is one that uses direct execution on the underlying processor instead of *eval*. This can be implemented by repeatedly doing the following steps:

1. Copy a block of program instructions into a buffer.
2. Generate additional instructions to trap control when it exits this block of instructions.
3. Load the program context from the context save area into the underlying processor registers.
4. Jump to the top of the buffered block of instructions, effectively suspending the Interpreter, and starting direct execution of the block on the underlying processor.
5. When control exits the block, save the application program context in the underlying processor registers into its context save area, and jump to the interpreter entry point.

Such an interpreter is really using a just-in-time translation mechanism to do the native interpretation. The interpretive overhead involved in these two schemes depends on several factors. Both schemes need to fetch and decode the instructions, the first one for purposes of invoking the appropriate *eval* function, and the second for modifying branch instructions in the block to trap control if they exit the block. The number of interpreter instructions required to evaluate a native instruction in the first scheme is fixed depending on the instruction opcode, and the overall interpretive overhead is directly proportional to the number of instructions interpreted. On the other hand, the second scheme does not involve any "*eval*" step: the instructions are executed directly on the underlying processor. But there is the overhead of the context restore and save to move the program's context between memory and the underlying processor registers upon entry to and exit from the buffered block. In principle, a large enough block of instructions can be fetched into the buffer to offset the overhead of the fetch, decode, context restore and save steps in the second scheme. However, this increases the pressure on the interpreter's fetch mechanism to predict branch directions accurately.

A variation of the second scheme is to make the buffer large enough to hold several blocks. The idea is to treat the buffer as a "transient code cache" that contains code sequences that may become warm, but not quite hot enough to make their way into the main Fragment Cache. This design trades off extra memory space for lower interpretive overhead.

The overall interpretive overhead has a major impact on the design of the Dynamo system. The higher the interpretive overhead, the quicker we want to capture the program's working set within the Fragment Cache, since this code executes directly on the underlying processor. This in turn forces the Trace Selector to be more speculative in its selection of hot spots, since it does not get to observe the program behavior long enough to make a more judicious choice of selecting the hot spots. This has a ripple effect on the Fragment Cache, which now has to compensate for the increased rate of new fragment generation by increasing its size, since frequent replacement can increase Dynamo overhead. The relationship between interpretive overhead (as a percentage of total execution time) and the hot spot selection threshold (how often a program address is encountered before it is considered to be part of a hot region) is shown in Figure 6.

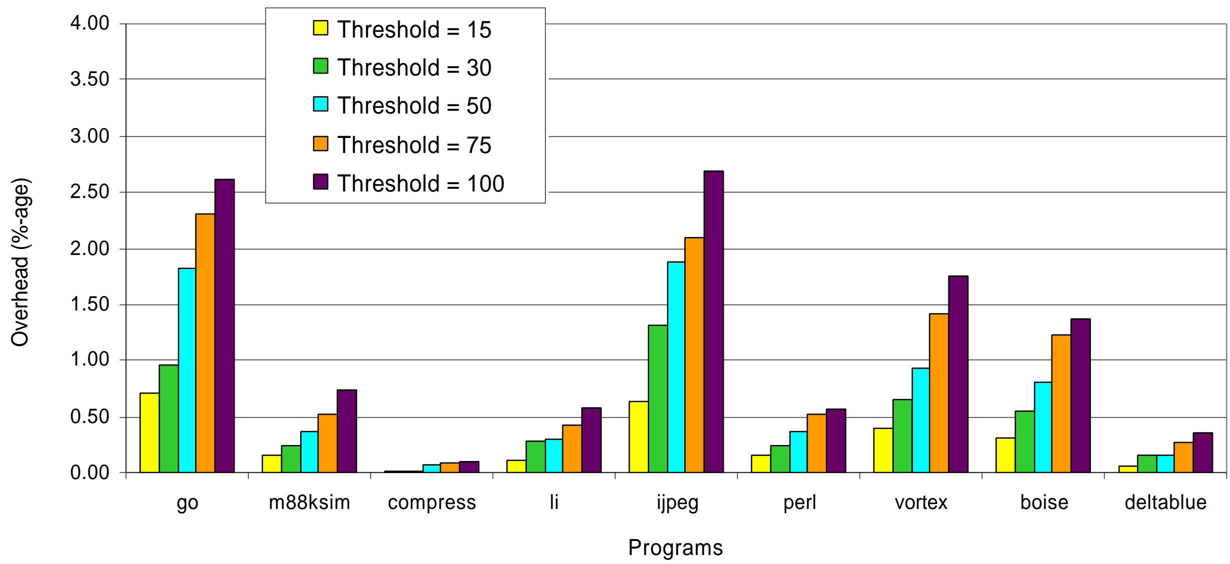


Figure 6: Interpretive overheads

Interpretive overhead can be significantly lowered if the underlying processor (i.e., the native hardware interpreter) can serve as the Dynamo Interpreter. In principle, the underlying processor hardware could itself be used as a plug-in Interpreter, since it is after all an interpreter of the native program code. In order to do so, the hardware may have to be sufficiently enhanced so that it satisfies the Dynamo interpreter plug-in interface. The details of how one might accomplish this are outside the scope of this report.

7 Trace Selector

Dynamic optimization is based on identifying hot spots in the executing program. The Trace Selector extracts hot code regions around the detected hot spots and passes them to the Fragment Optimizer for optimization and eventual placement in the Fragment Cache. The arrangement of the extracted hot code regions in the Fragment Cache leads to a new code layout, which has direct impact on future performance of the hot spot code. Furthermore, by passing isolated hot code regions to the Fragment Optimizer, the Trace Selector dictates the scope and kind of runtime optimization that may be performed by the Fragment Optimizer. Thus, the goal of trace selection is twofold: to improve the *code layout* and to expose *dynamic re-optimization* opportunities.

Few new optimization opportunities or improvements in code layout can be expected if the Trace Selector merely copies static regions from the native binary. Regions such as basic blocks or entire procedures are in the static scope of the native program and were already exposed to and possibly optimized by the generator of the native binary. New optimization opportunities are more likely to be found in the dynamic scope of the executing program. Thus, dynamic control flow should be incorporated into the selected code regions.

The code region of choice in Dynamo is a *partial execution trace* (trace for short). Traces fall into the dynamic scope of the native program and directly represent dynamic control flow information. A trace is a dynamic sequence of executed basic blocks. The sequence may not be contiguous in the program memory; it may even be *interprocedural*, spanning one or several procedure boundaries, including dynamically linked modules. Thus, traces are likely to offer opportunities for improved code layout and re-optimization. Furthermore, traces do not need to be computed; they can be inferred simply by observing the runtime behavior of the program through interpretation.

Some profiling is necessary to differentiate hot code region from cold ones. Profile information is collected as a side effect of interpretation, and trace selection is triggered only when a hot program address is interpreted. After selecting a hot trace, the Fragment Optimizer transforms the trace into a location-independent, contiguous code sequence called a *fragment*, which is later placed in the Fragment Cache for execution.

7.1 Online profiling

The first task of trace selection is the on-the-fly detection of hot spots. Hot spot detection in the Dynamo context is challenging for two reasons. First, instead of detecting individual hot program points, entire trace information is needed; and second, the profile information is needed *online*. Online profiling is difficult since the profile information is inherently predictive. Hot spots must be detected as they are becoming hot, not afterwards as in conventional offline profiling applications. Furthermore, the profiling techniques must have very low overhead to be of any use as an online technique.

Previously, profiling has commonly been used in an offline manner in feedback systems, such as in profile-based optimization. In such an offline-profile feedback system, a separate profile run of the program is necessary to gather the accumulated profile information that is then fed back to the compiler. There are two conventional approaches to offline profiling: statistical PC sampling and branch or path profiling through static binary instrumentation. Statistical PC sampling [Anderson et al. 1997, Ammons et al. 1997, Conte et al. 1996, Zhang et al. 1997] is an inexpensive technique for identifying hot code blocks by recording program counter hits (i.e., PC samples) at regular time intervals. While PC sampling is efficient for detecting individual hot blocks it provides little help in finding entire hot paths. One could build a hot path by stringing together the hottest blocks, but the individual blocks in such a path may have been hot along entirely disjoint program paths and the resulting path may never execute from start to end.

Another problem with statistical PC sampling is that the timer interrupt signal required to implement it may interfere with programs that arm this signal with a custom handler.

Profiling techniques through binary instrumentation are accurate rather than statistical. However, *branch profiling* techniques suffer the same problem of having to build entire paths from individual branch or block frequencies. Recently proposed *path profiling* techniques offer a way for directly determining the hot paths [Ball and Larus 1996]. The program binary is statically instrumented prior to execution, to collect path frequency information at runtime in an efficient manner. Such path profiling techniques are primarily intended for post-mortem construction of hot path information. The large amount of profile information collected by these techniques, and the overhead of the analysis required to process it, makes them unsuitable for use in a runtime system. Furthermore, instrumenting the program binary perturbs the original program, making it difficult to reach our transparency goal.

Dynamo follows a different approach that is suitable for online profiling. In contrast to traditional profiling methods, the program binary is not instrumented. Rather, interpretation of the program instructions is used to observe its behavior without instrumentation. The profile information gathered via interpretation is consumed as it is produced rather than being used as feedback for a later run.

7.2 Speculative trace selection (SPECL)

Dynamo implements a new lightweight profiling strategy by limiting the program points for which execution counts are maintained. The special program points that mark the potential entry of a hot code region are called *trace heads*.

Trace heads are determined as the program executes as the program points that satisfy a *start-of-trace* condition. Since frequently executing code generally involves looping in some form, the start-of-trace condition is satisfied for any program address that is reached via a backward taken branch. Furthermore, hot code regions may extend beyond individual traces. Thus, the targets of branches that exit a previously selected hot trace also meet the start-of-trace criterion. The Trace Selector profiles trace head execution by associating a counter only with the dynamically encountered trace head.

When the Interpreter operates in its normal mode, it always calls the Trace Selector just prior to interpreting the instruction at the start of a trace head. The Trace Selector then increments the counter associated with the trace head, and if the counter exceeds a preset threshold, it toggles the Interpreter's state to trace generation mode (see Figure 5). Since only trace heads are profiled, Dynamo employs a *speculative selection* scheme to form hot traces. The idea here is that when a trace head becomes hot, the sequence of blocks executed immediately following the trace head is statistically likely to be hot. A trace can therefore be grown starting from the hot trace head by simply collecting the subsequently interpreted basic blocks in a history buffer. The scheme is speculative, because it does not use any path profiling information to select the trace.

Once the Trace Selector detects a hot trace head, it toggles the Interpreter's state so that it now operates in a trace generation mode. When executing in the trace generation mode, the Interpreter calls the Trace Selector after each basic block is interpreted. The Trace Selector uses this opportunity to record the sequence of basic block instructions in a history buffer. History collection terminates when the Trace Selector detects an *end-of-trace* condition. An end-of-trace condition occurs when (1) the next block to be executed is identical to the first block in the history buffer (i.e., a cycle), or (2) the current instruction is a backward taken branch (i.e., a new start-of-trace), or (3) the history buffer has reached a size limit.

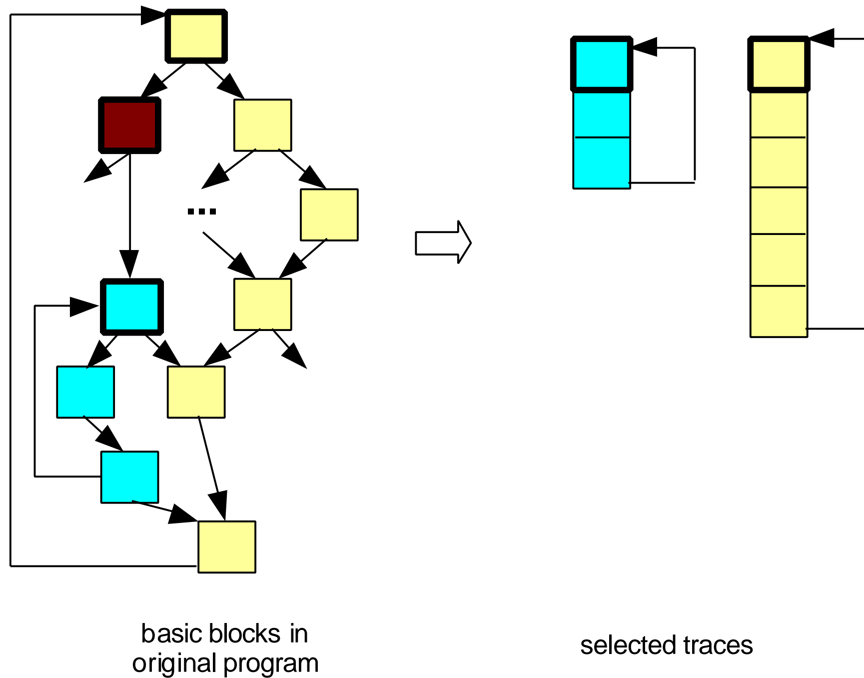


Figure 7: Dynamo's trace selection scheme with trace heads shown in bold

When the end-of-trace condition is satisfied, the trace currently in the history buffer is handed to the Fragment Optimizer. The Fragment Optimizer re-lays out the blocks into a straight-line sequence as shown Figure 7. Finally, before returning to the Interpreter, the Trace Selector toggles its state so that the Interpreter resumes its normal mode of interpretation. At this point, the counter associated with the current trace head is recycled [5].

Dynamo's speculative trace selection scheme is not only simple, but highly effective as well. This might seem surprising since no path profiling information is used to select the hot traces. The intuition behind the scheme's success is that in program's that have few dominant hot paths the SPECL scheme is statistically very likely to pick one of the dominant paths. On the other hand in program with several competing hot paths, for example program with several hot paths through the same loop, it is not that important which of the competing paths will be picked since the penalty for not picking the truly hottest path cannot be very high.

In order to empirically evaluate the effectiveness of our scheme, we compared it against the following three alternative selection schemes. A *basic block selection* scheme, which serves as the base case, a *static selection* scheme that is based on static branch prediction, and a *two-level hierarchical selection* scheme that enhances the static strategy with dynamic execution history information. The three schemes use progressively more sophisticated profiling schemes to detect hot traces. These schemes are discussed here only as a comparison, and in the rest of the paper, unless specifically stated otherwise, the speculative selection (SPECL) scheme should be assumed.

7.2.1 Basic block selection (BLOCK)

A basic block scheme is one in which a trace is simply a single dynamic basic block. This is the profiling unit used in early implementations of dynamic translators. A *dynamic basic block* is

---

[5] This is a significant advantage of consuming the profile data on the fly. It is difficult for a static binary instrumentation scheme to mimic this because the profile data is only consumed at the end of the run.

a straight-line code sequence that starts at the target of a branch and extends up to the next branch. Note that unlike conventional static basic blocks, a dynamic basic block may extend across join points. Placing every hot dynamic basic block as an isolated fragment in the Fragment Cache is unlikely to lead to an improved code layout, and hence performance improvement. However, we include basic block selection as a comparison basis for more advanced trace selection schemes.

7.2.2 Static trace selection (STATIC)

The static trace selection scheme uses static branch prediction rules [Ball and Larus 1993] to grow traces. Unlike Dynamo's SPECL scheme, the STATIC scheme considers every target of a taken branch as a potential trace head. Correspondingly, execution counts are maintained at every branch target during interpretation. After a branch target has become hot, a static trace is grown as a sequence of basic blocks by repeatedly applying one of the following static branch prediction rules until a branch is encountered for which no rule applies:

(R1) *Static Branch Rule:* If the branch condition can be statically determined (i.e., by decoding the branch instruction or its immediate context), the branch is predicted according to the static evaluation of the branch condition.

(R2) *Ball/Larus Rules* for predicting branches that compare a value against zero: According to Ball and Larus many programs use negative values to indicate errors [Ball and Larus 1993]. Hence, branches whose branch condition is <= 0 are predicted as not taken.

(R3*) Unconditional Branch Rule*: Unconditional branches are predicted as taken.

On the PA-RISC, a compare and unconditional branch pair is treated as a conditional branch if the unconditional branch can be nullified as a result of the compare operation. Such a branch is treated as an ordinary conditional branch where the complement of the compare condition in the nullifying instruction serves as the branch condition.

The above rules apply only to direct branches with known branch targets. Indirect branches with dynamic targets are more difficult to predict. The prediction problem is not one of determining the branch outcome, but rather one of determining the branch target. Procedure returns are a special class of indirect branches that can be predicted by partial procedure inlining. To implement partial inlining, the Trace Selector maintains a *return address stack* as it constructs the trace in its history buffer. When a procedure call is encountered on the trace, its corresponding return address is pushed on the stack. If a return indirect branch is encountered later on the same trace, the following rule is used:

(R4) *Inlining Rule*: If there is a return address on the stack, pop the stack and predict the return address as the target of the branch.

Note that it generally cannot be determined for sure whether a particular branch represents a procedure call or a procedure return. This is because the input to Dynamo is an unprepared binary, so it may contain hand-coded instruction sequences that use return branches in non-standard ways. Thus, the actual partial inlining of the return branch will not take place until later during optimization when the code is carefully analyzed. If the return pointer (link) register is not modified on the trace after the procedure call branch, optimization analysis will reveal that the return branch is redundant. Removing the redundant branch will complete the partial procedure inlining. Note that the inlining rule is safe and will not lead to incorrect execution sequences. The worst that can happen is that a redundant inlined return target will never execute.

The above branch prediction rules are local heuristics, since each branch is predicted in isolation based only on its immediate context. Local prediction heuristics cannot detect branch correlation, where the outcome of one branch strongly implies the outcome of a subsequent branch [Pan et al. 1992]. Branch correlation may be *semantic*. For example, if a branch with condition "X > 10" is taken, a subsequent branch with condition "X > 0" must be taken as well, if the value of X has not changed in the interim. Branch correlation may also be *statistical* based on

observed previous branch outcome sequences [Young and Smith 1994]. For example, if the outcome of one branch is usually followed by a certain outcome of another branch, the two branches may be statistically correlated. The set of prediction rules can easily be extended to include branch correlation rules.

The appeal of the STATIC scheme lies in its simplicity. If the STATIC scheme could pick traces as effectively as the SPECL scheme, it would be preferable because it involves lower interpretive overhead. The STATIC scheme would not require running the Interpreter in the trace growing mode, since it would be able to predict the basic blocks of the trace without resorting to interpretation. On the other hand, the trace growing mode of interpretation allows a trace to be grown past indirect branches, whereas the STATIC scheme cannot do this without a significantly greater profiling cost. Thus, a major drawback of the STATIC scheme is its inability to make useful predictions for indirect branches that cannot be inlined.

7.2.3 Two-level hierarchical selection (2-LEVEL)

The hierarchical history-based scheme addresses the shortcomings of the static trace selection scheme by enhancing it with a dynamic history collection mechanism that would allow it to construct traces past indirect branches. Trace selection takes place in a hierarchical fashion as illustrated in Figure 8. Static trace selection is used as a first-level selection scheme to construct *trace components*. The trace components are themselves subject to secondary trace selection by observing their dynamic interactions. A secondary trace is grown from trace components through history collection, which provides a natural way to extend the static trace components across indirect branches.

To enable secondary trace selection, the Trace Selector operates in one of two modes: a base selection mode and a secondary selection mode.

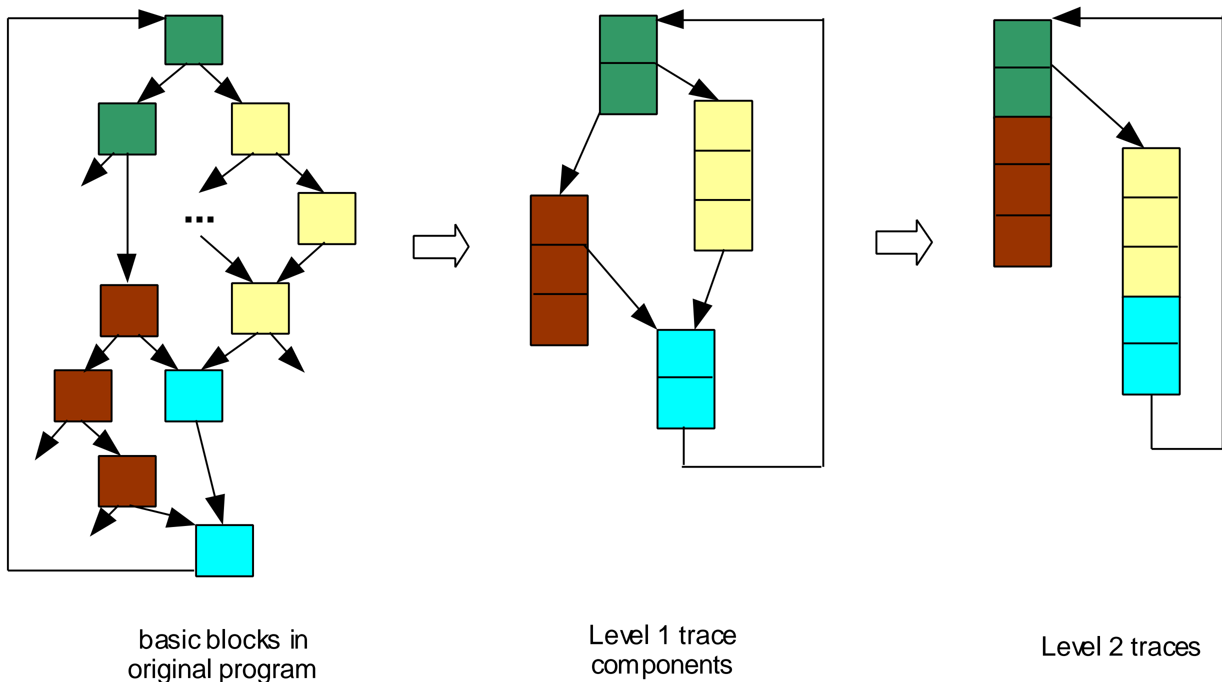


Figure 8: 2-Level hierarchical selection (2-LEVEL)

7.2.3.1 Base selection mode

In base selection mode, first-level static trace components are selected using prediction rules starting at a hot trace head, as described in the previous section. To trigger secondary trace selection, execution frequencies are tracked for a subset of these components: the *secondary trace heads*. A secondary trace head is any trace component that is the target of a backward taken branch from another component or the target of any branch that exits from a secondary trace. In order to update the execution count of a secondary trace head, direct linking of branches to secondary trace heads is suppressed, so that the Trace Selector always gains control prior to each secondary trace head execution. As an optimization, the counter update code upon entry to a secondary trace head can be embedded as part of the fragment prologue code. In our experiments however, this did not make a significant difference to the result.

7.2.3.2 Secondary selection mode

If the execution count at a secondary trace head exceeds a preset threshold, the Trace Selector's mode changes to secondary selection mode. As in the Dynamo selection scheme, a secondary trace is built by collecting the subsequently executing components in a fixed sized history buffer. A pointer to each executing component is entered into the history buffer along with a pointer to the branch that exited from that component. History collection terminates when the end-of-trace condition has been met as discussed for the SPECL scheme. When history collection terminates, the history buffer contains a sequence of trace components along with their exiting branches. The exit branches accurately dictate how much to include from each component to construct the final trace.

This section assumes that static trace selection is used to provide the first-level trace components. Conceptually, the origin of the components is irrelevant and the first level component may equally well be based on any other scheme, such as the BLOCK scheme.

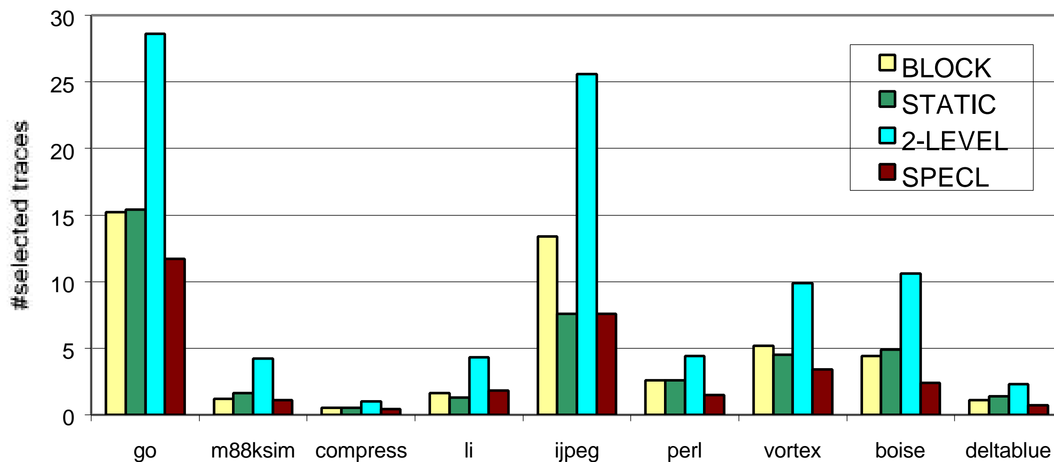


Figure 9: Average trace selection rate

7.3 Trace selection rate

The trace selection rate is the number of traces selected per second. It is measured using the Dynamo system profiler, discussed later in this report. The trace selection rate together with the Fragment Cache hit rate (see Chapter 9) provide a useful measure to estimate how well different

selection schemes capture working sets for the same program. Intuitively, the lower the trace selection rate, the lower the fragment creation overhead.

Trace selection rates are highly program-dependent as shown in Figure 9. Programs that spend most of their time in a small region of code will have a relatively low selection rate, regardless of the selection scheme. On the other extreme, programs that have a constantly changing working set exhibit excessively high selection rates.

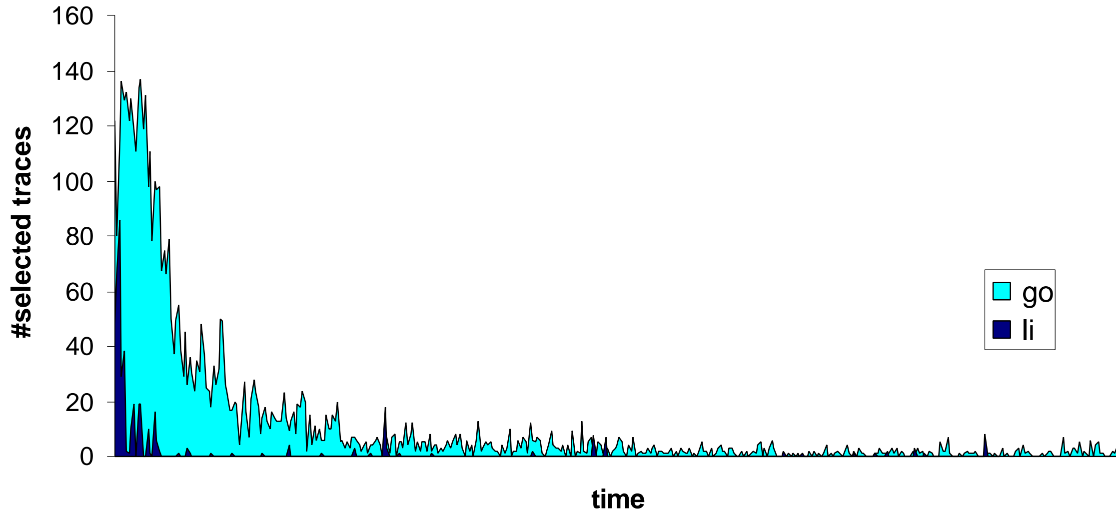


Figure 10: Trace selection rate for go and li

Figure 9 shows that in all cases, the 2-LEVEL scheme exhibits the highest trace selection rate. This is to be expected, because of its two-level selection approach. The selection rate of the STATIC scheme is almost the same as for the BLOCK scheme. This suggests that the prediction quality of our current heuristics to statically predicated traces may not be very high. The SPECL scheme exhibits a consistently lower trace selection rate in all cases.

The trace selection rate varies considerably during program execution. The selection rate peaks each time the program enters a new execution phase. Figure 10 displays the trace selection rates over time for go and li based on the SPECL scheme. For both programs, the trace selection rate is very high in the beginning when the program's initial working set is being built in the Fragment Cache. In li however, the selection rate decreases rapidly after the initial peak, and is barely discernable through the remainder of execution. This indicates that in the case of li, much of its working set has been captured in the Fragment Cache during the initial peak in the trace selection rate. In contrast, the selection rate for go does not decrease as rapidly, and remains at a noticeable level throughout its execution.

7.4 Bailing out

Excessively high trace selection rates, such as in go, create a significant fragment creation overhead. Clearly, if more time is spent selecting traces and less time in reusing the optimized code within the Fragment Cache, the benefits of trace selection and fragment optimization is lost. The longer the trace selection rate remains high, the less likely it is that Dynamo can "catch up" by offsetting the selection overhead through higher performance execution of the dynamically optimized fragments. Our benchmark set contains three programs with excessive trace selection rates: go, ijpeg and vortex. In these programs, the continuously high overhead of trace selection

and fragment creation will eventually cause a performance slowdown as shown under the "SPECL without bail-out" category in Figure 11.

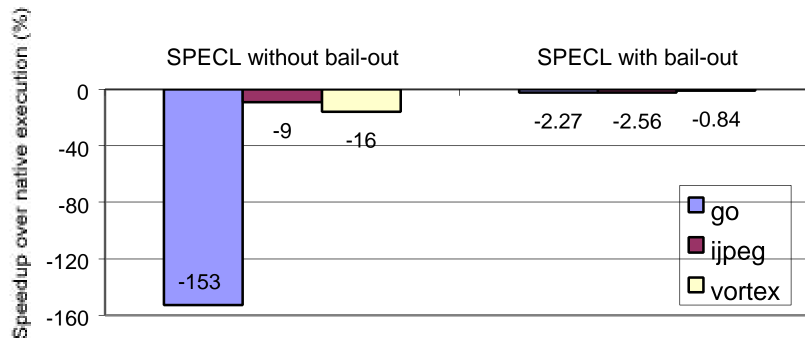


Figure 11: Overall Dynamo speedup based on SPECL trace selection without and with bail-out

To prevent such performance degradation, Dynamo keeps track of the current trace selection rate. The trace selection rate is measured by the time interval needed to select *n* traces: the shorter the interval, the higher the trace selection rate. If the interval is shorter than a preset threshold the interval is marked as a high rate interval. The overall trace selection rate is considered *excessive* if *k* consecutive high rate intervals have been seen. The parameters *n* and *k* are set at 35 and 4, respectively in our current prototype.

To preserve deterministic behavior, Dynamo does not actually rely on real time for tracking the trace selection rate. Instead, a notion of virtual time is used. Elapsed virtual time is measured by the number of interpreted basic blocks, since interpretation is the primary function performed whenever control is in Dynamo.

If the trace selection rate is determined to be excessive, further trace selection and optimization is disabled and Dynamo *bails out*, that is, control is relinquished and execution continues in the unaltered original program. The parameters are chosen so that an excessive trace selection rate can be detected very early, resulting in minimal performance loss on such programs (as shown under the "SPECL with bail-out" category of Figure 11). The downside of this is that some programs may exhibit a trace selection rate that appears excessive in the beginning, but is then low enough for Dynamo to provide a performance boost, but Dynamo bails out prematurely.

7.5 Evaluation

The previous sections introduced three different alternatives to Dynamo's speculative trace selection scheme. Each scheme will eventually materialize the hot code in the Fragment Cache. However, the various schemes will produce sets of traces that differ in three important ways: the *number* of traces selected, the amount of *pollution* and the amount of *code duplication* among the traces.

Keeping the number of traces low is important in order to control the fragment creation and optimization overhead. Pollution occurs when cold code fragments are inserted into the Fragment Cache (due to inadequate profile data). Depending on the degree of speculation used in trace formation, it is possible that the leading part of a trace is much hotter than the terminating part. Thus, execution may frequently exit (or mispredict), from a trace prior to reaching the tail end of the trace. Cold trace tails pollute the Fragment Cache contents by negatively impacting locality. The amount of pollution is governed by the prediction accuracy of the trace selection scheme. For

example, the BLOCK scheme uses no speculation. Thus, every fragment entered into the Fragment Cache is guaranteed to be initially hot and pollution can only be caused by fragments that are no longer hot because the program's working set has changed.

Code duplication occurs when a trace is grown across a block that has already been included in another trace. For example, in a program that calls the same procedure from multiple call sites, the amount of duplicated procedure path can be substantial. The amount of code duplication depends on both the program behavior and the selection scheme. The selection scheme can control the amount of code duplication by limiting the trace length: the shorter the trace the fewer the blocks that can be duplicated. However, while a high amount of code duplication may negatively impact locality in the Fragment Cache, code duplication is crucial in exposing new optimization opportunities.

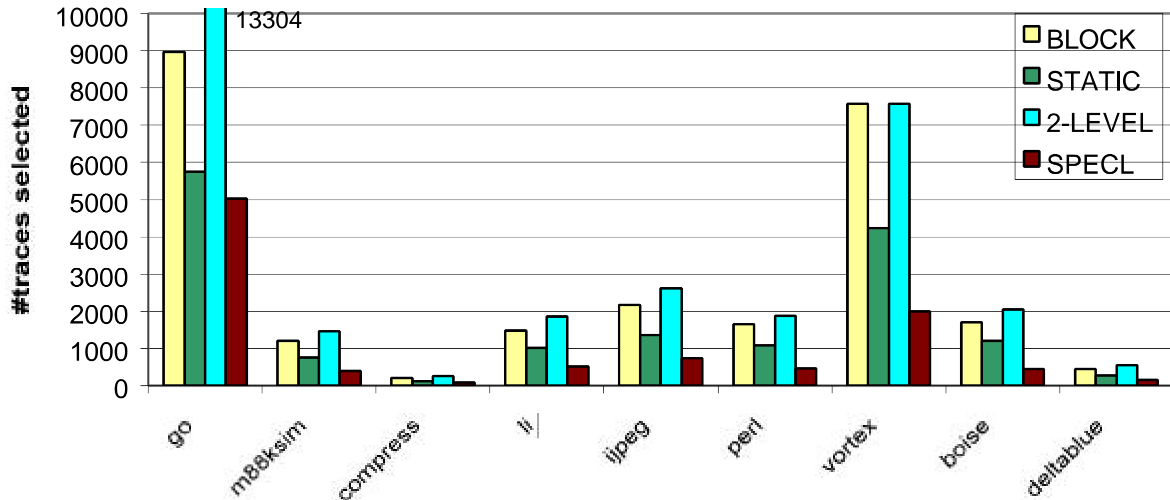


Figure 12: Number of traces selected by each selection scheme

Figure 12 shows the number of traces selected by each scheme for a set of benchmark programs (without the bail-out option). The 2-LEVEL scheme triggers the highest number of trace selections due to its two level selection approach. The BLOCK scheme, which selects the shortest traces, also creates a large number of traces. An important factor for the number of traces selected is the prediction quality of a selection scheme. The better future branch outcomes are predicted the fewer the number of early exits from the trace. Hence, fewer additional traces are triggered through early exits. Once again, the SPECL scheme consistently selects the lowest number of traces.

Figure 13 shows the accumulated fragment size in KBytes after all traces have been selected. The accumulated fragment size varies highly with each scheme. The BLOCK scheme produces the smallest accumulated fragment size since it causes no duplication. Comparing that with the other selection schemes provides insights about the amount of code duplication caused by each scheme. Since statically selected traces do not extend beyond indirect branches other than inlined procedure returns, static traces tend to be shorter than 2-LEVEL or SPECL traces. Shorter traces tend to cause less duplication and less pollution and hence lead to a more compact representation.

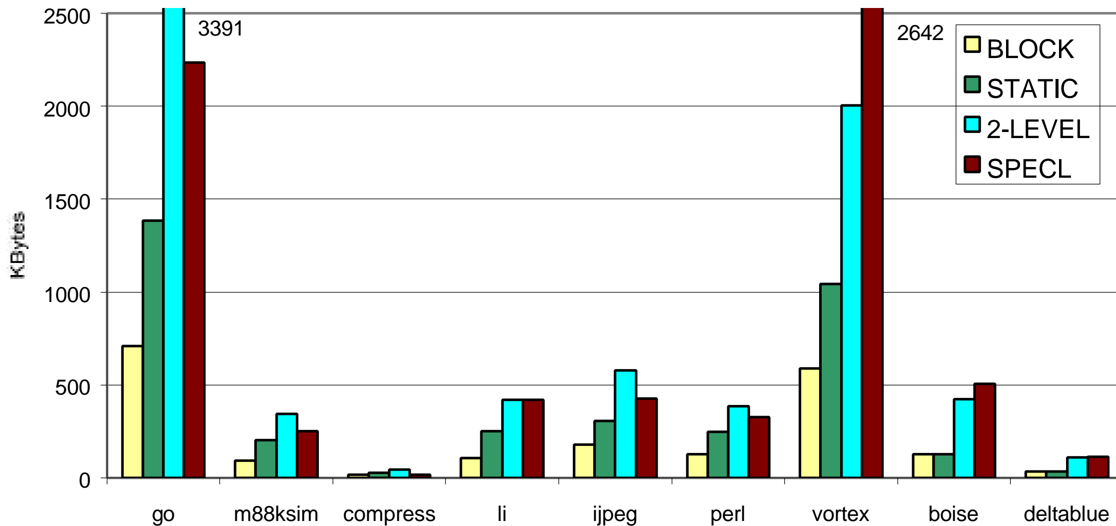


Figure 13: Accumulated fragment size in Kbytes for each selection scheme

7.6 Trace quality metric

While the compactness of a trace selection scheme has an impact on the locality of execution in the Fragment Cache, compactness alone does not directly translate into an effective trace selection policy. For example, even though the BLOCK scheme leads to the most compact selection, it cannot be expected to lead to an improved code layout compared to the original program binary, because it exposes far fewer optimization opportunities. Thus, while trace selection rates are useful in assessing the overhead and predictive power of a selection scheme, they do not reveal anything about a trace's potential for optimization.

In order to assess the quality of traces selected by a given trace selection scheme, we need a metric that combines these individual aspects of trace selection. A reasonable metric is one that focuses on the composition of the hottest portion of the Fragment Cache contents, by inspecting the *X% cover set* of a selection scheme. For a given percentage *X*, the *X*% cover set is defined as the smallest set of fragments whose combined execution percentage exceeds X. Two different trace selection schemes will lead to different cover sets; the number of branches and the amount of code duplication may vary from one cover set to the other. Generally, the lower the number of branches in a cover set the better the expected performance. Consider for example an extreme case where a program spends all its time in a single large loop of straight-line code. Now assume one selection scheme selects the entire loop as a single trace while another breaks the loops into several traces. Clearly, the former scheme is more likely to outperform the latter.

Figure 14 shows 90% cover sets for the programs on which Dynamo does not bail out. The Figure shows that the SPECL scheme requires the lowest number of traces to cover 90%, followed by the 2-LEVEL scheme, the STATIC scheme and finally the BLOCK scheme.

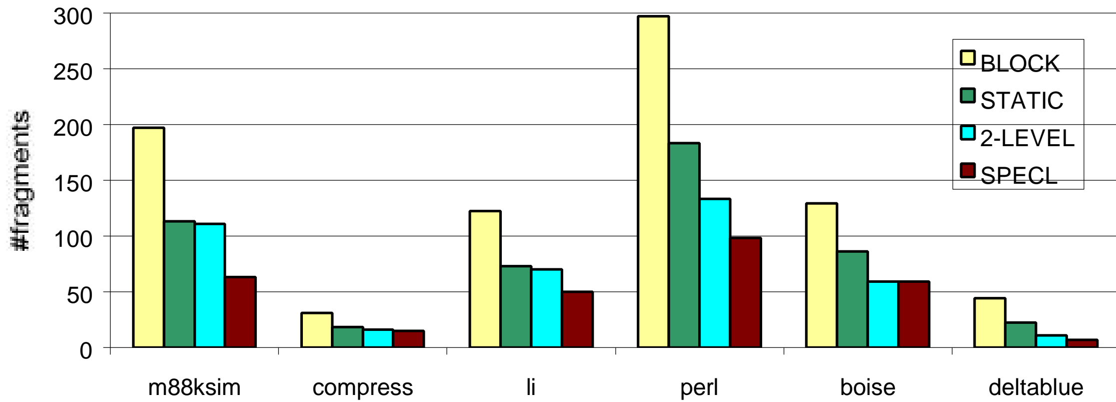


Figure 14: 90%-Cover sets

To evaluate the effectiveness of cover sets in predicting performance we compare the cover set data with the actual runtime performance (without bail-out). Figure 15 shows the speedup of running Dynamo with each trace selection scheme over native execution time. The measurements are based on running Dynamo in default mode, which includes fragment optimization. Thus, the speedups in Figure 15 include the speedup due to optimization. Details on our experimental framework are presented in the Performance Summary chapter later in this report.

As the performance data shows, performance is closely related to the cover set sizes. In all cases, if one scheme yields a higher speedup it also produces a smaller cover set. The BLOCK scheme clearly performs worst and causes a slowdown in all programs. The STATIC scheme performs much better and even achieves a slight speedup in compress. The performance is further improved by the 2-LEVEL scheme, which yields speedups in m88kim, compress, li, perl and deltablue. Once again, the best performer is clearly Dynamo's SPECL scheme, yielding speedups of over 20% in m88ksim and li.

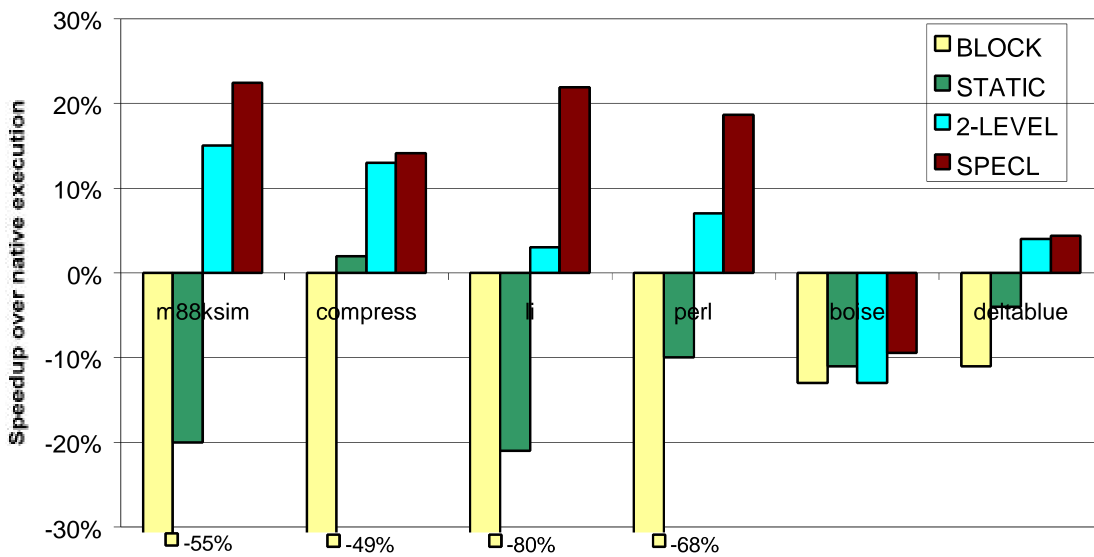


Figure 15: Overall Dynamo speedup with varying selection schemes

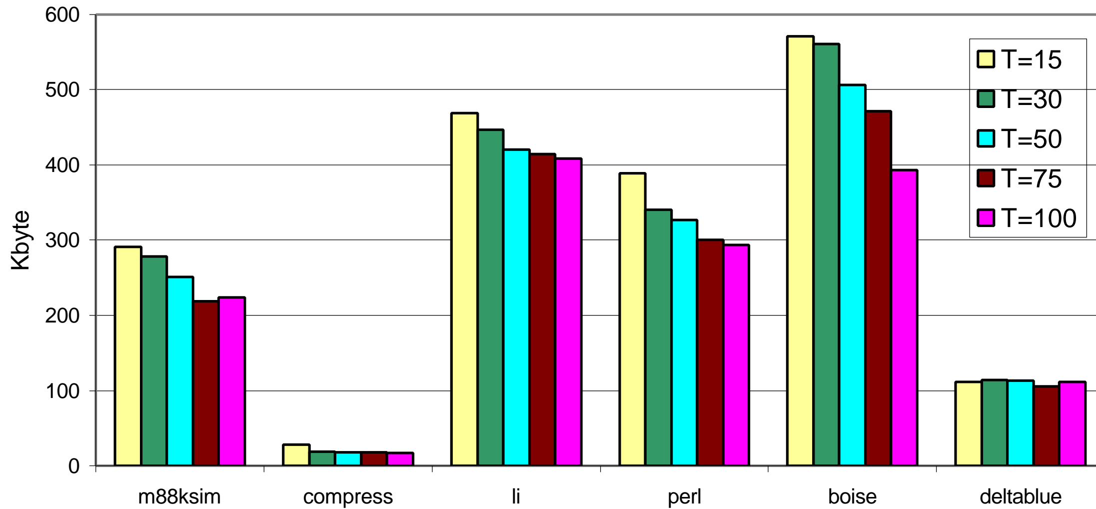


Figure 16: Accumulated fragment size with varying threshold T

7.7 Selection threshold

A remaining issue is the choice of the execution threshold threshold value that triggers a trace selection in the Dynamo scheme. The higher the threshold the fewer the program points that qualify as hot and the fewer fragments that are selected. Hence the lower the amount of pollution and the better the locality in the Fragment Cache. Thus, it appears the higher the threshold the more effective the trace selection scheme. However, a higher threshold also means a higher interpretive overhead. Furthermore, the later hot code is recognized, the lower will be the re-use benefits of the optimized code in the Fragment Cache.

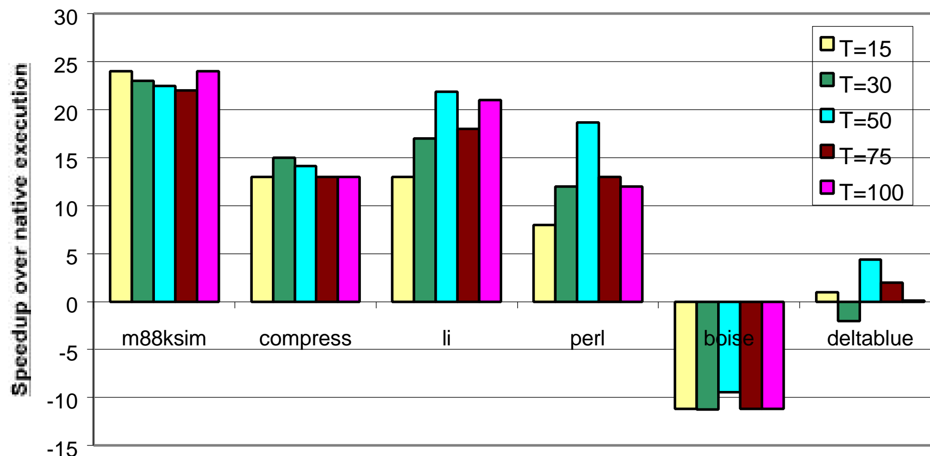


Figure 17: Overall Dynamo speedup with varying trace selection threshold for SPECL

We experimented with varying threshold values ranging from 15 to 100 to get a better understanding of this trade-off. Figure 16 shows how the accumulated fragment size changes with the threshold value. As expected, in almost all cases the higher the threshold the smaller the accumulated fragment size.

Figure 17 shows how the overall speedup changes with varying threshold. Except for m88ksim, the best speedup results at a threshold value of 50. Thus, we set the default threshold value in Dynamo at 50.

8 Fragment Optimizer

The Fragment Optimizer is invoked immediately after trace selection to produce an improved sequence of instructions by applying a set of optimizing transformations. The Fragment Optimizer is not intended to duplicate or replace conventional static compiler optimization. On the contrary, the Fragment Optimizer complements the static compiler by exploiting optimization opportunities that either arise only at runtime or that are too expensive to fully exploit statically. An example of the former is optimization across dynamically linked (shared library) calls. Examples of the latter are path-specific optimizations and, in particular, call-specific optimization. The considerable amount of code duplication that is necessary to perform path-specific optimizations statically usually limits their application at compile-time. However, in Dynamo the mere process of trace selection automatically exposes new path-specific optimizations. Since the Fragment Optimizer is handed individual traces, any optimization on these traces is essentially a path-specific optimization.

While trace selection simplifies the problem of path-specific optimization there are many other challenges that make runtime optimization difficult. Since the Fragment Optimizer can only see a small code window at a time, the scope of optimization is far more limited than at compile-time, where the optimizer can see entire procedures or even the whole program. On the other hand, considering only a small window of code also has its advantages. A code fragment built from a trace has very limited control flow. In particular, fragments do not contain internal join points. The absence of join points greatly simplifies analysis and optimization algorithms including algorithms for register allocation.

Another challenge for the Fragment Optimizer is that it operates on executable code. The input code to the Fragment Optimizer is fully register allocated and may have previously been optimized statically. Valuable semantic source-level information that is utilized by a static optimizer, such as type information, is not available and generally difficult to reconstruct. Finally, unlike an optimizing compiler, the Fragment Optimizer operates under severe time constraints forcing it to focus on only the most cost-effective optimizations.

To lower optimization overhead, the Fragment Optimizer design departs from the standard phased approach for compiler optimization. Instead of pre-computing data flow information exhaustively in a separate phase, the Fragment Optimizer retrieves the necessary data flow on-demand in an interleaved design so as to avoid the computation of information that is never used [Duesterwald et al. 1995].

The Fragment Optimizer operates in a few non-iterative passes in either a forward or backward direction. During each pass a set of optimizations is applied and data flow information is retrieved as needed. Since no iteration is required and data flow information is maintained only for the currently considered statement, both time and space requirements of the optimization passes are linear in the size of the fragment and the fragment resources (i.e., registers).

When passed a new fragment the Fragment Optimizer performs all applicable optimization in a single optimization stage. We also considered an alternative *staged design* with assigned time budgets for the Fragment Optimizer. In a staged design, the applicable optimizations are divided into several optimization stages based on their expected cost-benefit ratios. Optimizations are applied in a pipelined fashion progressing from one stage to the next by starting with low overhead or high-gain optimizations. Each fragment is assigned a time budget and optimization proceeds to the next stage only if the assigned time has not yet exceeded. If optimization is terminated due to an expired time budget remaining optimization stages are either ignored or delayed to a later point when the fragment has proven in some way to deserve further optimization. The time budgets may be fixed or variable. Variable time budgets may depend on fragment characteristics, e.g. loops may get higher time budgets than non-looping fragments.

Time budgets may also be based on the current state of Dynamo. For example, during times with high fragment selection rates the time budgets may be lowered.

While time budgeting in combination with a staged optimizer design appears a promising approach to control optimization overhead, it turned out that overhead control was actually not a critical issue for Dynamo. As will be shown later in this section, the complete optimization overhead is extremely low and is in most part caused by mandatory optimization passes such as fragment formation and register allocation. Thus, staging the set of application optimizations would have no noticeable effects in this context.

8.1 Dynamic optimization opportunities

The Fragment Optimizer applies a number of classical optimizations that are commonly performed in a static optimizing compiler, such as redundancy elimination and strength reduction. It might appear unlikely that many opportunities for these kinds of optimization can be found at runtime if the executing program was already optimized statically. However, there are important reasons why the Fragment Optimizer can still find classical optimization opportunities, even if the executing program was compiled at high levels of static optimization. The following sections present examples taken from actual traces selected by Dynamo based on -O2 compiled versions of the benchmark programs.

8.1.1 Exposed partial redundancies

The process of trace selection can be viewed as the dynamic counterpart to procedure in-lining and superblock formation [Hwu et al. 1993]. Both inlining and superblock formation have the effect of removing join points in the code, thereby enabling path-specific optimizations that would otherwise be unsafe. A static compiler generally cannot afford to perform inlining and superblock formation exhaustively over the whole program due to the prohibitive code growth. Moreover, in the case of dynamically linked libraries, inlining is not even an option for a static optimizer. In contrast, a dynamic optimizer can further exploit the optimization potential since dynamically linked code is available and inlining and superblock formation occurs automatically through trace selection.

A class of path-specific optimization opportunities that are exposed by inlining and superblock formation is partial redundancy elimination. A load is partially redundant if it is fully redundant on some program path, but not necessarily along all paths. A number of complex algorithms have been developed to remove partial redundancies by statically transforming each partial redundancy into a full redundancy through code motions and code duplication [Morel and Renvoise 1979; Knoop et al. 1994; Bodik et al. 1998]. Since static elimination is very expensive if applied over the whole program partial redundancies are likely to remain in the code in the presence of complex control flow or across procedure calls.

By extracting traces from the code, the code duplications needed to transform partial redundancies into full redundancies takes place automatically. In particular, the implicit inlining of procedure calls during trace selection exposes redundancies across procedure calls. For example, redundant register saves and restores around the call. The original partial redundancies are easily detected and eliminated as full redundancies on the seletcted trace.

Consider the object code below for procedure *mrglist* from the benchmark go. The instructions shown in bold present a particular trace that was selected across several taken branches (i.e., the instruction sequence 144-148, 208-224, 316, 336-348). The underlined load at 336 is redundant on this trace since the load is preceded by an identical load at 144 and register r23 is not modified between the two loads.

While the load is fully redundant along this trace it is only partially redundant when considering the complete procedure. Since register r23 is overwritten at 176 the load at 336 is not redundant along all paths and cannot be statically removed unless the register contents of r23 can

be saved in another register. However, there may not be another available register in which case the load cannot be removed without introducing new spills.

```
<mrglist>:
      . . .
 +144:      ldw 0(sr0,r5),r23
 +148:      combf,=,n r23,r4,<mrglist+208>
 +152:      bl 80f8 <cpylist>,rp
 +156:      ldo 0(r5),r25
 +160:      ldw 0(sr0,r5),r31
 +164:      comb,=,n r4,r31,<mrglist+124>
 +168:      ldo 0(r3),ret0
 +172:      addil 800,dp
 +176:      ldo 550(r1),r23
 +180:      ldwx,s r31(sr0,r23),r31
 +184:      ldo 1(r3),r3
 +188:      combf,=,n r4,r31,<mrglist+184>
 +192:      ldwx,s r31(sr0,r23),r31
 +196:      b <mrglist+124>
 +200:      ldo 0(r3),ret0
 +204:      nop
 +208:      addil 5c800,dp
 +212:      ldo 290(r1),r31
 +216:      ldwx,s r26(sr0,r31),r24
 +220:      ldwx,s r23(sr0,r31),r25
 +224:      comb,<=,n r25,r24,<mrglist+316>
        ....
 +316:      combf,=,n r25,r24,<mrglist+336>
 +320:      addil 800,dp
 +324:      ldo 550(r1),r23
 +328:      b <mrglist+300>
 +332:      ldwx,s r26(sr0,r23),r26
 +336:      ldw 0(sr0,r5),r23
 +340:      addil 35000,dp
 +344:      ldw 518(sr0,r1),r25
 +348:      addil 800,dp
```

8.1.2 Suboptimal register allocation

Another class of dynamic optimization opportunities results from sub-optimal register allocation decisions made by the static compiler when creating the binary. Optimization opportunities of this class don't arise until after code generation and therefore cannot be exploited by a standard global optimizer.

Since optimal register allocation is an intractable problem, static compilers are bound to make some sub-optimal decisions. Consider the problem of register allocation for individual array elements for example [Duesterwald et al. 1993]. While live ranges of individual array elements may easily be allocated the same register within a basic block, they are usually not considered during global register allocation due to the potential for aliasing and the difficulties in disambiguating array references. As a result, live ranges of individual array element that cross block boundaries are not likely to be held in the same register and redundant load instructions may be generated in every block within the live range.

Consider the source code below for function *findcaptured* from the benchmark go that contains several array references. Array element gralive[g] is the only array element that is

accessed more than once in each loop iteration. There are no aliasing problems since each iteration accesses only element gralive[g]. Thus, element gralive[g] could easily be held in a register to avoid reloading it multiple times.

However, consider the object code for function *fincaptured*. The underlined indexed load instructions load the exact same element (i.e., gralive[g]) . Although the loads are not within the same basic block, the second load is clearly redundant and could be replaced with a register copy from r31 to r18. This redundancy arises if the register allocator does not consider the live ranges for array elements. Thus, a load is generated for every array reference, even in the absence of aliasing.

Object code for function *findcaptured* from benchmark go:

```
          ...
+228:    ldo 0(r4),r26
+232:    comb,=,n ret0,r0,<findcaptured+592>
+236:    ldwx,s r4(sr0,r9),r31
+240:    combf,=,n r31,r6,<findcaptured+260>
+244:    sh2addl r4,r9,r25
+248:    ldw 0(sr0,r25),r26
+252:    depi -1,26,1,r26
+256:    stws r26,0(sr0,r25)
+260:    ldwx,s r4(sr0,r9),r18
+264:    combf,=,n r18,r15,<findcaptured+276>
+268:    bl <fixwasdeadgroup>,rp
+272:    ldo 0(r4),r26
```

Source code for function findcaptured from the go source file g25.c:

```
void findcaptured(void){
  int g,g2,comblist=EOL,ptr,ptr2,lvl,libs;
  for (g = 0; g < maxgr; ++g) {

        if (!grlv[g])continue
              lvl = playlevel;
        libs = taclibs;
        if (!cutstone(g)) {
              lvl = quicklevel;
              libs = quicklibs;
        }
        if (grlibs[g] > taclibs || cantbecaptured(g,taclibs)) {
              if (gralive[g] == DEAD)gralive[g] |= 32;
        }
        else if (iscaptured(g,80,lvl,libs,grcolor[g],NOGROUP)) {
              newdeadgroup(g,DEAD,gralive[g]&31);
              addlist(g,&comblist);
        }
        else if (gralive[g] == DEAD) {
              gralive[g] |= 32;
        }
        if (gralive[g] == (32 | DEAD))
              fixwasdeadgroup(g);
  }
  ...
```

Conceptually, some of the code deficiencies that are created by register allocation and code generation can also be eliminated by late code modification techniques [Wall 1986; Wall 1992]. Late code modification techniques include peephole optimization of basic block as well as more

elaborate global code reorganizations. Due to its additional cost late code modifications are usually not considered at standard levels of optimization.

8.1.3 Shared library invocation

While the use of shared libraries offers numerous advantages from a software engineering standpoint, their use is less desirable from a static optimization standpoint. References to shared libraries cannot be resolved until dynamic link-time. To resolve shared library references an entry in the Procedure Linkage Table (PLT) is created for every shared library procedure that is called by the program. Each call to a shared library procedure is redirected at link-time to a newly created import stub. The import stub accesses the PLT to retrieve the correct procedure entry address. Similarly, an export stub is created for every shared library procedure to handle the return from the shared library.

The import and export stubs for the PA-RISC architecture are as follows:

Import Stub:

```
ADDIL L'lt_ptr+ltoff,dp         ; get procedure entry point
LDW   R'lt_ptr+ltoff(1),r21
LDW   R'lt_ptr+ltoff+4(1),r19  ; get new Linkage Table value for r19
LDSID (r21),r1                  ; load new space ID
MTSP  r1,sr0                    ; set new space ID
BE    0(sr0,r21)                ; branch to procedure export stub
STW   rp,-24(sp)                ; store return point
```

Export Stub:

```
BL,N  X,rp                      ; jump to procedure X
NOP
LDW   -24(sp),rp                ; restore original return pointer rp
LDSID (rp),r1                   ; restore original space ID
MTSP  r1,sr0                    ; set original space ID
BE,N  0(sr0,rp)                 ; return
```

The import stub also handles the possibility of inter-space procedure calls through the loading and restoring of the appropriate space register. Thus, the original calling sequence consists of the following steps: call to import stub, look-up procedure entry, call to export stub, call to procedure and return from procedure.

Repeated lookups of the procedure entry point is redundant but cannot be avoided unless the original code is rewritten at runtime after the shared library is first invoked. Thus, invocation of shared libraries presents an ideal opportunity for dynamic optimization since redundant lookup is easily removed at fragment optimization time. There are two scenarios. Ideally both the shared library call and the return and are included in the same trace. In this case both the call and return can be inlined. If only the call is included in the trace, there are still substantial savings possible. The original calling sequence can be simplified to: call to export stub, call to procedure and return from procedure. The call to the import stub and the table look-up of the entry point can be saved.

8.2 Intermediate form

Before placing a trace into the Fragment Cache for execution it is first brought into an intermediate form called a *fragment*. A fragment is an executable and location independent region of code blocks with a unique entry block called the *fragment head*. Except for the fragment head, all internal blocks are join-free.

A fragment is described using a graph-based *Intermediate Representation (IR)*. The IR of a fragment serves as a temporary vehicle to transform the original instruction stream into optimized form for placement into the Fragment Cache. Unlike in a static compiler, both the Fragment Optimizer's input and output are at the machine instruction level. In order to enable fast translation to and from the IR, the abstraction level of the IR is therefore also kept low and close to the machine instruction level. Abstraction is introduced only when needed: to provide location-independence through symbolic labels and to facilitate code motion and code transformations through the use of virtual registers. Otherwise, the IR maintains a one-to-one mapping with the machine instruction set enabling both fast IR construction and fast code generation.

After final code generation, the IR for a fragment is discarded. Thus, the IR and other optimization-internal information are transient data structures that can fully be destroyed after final code generation. This transient nature allows for efficient memory management of all optimization-related internal data structures. The Fragment Optimizer maintains a fixed size memory pool, the *IR pool*, for all its internal data structures, including the fragment IR. Each optimization session starts with an empty pool and memory is allocated from the pool as the optimizations proceed. Importantly, memory can only be allocated but not freed individually during the session. After code generation, when the optimization session terminates, the IR pool is simply flushed and reset to prepare for the next session.

The only complication arises when the fixed size memory pool is full but further allocation requests are pending, i.e., an *out-of-memory condition* is encountered. To avoid complicating the IR pool management, the approach taken in Dynamo is to simply give up and reset. The current fragment that caused the out-of-memory condition is broken into two pieces and a new separate optimization session is started for each of the smaller pieces. Recall that all data generated during the optimization session is transient. Thus, the optimization session can be abruptly terminated and restarted at any time by simply resetting the memory pool. Termination and reset is implemented using the Unix setjmp/longjmp routines. The reset point is prior to construction of the fragment IR after trace selection. At any time during the optimization session control can safely return to the reset point via the longjmp at which time the IR pool is reset.

The Unix setjmp/longjmp routines are expensive since the runtime stack has to be unwound safely to restore the correct context for the reset point. Thus, the IR pool is sized so as to minimize the likelihood of an out-of-memory condition.

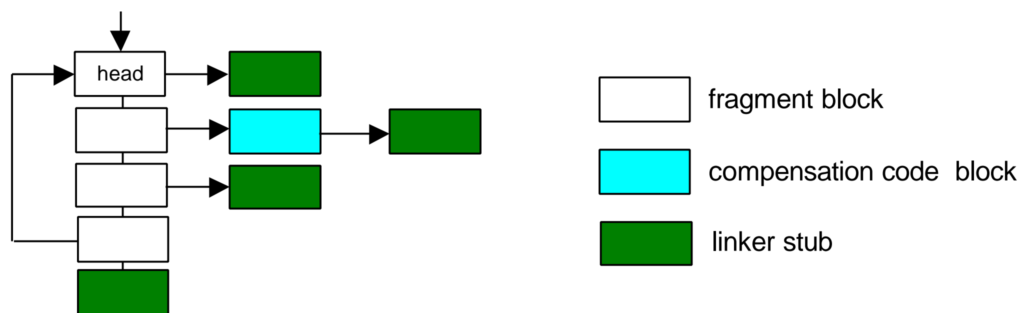


Figure 18: Anatomy of a fragment

8.3 Fragment formation

The process of transforming a trace into a fragment is called *fragment formation*. Fragment formation involves the adjustment of taken branches on the trace and the generation of *linker stubs* for trapping control when a branch attempts to exit the Fragment Cache. Linker stubs play the same role as *epilogues* in the Shade dynamic translator [Cmelik and Keppel 1993]. Since a

trace may contain several taken branches, the resulting code fragment has a single-entry and multiple-exits and a shape similar to a tree, except that the fragment head may be the target of a loop back edge. Figure 18 shows an example of a fragment.

8.3.1 Direct branch Adjustments

Unconditional direct taken branches can simply be eliminated. In the case of conditional direct taken branches, the sense of the branch must be flipped by inverting the branch condition. The previous taken target becomes the new fall-through successor and a new linker stub is inserted as the new taken target.

8.3.2 Indirect branch Adjustments

Indirect branches on the trace are transformed into conditional direct branches with the indirect branch target inlined [Witchel and Rosenblum 1996]. This transformation is based on the assumption that the indirect target that is part of the dynamic trace represents a likely target for future executions of the same indirect branch. The target is inlined as the new successor of the indirect branch and the branch is replaced by a test to verify that the inlined target is indeed the current indirect branch target. The test compares the dynamic value of the indirect branch target register with the inlined target address as illustrated in Figure 19. If the comparison succeeds control stays on the trace. Otherwise, control is directed to a hand-coded lookup code sequence that is permanently kept within the Fragment Cache (to avoid context switch). The lookup code implements a hash lookup of the fragment lookup table for the actual indirect target, and if it hits, jumps to the top of the appropriate target fragment. A failed lookup results in a trap to Dispatch. This is similar to the "speculative chaining" approach used in Embra [Witchel and Rosenblum 1996].

```
0x00: add gr1,gr3,gr12
0x04: bv  gr0,gr2
…

0x20: load  …
0x24: …
```

(i)

```
0x00: add     gr1,gr3,gr12
0x04: loadIm  0x20, gr5
0x08: cmbB,<> gr5,gr2,<exit>
0x0c: load    …
```

(ii)

Figure 19: (i) Indirect branch with actual target 0x20, and (ii) its conversion into a direct branch

8.3.3 Linker Stubs

Linker stubs are inserted for every fragment exiting branch and are responsible for initiating the control transfer to Dispatch via the context switch routines. Information about the original branch, including its original target address, is put in a *link record* data structure. A pointer to this data structure is embedded in the unique linker stub associated with the branch, to communicate it to Dispatch (see Figure 4, page 22). To identify the actual branch target upon exit from the Fragment Cache, Dispatch can simply extract the embedded pointer from the linker stub and retrieve the necessary information from the link record it points to. If the actual target is currently in the Fragment Cache the exiting branch can also directly be linked to the target fragment and future executions of the branch can possibly bypass the linker stub entirely. Fragment linking is discussed in detail in the upcoming Fragment Linking section.

8.4 Optimization

After the fragment IR has been constructed, the fragment is optimized during a few non-iterative passes. During each pass data flow analysis and optimization are interleaved. To minimize the storage needed for data flow analysis, data flow information is maintained in a fixed size buffer for the currently considered statement only.

This section describes the optimizations performed by the Fragment Optimizer for our Pa-RISC implementation of Dynamo. Depending on the architecture different or additional optimizations might be of interest. For example, when considering a statically scheduled VLIW architecture, instruction scheduling will be an important dynamic optimization. For superscalar architectures that implement instruction reordering in hardware, such as the PA-8000, instruction scheduling is of less importance.

The optimizations performed by the Fragment Optimizer include a number of standard static compiler optimizations. However, since these optimizations are applied to traces rather than complete procedures, the optimization task is simplified [Cohn and Lowney 1996; Cohn et al. 1997]. In addition to standard compiler optimization, the Fragment Optimizer also applies new transformations specifically developed for optimizing fragments.

8.4.1 Optimization levels

The optimizations performed in Dynamo are classified into two categories: *conservative* optimization and *aggressive* optimization. Conservative optimization guarantees the preservation of the machine context and deterministic bug-for-bug compatibility. Preserving the machine context is relevant for delivering precise exceptions. Deterministic bug-for-bug compatibility means that all deterministic program faults are faithfully reproduced. In order to preserved machine context instructions that change any part of the machine state, including carry-borrow bits, cannot be removed; even if these changes appear to be irrelevant side effects and the instructions are otherwise dead. Furthermore, in conservative optimization mode every memory access is assumed to be volatile. Thus, what appears to be a redundant load access cannot be removed. Finally, to ensure bug-for-bug compatibility the order of memory accesses must also be preserved. Even uninitialized memory accesses must be faithfully reproduced.

Self-modifying code does not generally cause problems for Dynamo. However, there are certain kinds of non-deterministic program faults involving self-modifying code that may not be preserved. Non-determinism is introduced in self-modifying code if the code does not execute an instruction cache flush immediately after the self-modification. Dynamo translates an instruction cache flush into a corresponding flush of its own Fragment Cache. Thus, Dynamo is guaranteed to pick up the modified code. However, if the original code does not flush the machine instruction cache after a self-modification the original program would behave in an unpredictable way and Dynamo will not be able reproduce that behavior.

Aggressive optimizations do not guarantee complete context preservation or bug-for-bug compatibility. Upon signal arrival only the PC (program counter) is guaranteed to be delivered correctly. Since the memory context may no longer be preserved, optimizations of this class may interfere with volatile memory accesses.

The specific kinds of optimizations that can be performed at the conservative level depend on the instruction set architecture. Our Dynamo prototype for the PA platforms performs the following safe optimizations[6]:

Constant propagation
Constant folding
Strength reduction
Copy propagation

[6] Constant folding and copy propagation are performed in order to trigger further strength reductions.

Redundant branch removal[7]

Deleting or moving instructions, even redundant ones, may make it difficult or impossible to guarantee faithful reconstruction of the original program's context at a signal or bug-for-bug compatibility. Thus, all remaining optimizations are classified as aggressive. The aggressive class of compiler optimizations performed in Dynamo are:

Redundant load removal

Note: A load is redundant if it is preceded by a load from (store to) the same memory location and the value still remains in a register. The redundant load is eliminated by replacing it with the appropriate register copy.

Dead code elimination

Loop unrolling

Loop invariant removal

The scope of aggressive optimizations can further be enlarged if certain assumptions about compilation conventions can be made. For example, assumption about certain calling or register usage conventions. If it can be assumed that the stack is only accessed via a dedicated stack pointer, stack references can always be disambiguated from other memory references. Enhanced memory disambiguation may in turn trigger the removal of additional redundant load instructions.

Recall that optimizations are classified as aggressive if they do not preserve the correct machine context or if they change the order of memory accesses. Assume the order of memory accesses is left intact. If it is possible to recreate the correct machine context rather than preserve it, an aggressive optimization can be moved to the conservative category. Recreating the correct machine context requires some form of de-optimization capability. If any discrepancy from the original machine context can de undone via de-optimization, the optimization may be promoted to the conservative level. The capability to de-optimize is also a critical issue for providing full support for symbolic debugging. De-optimization is currently not supported in Dynamo and is the subject of future research. The optimizations discussed in the following sections all fall under the aggressive class.

8.4.2 Speculative optimizations

Control-speculation leads to optimizations whose benefits are based on the assumption that the selected fragment remains hot and execution is unlikely to take an early exit from the fragment. Thus, to be effective control-speculative optimizations are highly dependent on an accurate prediction of the trace by the trace selection mechanism. Moreover, if the selected trace provides a poor prediction of future control flow, applying control-speculative optimization may actually result in misprediction penalties and performance degradation.

The Fragment Optimizer performs code sinking as a control speculative optimization. The objective of code sinking is to move instructions from the main execution path into fragment exits to reduce the number of instructions executed on the expected path. The instructions that can be sunk into fragment exits are the partially dead instructions, that is, instructions that define registers that are not subsequently read on the fragment. Such an instruction, while it appears dead on the fragment, cannot be removed since it is not known whether it is also dead at all subsequent fragment exit.

Code sinking is the symmetric counterpart to code hoisting for partial redundancy elimination [Morel and Renvoise 1979; Knoop et al. 1994; Bodik et al. 1998]. It can be performed in an "optimal" manner, that is, while guaranteeing that no additional instructions are introduced on any path. Code sinking is optimal as long as no additional compensation block

[7] Redundant branches may result from trace selection. For example, every unconditional direct branch on the trace is trivially redundant in the fragment. Redundant branches may also be detected by constant propagation.

must be created at fragment exits as a placeholder for the sunk instructions. Creating new compensation blocks may lead to misprediction penalties. Heuristics may be used to reduce the likelihood for misprediction penalties. For example, code sinking that requires compensation block creations may only be performed for high-latency operations and disabled otherwise.

In addition to control-speculation, the Fragment Optimizer may also speculate on data locality. Prefetching is an optimization that is based on data-speculation, i.e., on the assumption that the respective data are currently not in the data cache.

```
0x00: load  0(gr5),gr7
0x04: store gr12,0(gr13)
0x08: load  0(gr5), gr8
```

(i) original

```
0x00: load        0(gr5),gr7
0x04: store       gr12,0(gr13)
0x08: copy        gr7,gr8
0x0c: cmpclr,<>   gr13,gr5,gr0
0x10: load        0(gr5),gr8
```

(ii) guarded load

Figure 20: Example of guarded load optimization

8.4.3 Multi-versioning optimization

During optimization one often has to make worst case assumptions that may prevent optimizations that would be safe in certain special cases. Multi-versioning is a technique to enable special-case optimization by creating two versions of the code: an unoptimized version for the general case and an optimized version for the special case. Procedure cloning [Cooper et al. 1993] is an example of multi-versioning. The decision as to which version to execute can be made either statically, as in procedure cloning, or dynamically by inserting an appropriate run-time test. The Fragment Optimizer performs *instruction guarding*, which is a form of single-statement multi-versioning.

Instruction guarding is performed in order to eliminate load instructions that may or may not be redundant depending on an intervening store that cannot be disambiguated by the optimizer. Consider the code in Figure 20 (i). Since nothing is known about the register contents upon fragment entry, it must be assumed that the store *kills*, i.e., writes to same memory location as the preceding load. Thus, the second load cannot be determined as redundant. However, if a comparison of the contents of registers gr5 and gr13 indicates that the store actually does not overwrite the loaded location, then the second load can be removed as redundant.

Since the necessary memory disambiguation cannot be completed at optimization time it will be postponed to execution time by inserting instruction guards. The idea is to optimistically assume that the second load is redundant by replacing it with the appropriate register copy instruction. To ensure correctness, the load instruction still executes if the optimistic assumption turns out to be false. This is achieved by guarding the load with at runtime test as shown in Figure 20 (ii). Instruction guarding can be implemented efficiently if the target architecture provides predication. The PA-RISC architecture supports a limited form of predication in the form of instructions that can nullify the following instruction. An example is the `cmpclr` (compare and clear) instruction that compares the contents of two registers and conditionally nullifies the following instruction based on the results of the comparison.

8.4.4 Fragment-link-time optimization

The optimizations described in the previous sections are strictly local to each fragment. Worst case assumptions are made about a fragment's execution context at its entry and exit points. Information about the dynamic execution context of a fragment is available when fragments are linked by the Fragment Linker (details of the linking process are provided in the Fragment Linker section later in this report).

Each time a link is established between two fragments in the Fragment Cache, information can be propagated across the new connection. One approach to exploit this additional information would be to re-generate and re-optimize the combined connected fragment. A less expensive approach, and the one followed in Dynamo, is to apply peephole optimizations around the new connection. The goal of these optimizations is the removal of instructions that are dead across fragments and could not have been eliminated prior to establishing the connection.

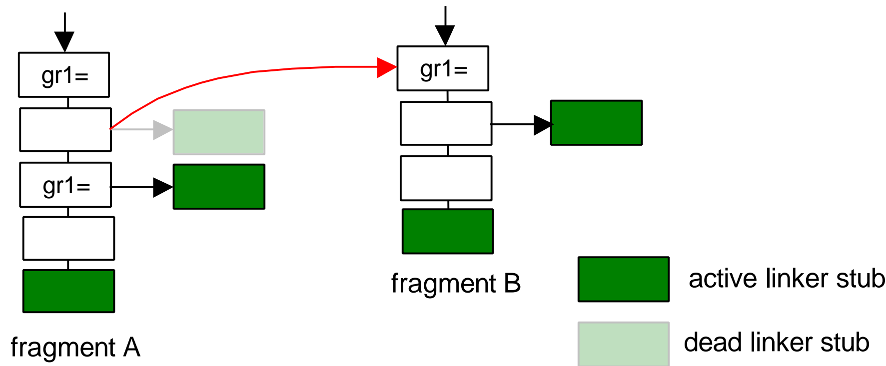


Figure 21: Example of link-time optimization

We distinguish between assignments that are *dead* and that are *not live* on the fragment. A register assignment is dead on the fragment if prior to the next exit the register is re-assigned without being read first. The assignment is not-live if the register is not read on the fragment. Every dead assignment is not-live but not vice versa.

An assignment is *dead across fragments* if it is not live in one fragment and it becomes dead after linking. Figure 21 illustrates an example. Fragment A contains an assignment to register gr1 in its first block that is not live. The assignment cannot be removed prior to linking since it is not known whether the value of register gr1 will be used after exiting from fragment A. However, after linking to fragment B, the assignment can safely be removed as dead since register gr1 is assigned in fragment B without being read first.

One way to detect these additional dead instructions after linking would be to completely re-analyze the combined code. Dynamo employs a more efficient technique to perform link-time optimizations that avoids any form of re-analysis. Instead of discarding the unused information at fragment generation time and re-computing it later at link-time, the relevant information is stored in the form of a fixed-sized *epilog* at each unit's exit point and a fixed-size *prolog* at each entry point.

8.4.4.1 Link-time prologs and epilogs

The epilog structure associated with each exit e is a size k array of pointers to instructions that represent the non-live assignments that may become dead after linking. An assignment is a candidate for dead code removal at exit e if and only if the assignment (i) is not live and (ii) if the assigned register is dead immediately after the exit e.

Up to (k-1) such candidates are selected such that each candidate writes to exactly one register and at most one candidate writes to each register. The set of candidates is sorted by increasing value of the register they write to. A list of pointers to the actual code positions of the candidates, sorted by their position in the fragment, is stored in the epilog. Figure 22 shows an example of an epilog for k=5, i.e., there is room for four instruction pointers in the epilog. In this example there are only two candidates: non-live assignments of registers gr1 and gr4. The remaining unused pointers are set to NULL.

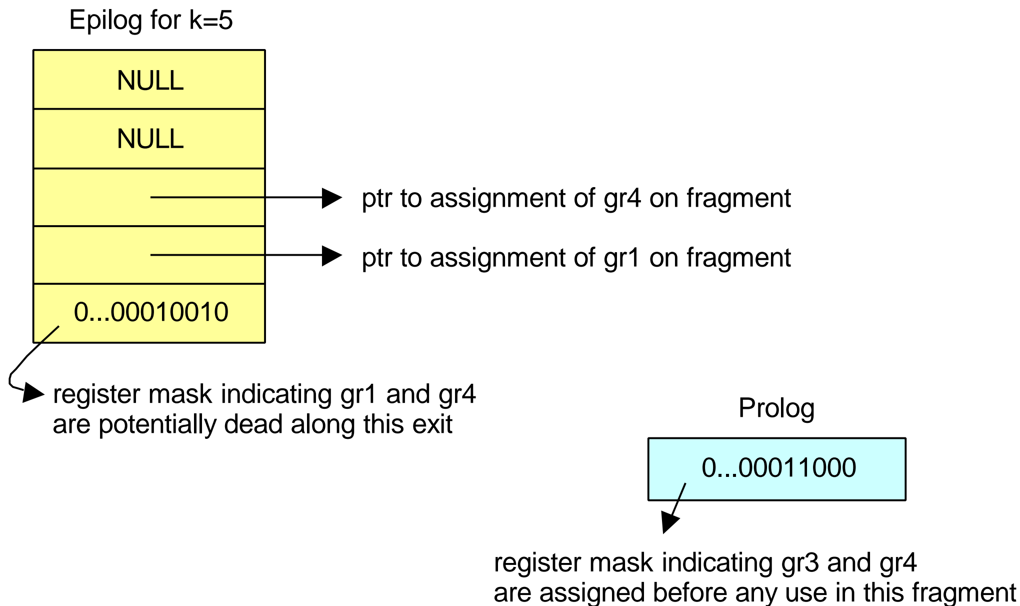


Figure 22: Fragment epilog and prolog used for link-time optimization

To quickly access the correct pointer at runtime the k-th word in the epilog contains a register mask. The bit at position *i* in the mask is set if and only if there exists a candidate that writes to register *i*. For example in Figure 22, the register mask has the bits one and four set. Given a bit position *i* that is set in the register mask, the correct pointer to the corresponding non-live assignment still has to be located. Since the pointers have been sorted by their position in the fragment, the register mask also serves as a means to access the correct pointer. The correct pointer is found simply by counting the number of bits in the mask that are set prior to the bit position of interest. If there are j such bits, the correct pointer is the (*j*+1)-th pointer in epilog. In our example, bit number 1 is the only bit set prior to bit position 4. Thus, the correct pointer to the assignment to register gr4 is the second pointer as shown in Figure 22.

The prolog associated with each fragment entry contains a single word to store a register mask. The register mask indicates the registers that are defined in the fragment prior to being read. Specifically, bit *i* in the mask is set if and only if register *i* is defined before being read. In the example of Figure 22, the prolog indicates that registers gr3 and gr4 are defined prior to being read.

Based on the information stored in the epilog and prolog the actual dead code removal when linking a fragment exit and fragment entry point consists of two simple steps. First, the logical conjunction of the exit and entry register masks of the linked fragments is computed. The bits that are set in the result vector indicate the assignments that are dead across the exit and entry point. The next step consists of locating and deleting the dead instructions by accessing the correct pointer in the epilog as described above.

Following the above algorithm, up to (*k-1*) dead instructions can be removed each time an exit branch is linked to a fragment entry.

8.5 Undesirable side-effects of optimization

Besides the intended performance improving effect, optimizations may have other unintended side effects. As pointed out earlier speculative optimizations may result in misprediction penalties. Other architecture-dependent optimization side effects are subtler and

may either improve or degrade performance. Examples include the introduction or prevention of machine stalls, such as cache misses, TLB misses, or Branch Prediction Table collisions. These side effects may be caused by the code growth and branch repositioning that results naturally from optimization. Architecture-dependent side effects cannot easily be controlled by the Fragment Optimizer unless the optimization framework includes a complete model of the architecture.

Another class of side effects results from the use of virtual registers. Using virtual registers in the fragment without awareness of the register pressure may result in additional spill code. Whether additional virtual register usage results in a spill code is not easily determined and would require an integrated framework of optimization and register allocation.

8.6 Register allocation

At the time the fragment is formed, the code has a complete register assignment. During the fragment formation the original register assignment is retained. However, the intermediate representation also supports the use of virtual registers which may be introduced during fragment formation and fragment optimization

During fragment formation an extra register is needed when adjusting indirect branches. Recall that for indirect branch adjustment, the taken target on the trace is compared against the current target. To implement this comparison on the PA-RISC architecture it is necessary to load the address of the previous target into a new virtual register.

During fragment optimization extra registers maybe needed to facilitate code motion. Finally, to enable context switches between Dynamo and the Fragment Cache Dynamo may have to free a dedicated register to serve as the context pointer (depending on the architecture, there may not be a need to free a register for this purpose, as discussed in Section 5.1). Thus, during fragment formation every reference of the dedicated context pointer is replaced with a reference to a virtual register.

If a fragment contains references to virtual registers, a register re-allocation pass is necessary after fragment optimization. The task of re-allocation differs form ordinary register allocation in a static compiler in that all previous non-virtual register assignments are retained. During a single pass over the fragment live ranges for virtual registers and a map of free available registers are constructed. Note that a register is free from the point of its last use to the point of its re-assignment on the fragment. If a free register is available, the free register replaces the virtual one. Otherwise, a register is spilled and restored around the live range.

An alternative approach would be completely redo the previous allocation. Thus every register is treated as a virtual register and previous static register allocation algorithms may be used. Since the fragment contains no join points, registers can be allocated optimally using fast interval-graph based algorithms. However, any re-allocation must restore the original register mapping at fragment exit points. Shifting the registers to obtain the original mapping when exiting the fragment would be too costly compared to the expected benefits of completely redoing the register assignment on the fragment.

8.7 Experimental evaluation

We experimentally evaluated our Dynamo prototype and examined its performance for a set of benchmark programs[8]. To evaluate the benefits of dynamic optimizations it is important to note that the benchmarks programs were already statically optimized during compilation with optimization level –O2. The optimization performance was evaluated at the two fragment optimization levels as discussed earlier in Section 8.4.1:

[8] See Section 12.1 on page 80 for a detailed description of our experimentation environment.

A conservative category at which only the following optimizations are performed: constant propagation, constant folding and strength reduction, copy propagation and redundant branch removal.
An aggressive category that includes all optimizations.

	strength reduction	redundant load	redundant branch	dead code	code sinking	guarded load	loop invariant motion
go	20620	2430	4315	2034	1172	0	21
m88ksim	2641	244	507	545	112	11	4
compress	194	9	30	28	1	0	2
li	3974	969	590	1162	122	0	3
ijpeg	5710	377	939	733	74	0	2
perl	3748	919	372	304	123	0	4
vortex	50310	3186	5548	12864	557	2	1
boise	7151	701	1128	1108	176	0	0
deltablue	2513	101	210	259	13	0	1

Table 1: Optimization opportunities

We evaluated the performance of runtime optimization in the Dynamo prototype running on a HP PA-8000. The PA-8000 is a four-issue out-of-order superscalar 180 MHz processor. Aggressive instruction reordering is supported by a 56-entry instruction reorder buffer. The actual runtime improvements that we measured are specific to the PA micro-architecture and may vary on different machines.

Table 1 shows the number of optimization opportunities found in each program. The shaded columns denote the optimizations that are done at the conservative level; all optimizations are performed at the aggressive level. The numbers indicate that, despite the fact that the programs were statically optimized, optimization opportunities at runtime are plentiful.

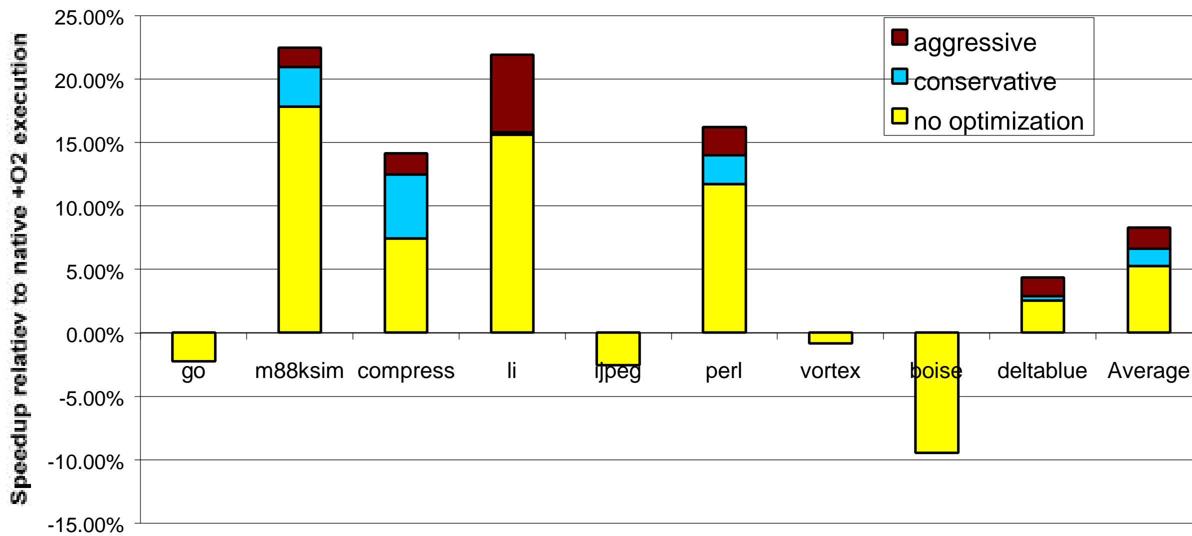


Figure 23: Overall Dynamo speedup with optimization contribution

Table 1 does not show to what extent the exploited optimization opportunities cause actual performance improvements. Figure 23 shows the performance improvements that are solely due to optimization. The figure is based on running Dynamo on –O2 compiled binaries and the speedups are shown over the native performance of these binaries. The speedup is broken up into the speedup achieved without optimization and the speedup contribution of the conservative and aggressive optimization levels.

Figure 23 shows that for about half the programs a significant portion of the optimization benefits is due to conservative class of optimizations. The most effective in this category is the redundant branch removal. For example, besides removing a procedure call branch, procedure inlining removes the indirect return branch which is likely to be mispredicted on the PA-8000. High benefits from branch removal can be expected in deeply pipelined architectures since the relative penalties of branch misprediction are high. In the boise benchmark, the conservative category of optimizations alone do not appear to have sufficient benefits to outweigh the optimization overhead.

Figure 24 shows the optimization and total Dynamo overhead as a percentage of the complete execution time. The optimization overhead presents a negligible fraction of the overall Dynamo overhead, on average less than 0.2% of the total execution time. With overheads so low, the Fragment Optimizer can be extended to include much more aggressive optimizations while still maintaining efficiency.

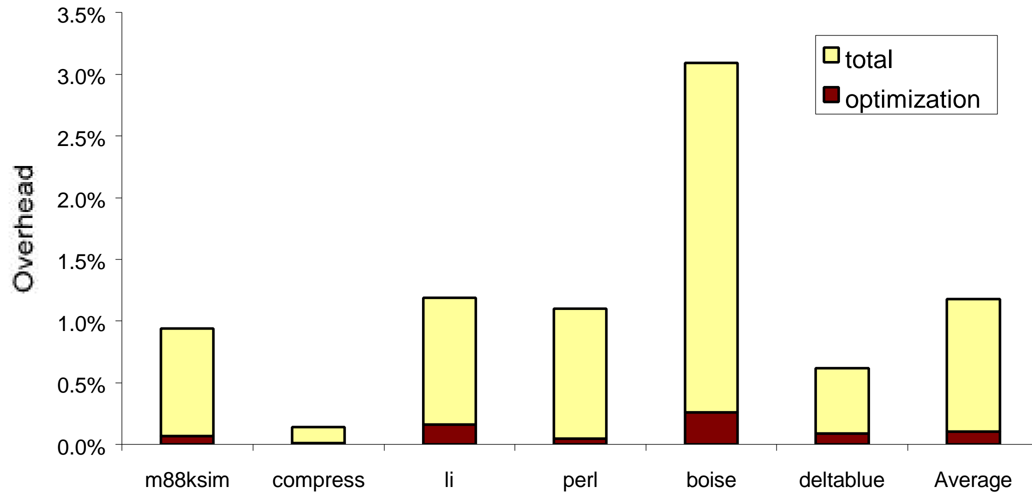


Figure 24: Optimization overhead

9 Fragment Linker

Fragment linking is the mechanism by which fragments are made to transfer control to one another without exiting the Fragment Cache. An important performance benefit of linking is the elimination of unnecessary context switches between the Fragment Cache and Dynamo. The Fragment Linker is also responsible for emitting the fragment code into the Fragment Cache, a step that precedes the actual linking itself.

9.1 Emitting the fragment code

The Fragment Linker is invoked after the Fragment Optimizer completes. As stated in the previous section, the fragment is now in the form of a low-level IR (intermediate representation). The IR instructions map directly onto instructions in the underlying processor's instruction set, and have already been register allocated by the Fragment Optimizer. Thus, code generation is a very straightforward step.

The Fragment Linker first invokes the Fragment Cache Manager to allocate sufficient room in the Fragment Cache for emitting the fragment code. It supplies the program PC corresponding to the fragment entry point, which the Fragment Cache Manager uses to tag the newly allocated space in the cache. The Fragment Cache Manager returns a pointer to the top of the allocated space, which is used by the Fragment Linker to materialize the fragment's instructions into the space. Since the Fragment Cache is a software instruction cache in memory, any write into this area potentially creates an inconsistency with the underlying processor's instruction cache, since portions of the Fragment Cache may be resident in the processor's instruction cache. Since modification of text segments is rare, most machines do not automatically keep the instruction cache in sync with memory. Instead, they usually provide a user-mode flush instruction that invalidates a part or all of the machine instruction cache, causing it to be re-filled, and thereby re-sync its contents with the memory image. The PA-RISC provides a flush instruction at single line granularity. After the fragment code is emitted into the Fragment Cache, the machine instruction cache lines to which the fragment's address range map are explicitly flushed by the Fragment Linker.

Once the fragment code has been emitted into the Fragment Cache, the actual branch linking is performed as described below, following which the fragment IR is deleted [9].

9.2 Pros and cons of branch linking

Recall from Section 8.3 (page 45) that every branch that exits a fragment is initially set up to jump to a unique linker stub associated with the branch. The purpose of the linker stub is to trap control back to Dynamo. Figure 4 (page 22) illustrates how this is accomplished. Implementing such a trap involves saving the program's context (actually the context of the program's instructions executing within the Fragment Cache), prior to jumping to the entry point of Dispatch. Since the context switch adds to the overall Dynamo overhead, it has to be kept low. One way to accomplish this without relying on any hardware support is to short circuit exit branches whose targets have already been materialized in the Fragment Cache.

This can be done by replacing the exit branch instruction on the fragment with another one, whose taken target is another fragment's entry point instead of its linker stub. Subsequent executions of the branch, when taken, will go directly to the target fragment, bypassing its linker stub entirely. The branch is now termed as *linked*. Linking is also referred to as "translation chaining" in dynamic translator implementations such as Shade [Cmelik and Keppel 1993].

[9] All the fragment IR objects are allocated from a single memory pool. Thus deletion simply involves a quick reset of the pool.

While the advantage of linking is clear, it also has some disadvantages that need to be kept in balance when designing the linker. For example, linking makes the removal of an arbitrary fragment from the Fragment Cache more expensive since all incoming linked branches to the fragment have to be unlinked. It also makes it expensive to relocate linked fragments in the Fragment Cache, which might be desirable for instance to defragment the Fragment Cache storage periodically. Another problem with linking is that it makes it harder to bound the latency of asynchronous signal handling. Asynchronous signals arise from events like keyboard interrupts and timer expiry, and their handling is postponed by Dynamo until control exits from the Fragment Cache (this is elaborated further in the Signal Handler section later in this report). When fragment branches are linked, control could end up spinning in a loop within the Fragment Cache for an arbitrarily long time before the next Fragment Cache exit occurs following the arrival of such a signal. Preventing this is especially important in systems that require at least a soft real time response to asynchronous signals.

Our approach in designing the Fragment Linker is to make unlinking a very fast operation, so that at any time all linked branches in a given fragment can be quickly unlinked. Also, rather than use an on-demand approach, where linking is performed only on branches that exit the Fragment Cache, we do it preemptively, at the time a fragment is emitted into the Cache. The mechanics of linking and unlinking are explained in detail below. Linking is essentially what sets a software code cache apart from its hardware counterpart: when a taken branch is executed, the hardware cache has to do a fresh tag lookup on the target address, whereas the software cache does not.

9.3 Link records and the fragment lookup table

When a new fragment is added to the Fragment Cache, two sets of branches have to be linked: exit branches in the new fragment, and exit branches from other fragments that target the new fragment's entry address. The data structures used for linking are optimized to allow quick access not only to the new fragment's exit branches but also to incoming branches from other fragments. Every branch that can exit a fragment has a link record structure associated with it, that is created by the Fragment Linker prior to emitting the fragment code into the Fragment Cache. All link records are maintained as part of the central *fragment lookup table*, the hash table that used by Dispatch to check for Fragment Cache hits (see Section 5, page 20).

Entries in the fragment lookup table are indexed using the program PC corresponding to a fragment's entry point as a hash key. Each entry contains information about the fragment (such as its size, the address of its entry point in the Fragment Cache, etc.). It also contains two lists: an exit branch list and an incoming branch list. Each entry in these lists is a pointer to a link record, which contain information about a specific branch in the Fragment Cache. There is a one-to-one mapping between fragment exit branches and link records. The link record contains the branch instruction's address in the Fragment Cache, its original target address (before it was modified to jump to its linker stub), the address of its linker stub, the fragment the branch belongs to, and various flags indicating its linkage status.

When the Fragment Linker emits code for a new fragment into the Fragment Cache, it updates the fragment lookup table as follows:

1. A new link record is created and initialized for each exit branch in the new fragment.
2. At the entry F in the fragment lookup table corresponding to the new fragment, an exit branch list is set up that chains the link records created in step 1. The Fragment Cache address of the fragment's entry point is also initialized in entry F.
3. For each exit branch B in the new fragment, its original target address (found in its link record) is used to hash into a corresponding entry T in the fragment lookup table. Entry T may or may not contain a fragment in the Fragment Cache. In either case, a pointer to B's link record is added to the incoming branch list associated with entry T (if such a list is not

already there, a new one is created). This step registers branch B as one of the incoming branches that have to be linked if in future a fragment is added at entry T in the fragment lookup table. If a fragment already exists for T, branch B is linked immediately.

4. For each link record in the incoming branch list associated with entry F (i.e., the branches that are registered as targeting the entry address of the new fragment), the corresponding branch is linked to the top of the new fragment.

9.4 Direct branch linking

The target address of a direct branch is known when the fragment is formed, so linking a direct branch is straightforward. Linking works by replacing the original direct branch instruction (which targets a unique linker stub in the Fragment Cache) with another that targets a fragment entry point within the Fragment Cache. The original direct branch may be a conditional branch (such as a compare-and-branch instruction), or it may be an unconditional branch. An unconditional branch can arise in one of two situations on the fragment. The first case is when it is in the shadow (or delay slot) of an instruction that can potentially nullify its shadow. In this case, the pair of instructions with the unconditional branch in the shadow behaves as a conditional branch. The second case is at the end of the fragment body. The Fragment Optimizer will always generate an unconditional branch at the end of a fragment body if it does not already end in a branch instruction. This is required to trap control when it falls off the end of the fragment, in the same way that control must be trapped when executing a fragment exit branch. The linking process itself does not distinguish between any of these cases. However, the linking of a conditional branch is potentially complicated by its restricted reachability, whereas this problem does not arise in the case of an unconditional branch. On the PA-RISC, conditional branches have a 11-bit displacement, and can thus have only a 2 KB extent, which may not be enough to reach the intended target fragment within the Fragment Cache.

If the target fragment is more than 2 KB away from the branch, a landing pad is required to perform the long jump. Fortunately, the unique linker stub corresponding to the branch serves as a perfect place to keep the landing pad, since the moment a branch is linked, its linker stub becomes dead code. This is termed a *stub-based link*, because the original branch is not modified; instead, its linker stub (the current branch target) is modified to serve as a landing pad to get it to the target fragment. If the target fragment is within 32 KB of the linker stub, an unconditional direct branch can be used to overwrite the first linker stub instruction. Unconditional direct branches have a 15-bit displacement on the PA-RISC. If the target fragment is beyond 32 KB, a two instruction sequence consisting of a load of the target fragment address into the scratch register followed by a branch to the contents of this register. On the PA-RISC, we use a load immediate followed by a branch-and-link that uses gr0 (the bit bucket) as its link register [10].

Once the linking is done, the linkage status flags in the link record corresponding to the linked branch are updated. These flags indicate whether or not the branch is currently linked, as well as the type of linking that was done (direct versus stub-based link), and in the case of a stub-based link, whether the one or two-instruction sequence was used. These flags are used primarily to speed up *unlinking*, which is the reverse of linking, and is a procedure that we will discuss shortly. Since linking a branch involves writing into the Fragment Cache area, the modified location has to be synced up with the underlying processor's instruction cache. Thus, the instruction cache line to which the modified branch instruction address maps, is invalidated by doing a line flush.

---

[10] As an optimization, if the original conditional branch shadow is a `nop`, the branch can be converted into a compare followed by an unconditional branch (with the unconditional branch overwriting the `nop`), if the target fragment is > 2 KB but < 32 KB away from the original conditional branch instruction.

9.5 Indirect branch linking

The target of an indirect branch can only be determined when the branch is executed at runtime. As explained in Section 8.3.2 (page 46), the Fragment Optimizer converts an indirect branch to a conditional compare-and-branch that tests if the current target is equal to the inlined target (the program PC corresponding to the very next instruction in the fragment). The inlined target is the indirect branch target that was encountered at the instant that the trace corresponding to this fragment was formed. If the targets are not equal, the conditional branch jumps to its linker stub and exits the Fragment Cache. Otherwise it falls through to the inlined target on the same fragment.

The Fragment Linker treats this conditional branch specially. It is *always* linked to a special

```
spill %Rl, Rl_offset(%Rcontext); spill register Rl to the context save area
load &cntxtsave, %Rscratch   ; load the address of the context save routine
bl,n 0(%Rscratch), %Rl       ; branch-and-link to the context save routine
<shadow slot>                ; branch shadow is nullified
```

Figure 25: The linker stub template

fragment that is permanently resident in the Fragment Cache [11]. This special fragment is actually a hand-coded lookup code that accesses the fragment lookup table to see if there is a fragment corresponding to the actual indirect branch target [12]. If the lookup succeeds, control jumps to the top of the appropriate target fragment. Otherwise, it jumps to the context switch routine that eventually transfers control back to Dispatch. A link record is created for the conditional branch that tests the inlined target, just as with any other direct branch.

Thus, the Fragment Linker only sees direct exit branches in the fragment, so the actual mechanics of linking and unlinking is the same for all exit branches on a fragment.

9.6 Linker stubs revisited

Recall that the purpose of a linker stub is to trap control that is about to exit the cache, and give control to Dispatch, the central supervisor routine of the Dynamo system. Every linker stub consists of the same four-instruction sequence (henceforth referred to as the *linker stub template*), shown in Figure 25. "Rcontext" is a dedicated register that always contains a pointer to the start of the context save data structure within a fragment, and "Rscratch" is a scratch register whose original program value is always in the context save memory. "Rl" is a fixed link register that is used by all the linker stubs.

When control enters Dispatch, it needs to know the next program PC that has to be executed. In addition, some of the trace selection schemes also need to know the fragment from which control just exited the Fragment Cache. Thus, "come-from" information has to be provided to Dispatch, in addition to "go-to" information. There is already a structure that contains all of this information: the link record. Since there is a unique link record for each exit branch, and also a unique linker stub for each exit branch, a natural solution is to tie these structures together. This can be done by setting up the linker stub to communicate a pointer to its associated link record to Dispatch, so that Dispatch can then access the link record to determine all necessary come-from and go-to information.

---

[11] The fragment is marked so that it is never flushed out of the Fragment Cache.

[12] The actual indirect branch target is always present in a specific register by convention. This convention is enforced by the Fragment Optimizer when it does the indirect branch conversion.

This is accomplished by ending the linker stub with a branch-and-link instruction that jumps to the context save routine (as shown in Figure 25), which then jumps to the entry point of Dispatch. The advantage of using a branch-and-link instruction instead of a regular branch is that the link register implicitly records an address in the neighborhood of the linker stub. On the PA-RISC, the link point is 8 bytes (2 instructions) past the branch-and-link instruction itself. All branches, including the branch-and-link instruction on the PA-RISC have a shadow slot (or delay slot) that can be optionally nullified if the branch takes. This shadow slot is used to embed a pointer to the link record associated with the exit branch instruction corresponding to the linker stub, and the branch-and-link instruction is set up to always nullify this shadow. Note that this strategy of embedding a link record pointer will also work on machines that do not have a branch shadow slot, because control never "returns" to a linker stub. Thus, the "instruction" following the branch-and-link can never be executed.

The context save routine is written in assembler, and knows about the dedicated link register used in the linker stub template. Since the program's value of the link register is saved into the context as part of the linker stub code sequence (as illustrated in Figure 25), the context save routine does not save it to the context data structure. Instead, it saves the contents of this link register into a special part of the context data structure that is designated for the link record address, and Dispatch reads this value to pick up the link record pointer corresponding to the most recent branch that exited the Fragment Cache. In this way, when control traps to Dispatch upon a Fragment Cache exit, Dispatch knows exactly which branch instruction caused the exit, and what the original program PC of its target was.

9.7 Branch unlinking

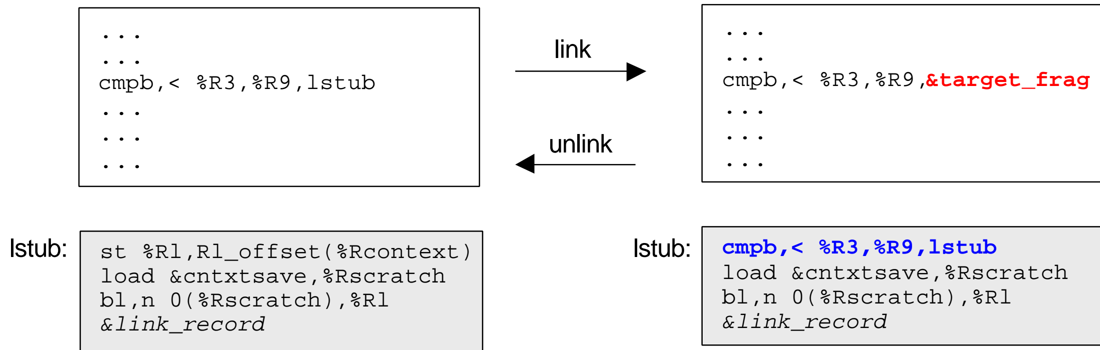
Branch unlinking is the reverse of branch linking. It involves re-materialization of the original exit branch instruction and its associated linker stub, so that if the branch takes, control traps into Dispatch instead of going directly to a target fragment. Unlinking may be required in two situations: for fragment deletion from the Fragment Cache, and for bounding the latency of signal handling in systems that rely on a soft real-time response to signals.

Fragments may get removed from the Fragment Cache in order to free up more room, or for purposes of forcing a re-formation of the fragment. In either event, when a fragment is removed, incoming branches from other fragments have to be unlinked.

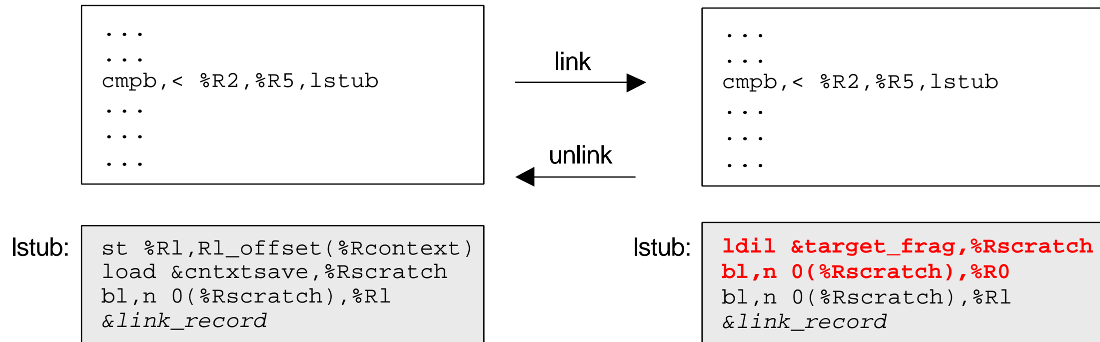
When control is within the Fragment Cache and a signal arrives, Dynamo has to construct a signal context that corresponds to the original program, not its version executing within the Fragment Cache. Because of optimization, this is a hard problem in general. However, certain kinds of signals are asynchronous in nature, and Dynamo can postpone dealing with them until control reaches a point where the original program signal context can be constructed. One such point is when control is in Dynamo. Examples of asynchronous signals include events like keyboard interrupts or timer expiry conditions, that do not have to be delivered at the precise PC at which they occurred. When an asynchronous signal arrives, the Signal Handler component of Dynamo adds it to a pending signal queue and continues Fragment Cache execution. When control eventually exits the Fragment Cache and goes into Dynamo, Dispatch first checks the pending signal queue and processes all the pending asynchronous signals. When fragments are linked, however, the time interval between the arrival of such a signal and when Dispatch gets to process it can be arbitrarily long because of loops within the Fragment Cache. In order to bound the latency of asynchronous signal handling, all exit branches in the current (executing) fragment are unlinked by the Signal Handler before execution is resumed in the Fragment Cache. If soft real time response is a requirement, linker stubs and link records can also be generated for backward branches that go to the top of their own fragments (i.e., self-loops). This allows even self-loops to be unlinked upon arrival of an asynchronous signal. Unlinking these branches on the current fragment forces a Fragment Cache exit at the next taken branch. Since fragments are

never longer than 1024 instructions in Dynamo, this strategy bounds the latency between signal arrival and signal delivery.

Unlinking is designed to be very fast in Dynamo. The trick to accomplishing this is to cache a copy of the original branch instruction (prior to linking it), so that it can be re-materialized simply by overwriting the linked version of the branch with its original bits. Once again, the unique linker stub associated with the branch instruction serves as a perfect place to cache the original branch instruction bits. This works because the moment a branch is linked, its linker stub (or the portion of it that is not used for stub-based linking) is dead code, and can be treated like a buffer in which to cache this data.



(a) direct linkage



(b) stub-based linkage (shown for the case when the target fragment is > 32 Kb away)

Figure 26: Illustration of linking and unlinking

Two cases arise. First, if the branch linkage is direct, the original branch bits can be cached in the first linker stub instruction slot. Upon unlinking, these bits can be copied back to the original location, and the first linker stub instruction can be recreated from the linker stub template. Second, if the branch linkage is stub-based, only the first two linker stub instructions have to be recreated, since the original branch is not modified during linking. This can be done from the linker stub template. Note that the fourth linker stub "instruction" contains the link record pointer, and must always be preserved, regardless of the type of unlinking involved. Figure 26 illustrates the two cases of linking and unlinking.

9.8 Performance benefit from linking

Figure 27 illustrates the performance benefits that result from linking for the six programs that do not bail-out under Dynamo. The chart shows for each program the performance slowdown factor that results if linking is not enabled. The average slowdown factor is 39. Clearly, linking is mandatory to avoid excessive context switching and the resulting severe performance degradation.

Figure 27: Performance slowdown when linking is disabled

10 Fragment Cache Manager

The Fragment Cache Manager controls Dynamo's software code cache. The cache is designed around a malloc-like memory allocator and stores fragments in the order in which space is requested by the Fragment Linker. The design enables inter-fragment locality and a simple overall implementation. The following sections described three aspects of the Fragment Cache. Section 10.1 provides details on the design. Section 10.2 describes a novel Fragment Cache flushing technique for managing memory usage based on the trace selection rate. This technique is one of the key innovations in the Fragment Cache and yields substantial reductions in Dynamo's memory usage. Lastly, Section 10.3 describes an alternative Fragment Cache organization for 2-level hierarchical trace selection.

10.1 Fragment Cache design

Conceptually, the Fragment Cache need not have any knowledge of activities that occur outside of its domain of space management. From the Fragment Cache's perspective, the other Dynamo components simply request space (memory) from the Fragment Cache or ask the Fragment Cache to free previously allocated space. The view of the Fragment Cache as a simple space manager can yield a very simple design and implementation. Other portions of Dynamo – the Trace Selector, the Fragment Linker, etc. – can use their knowledge of specific aspects of the system to intelligently instruct the Fragment Cache to perform specified actions, such as displace a specific fragment.

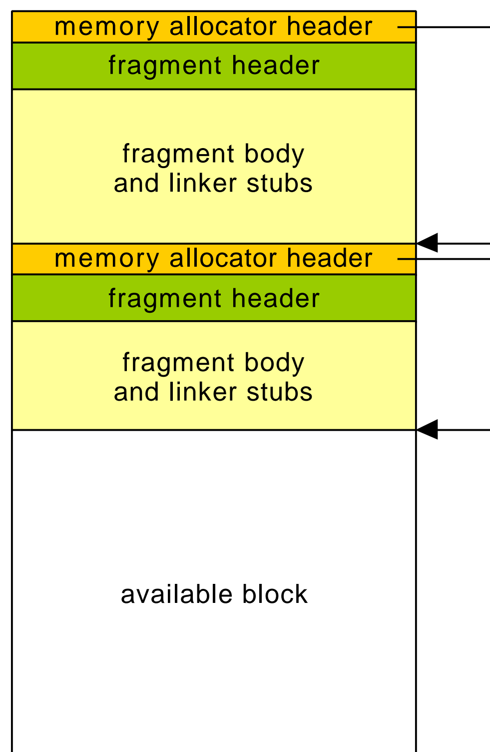


Figure 28: Fragment Cache memory map with 2 fragments

In fact, in the current software-only Dynamo prototype, the Fragment Cache does not have the means with which to make complex decisions. To illustrate the point, contrast the Fragment Cache with a hardware I-cache. An I-cache can make informed decisions about, for example, line

replacement because it can access canonical information about line usage patterns (subject to hardware constraints). However, the Fragment Cache has no knowledge of what fragments are accessed at what times. When execution transfers into the Fragment Cache, Dynamo no longer has control – it cannot "see" inside the Fragment Cache – and therefore can gather limited information about in-cache execution.

The Fragment Cache is designed as a simple memory allocator. It uses the poolmalloc memory allocator to maintain a fixed block of storage from which it attempts to satisfy allocation requests. (poolmalloc is described in detail in the System Development Tools and Utilities chapter.) All blocks – allocated and unallocated – have an 8 byte header that holds block size information and pointers for a singly-linked list of blocks. An allocation request of n bytes results in n+17 bytes being allocated. The 9 additional bytes are a header used to hold fragment-specific data such as the fragment entry point. The first instruction in each fragment is double-word aligned within the cache memory space. An example of a Fragment Cache memory layout is shown in Figure 28.

The Fragment Cache interface provides support for a limited set of primitive actions.

1. Space can be requested via an allocation request. An allocation request specifies the number of bytes requested and the fragment entry point.
2. Allocated space can be freed on a per-fragment basis by providing the fragment entry point. When space for a fragment is freed, space for the associated linker stubs is freed also.
3. The Fragment Cache can free all allocated space, *i.e.*, a Fragment Cache flush. This can be in response to an external request or initiated by the Fragment Cache itself (an internal request) as a replacement policy.
4. Given an address within the Fragment Cache memory space, the Fragment Cache can return the fragment entry point associated with the address.

The interface was designed under the principle that most of the decision-making logic will be located externally to the Fragment Cache.

Even though it is a simple memory manager, the Fragment Cache has some interesting properties, especially when compared to proposed hardware trace-based caching schemes. Since the Fragment Cache has full control of fragment placement, it does not have to adhere to the "hard-wired" placement that is used in most hardware I-caches and trace-based caches. Since fragments are stored in contiguous memory blocks, if the effective size of the Fragment Cache is less than or equal to the size of the machine I-cache, there will be no collision or capacity misses in the machine I-cache. Contrast this with the hardware schemes. The trace cache [Rotenberg et al. 1996] and the schedule cache [Banerjia et al. 1998] use "hard-wired" placement and so could suffer from conflict misses or use set-associativity as a circumvention[13]. The size of an instruction trace in the trace cache is limited by the [hardware] cache line size. The Fragment Cache imposes no limits on fragment size. (The Trace Selector uses a limit of 1K instructions per trace but this limit was never encountered in any of our experiments.) The schedule cache permits variably sized traces but suffers from conflict misses due to its hard-wired placement. Perhaps most compelling, however, is the fact that, like all of Dynamo, the Fragment Cache is transparent to the entire computer system.

The next two sections describe Fragment Cache management techniques constructed using the Fragment Cache primitives.

10.2 Working-set based Fragment Cache flushing

This section describes a preemptive Fragment Cache replacement strategy based on the changes in the working set of fragments. Under normal Dynamo execution, a fragment remains in the Fragment Cache for the remainder of [Dynamo] execution from the time it is placed into the

[13] The DIF cache [Nair and Hopkins 1997] is another hardware-based trace caching scheme, but the literature does not provide enough detail on its organization for us to make any comparisons.

Fragment Cache. (An exception is static-to-secondary fragment conversion with 2-LEVEL trace selection.) Therefore, the Fragment Cache must be able to hold all fragments generated over the entire execution. The standard approach in dynamic translation systems has been to size the software code cache large enough to hold all translations, because code cache replacement and associated translation unlinking are costly [Cmelik and Keppel 1993].

Dynamo's Fragment Cache and Fragment Linker support single-fragment replacement. A single-fragment replacement capability can be used to construct a wide range of replacement heuristics. Other Dynamo components can explicitly request the Fragment Cache to expel a specific fragment (as done in 2-LEVEL trace selection), or the Fragment Cache can choose which fragments to discard when it cannot find enough space to satisfy an allocation request. The latter is *reactive* behavior since the Fragment Cache initiates the replacement in response to an out-of-space condition.

Although the Fragment Cache and Fragment Linker are capable of efficient single-fragment replacement, it turns out that the capability is not particularly useful. Dynamo uses fast, simple memory allocators and attempts to allocate memory in a way so that there will be minimal fragmentation. (Dynamo's memory allocators are described in detail in the Dynamo System Development Tools and Utilities chapter.) A lack of fragmentation allows Dynamo to avoid potentially costly garbage collection (GC) and the logic required to manage an out-of-memory condition at points in Dynamo where memory is allocated. When the Fragment Cache selectively discards fragments to satisfy an allocation request, all memory blocks associated with the discarded fragments are freed. The blocks from the different fragments are possibly at non-contiguous locations, which can cause Dynamo's memory pools to become fragmented, effectively nullifying the efforts to prevent fragmentation. For this reason, the Fragment Cache discards all fragments when an out-of-space condition occurs. We term this *reactive flushing*. If the Fragment Cache is sized too small, it may be flushed constantly and thrashing will occur. Similar to trace selection rate-based bailout, Dynamo monitors the flushing rate, and if the rate exceeds a preset threshold, Dynamo bails out. The threshold depends on the size of the Fragment Cache, since the flushing rate is closely tied to the size.

10.2.1 The case for Fragment Cache flushing

Regardless of its replacement capabilities, requiring the Fragment Cache to hold all fragments created over an entire execution is counter to the adaptive, dynamic nature of Dynamo. Ideally, the Fragment Cache should hold only fragments needed for the current phase of program execution. For a program that executes across multiple, distinct phases that exercise different regions (what we term *phased execution*), the Fragment Cache should be sized to hold only the largest working set of fragments rather than the sum of all of the fragment working set sizes. An example of phased execution is shown in Figure 29. Intrinsic to this style of operation is for Dynamo to detect changes in the working set or a *phase change*. An example of a phase change is shown in Figure 29 just before the 106s mark. The fragment creation rate (FCR) climbs sharply and remains relatively high for about 3-4s as new fragments are formed and then drops sharply, after which [presumably] the new fragments are executed within the Fragment Cache. When a phase change is detected a cache flush can be performed to discard all fragments from the previous working set. For phased execution, cache flushing results in a negligible amount of fragment recreation. We term flushing based on phase changes *working set-based flushing* or *WS flushing*. There is a fundamental difference between WS flushing and reactive flushing. Reactive flushing flushes the cache only when an out-of-space condition is encountered. WS flushing flushes based on the detection of phase changes, independently of an out-of-space condition. WS flushing can invoke flushing well before a reactive flush would be performed; hence, it is a preemptive technique.

WS flushing can yield two memory-related benefits. First, since the Fragment Cache need hold only the largest working set instead of the sum of all of the working set sizes, it requires less memory. Second, a cache flush resets all of Dynamo's memory pools also, reducing their space requirements as well (since they will hold objects only from the current working set also). There are potential performance benefits also, listed below.

- *Reduction in virtual memory (VM) pressure*: Since less memory is needed for the Fragment Cache, fewer code pages and instruction TLB (I-TLB) entries are required, leading to possible reductions in code page and I-TLB misses. In processors that use a unified I- and D-TLB – like the HP PA-8000 – this can reduce TLB misses caused by data accesses also.
- *Execution of fewer conditional branch instructions*: Since only the current working set is stored, the cache memory space is more compact. This means that inter-fragment control transfers can require the execution of fewer branch instructions, since fewer stub-based links will be required. The performance difference can be significant, since a control transfer through a linker stub requires execution of 4 instructions, including 2 branches, whereas a "direct" transfer requires only 1 instruction (the original modified branch).
- *Reduction in I-cache misses*: The I-cache footprint of the cache memory space is smaller. This can reduce the number of I-cache misses particularly conflict misses. This effect will not occur on processors where the Fragment Cache fits entirely within the I-cache. However, for processors with relatively small caches, this could provide a significant performance boost.

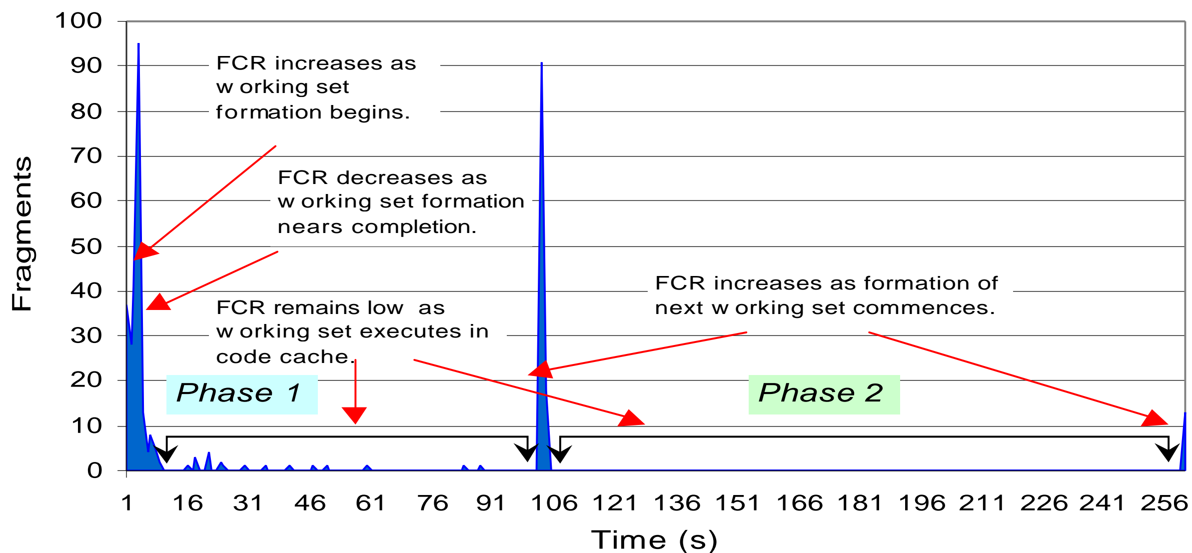


Figure 29: Fragment creation rate pattern for m88ksim

A cache flush can have an intriguing side effect: a regenerated fragment could be constructed along a completely different path than before. A trace/fragment for a starting address *X* could be selected along a [different] path that is more beneficial during the current [upcoming] phase than the earlier version of the trace/fragment. The Trace Selector may or may not select the trace along a different path. However, if the Trace Selector can adapt to changes in program behavior it has the potential to reform fragments profitably.

10.2.2 Selecting a time to flush

Since WS cache flushing can have positive effects, the issue is how to detect phase changes given that such times could be good flush points. One type of a phase change is a change in the set of paths exercised by the program. (Note that this is not necessarily in conjunction with the selection of new traces, as the different paths can already exist across the current set of cache-

resident fragments.) Dynamo cannot observe this type of behavior since it cannot view changes in branch outcomes. This is an area where microarchitectural support could assist Dynamo. The processor could provide hardware support to detect changes in branch behavior within the processor core.

Another type of phase change is the type associated with phased execution as defined previously: new regions of code are executed. As execution commences through the new regions, the FCR will increases dramatically as a new set of fragments is created. This is depicted at three separate points in Figure 29: just prior to Phase 1 and Phase 2 and near the end of the time scale. Dispatch can detect a change in FCR using one of several methods: a timer interrupt to keep an exact measure of the number of fragments created within fixed periods of time; the `time()` function to measure the elapsed time between successive creations of *N* fragments; count the number of executions of the interpreter as a measure of virtual time, and use the interpretation count (IC) to derive FCR. The state diagram in Figure 30 shows the precise operation of the phase change detection logic. The first time that FCR rises above a threshold R_{high} indicates that the construction of the first working set has commenced (transition from state A to B). As the working set is formed, FCR remains above R_{high} (state B). When FCR drops below a threshold R_{low}, this means that working set construction has completed (transition from state B to C); the working set will now execute in the Fragment Cache (state C). When FCR once again exceeds R_{high}, construction of the next working set has begun (transition from state C to B). As part of the state C-to-B transition, Dispatch instructs the Fragment Cache to flush all fragments. The detection of the high-to-low-to-high transition in FCR triggers the flush.

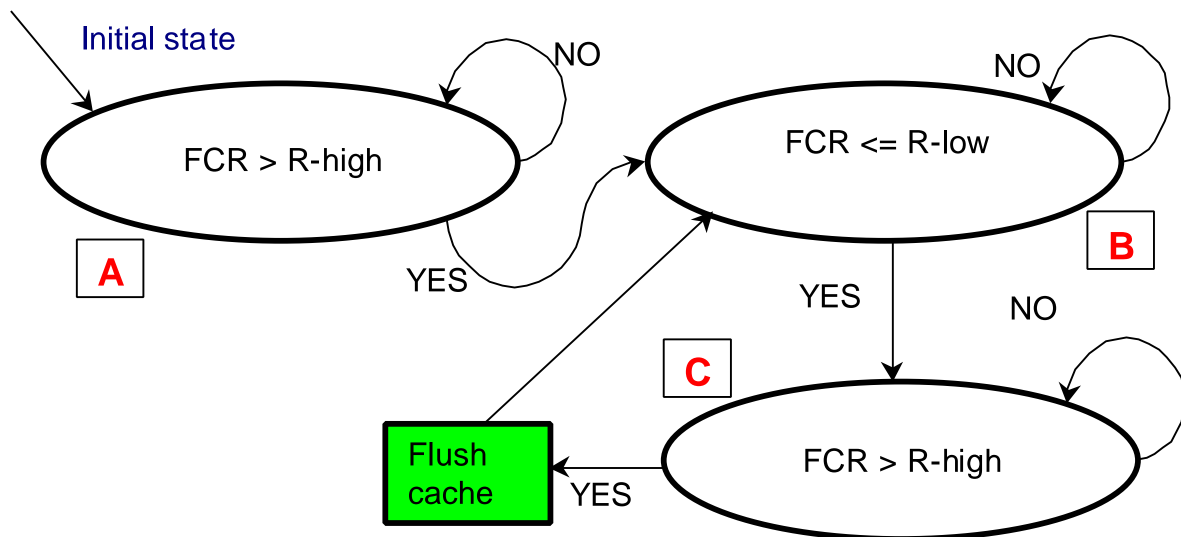


Figure 30: State diagram for phase change detection logic

The thresholds R_{high} and R_{low} have to clearly distinguish working set construction from other aspects of Dynamo execution. If they are set too aggressively, Dispatch will initiate an excessive number of flushes; this could cause a dramatic increase in execution time. If they are set too conservatively, no Fragment Cache flushing will be performed. We experimented with a range of values for R_{high} and R_{low} and found that it suffices to have one value R where R=R_{high}=R_{low}. We term R the *working set threshold* or *WS threshold*. Using different thresholds to signal the beginning and end of working set construction did not cause the timing or frequency of flushing to differ significantly. As for detecting changes in FCR, we experimented with both the `time()`

function and the IC to derive FCR The `time()` function permits precise control of flush timing but can be perturbed by factors that vary across different executions, such as I- and D-cache misses and TLB misses. Using IC is less precise and causes much more flushing, including multiple flushes in the midst of working set construction. However, IC-based flushing is deterministic; the flushes for a Dynamo execution of a program *P* occur at identical times across multiple executions, since the pattern of interpretation is precisely the same across all executions. Because of its deterministic nature, IC-based flushing is used within Dynamo. Also, we found that detection of a high-to-low transition is slightly more effective than detection of a high-to-low-to-high transition. The high-to-low transition is detected just slightly earlier than the latter part of the high-to-low-to-high transition. Therefore, the flush is initiated earlier. Although there is a danger that the flush could be premature, our experiments showed that any such effect is negligible. Memory usage reductions are larger and run-time performance is better when using high-to-low timing.

We found it useful to tie the WS threshold to the hot threshold used by the Trace Selector, since the amount of interpreter execution is tied to the threshold. We derived the WS threshold by multiplying the hot threshold by a constant factor that we term the *flush factor*. A flush factor of 16 is used across all programs. The value was determined empirically by experimenting with a range of values.

10.2.3 Performance: memory usage

This section presents data on memory usage reductions due to WS flushing. Some of Dynamo's memory pools are used for allocations only once across an execution: memory is allocated from them only at Dynamo initialization. These are termed static pools. Other memory pools are allocated from continuously – dynamically – during Dynamo execution. (The Fragment Cache is a dynamic pool. Details on static and dynamic pools are discussed in the Dynamo System Development Tools and Utilities chapter.) WS flushing can alter the usage pattern of a dynamic pool but not of a static pool. Therefore, only changes in the usage of dynamic pools are shown.

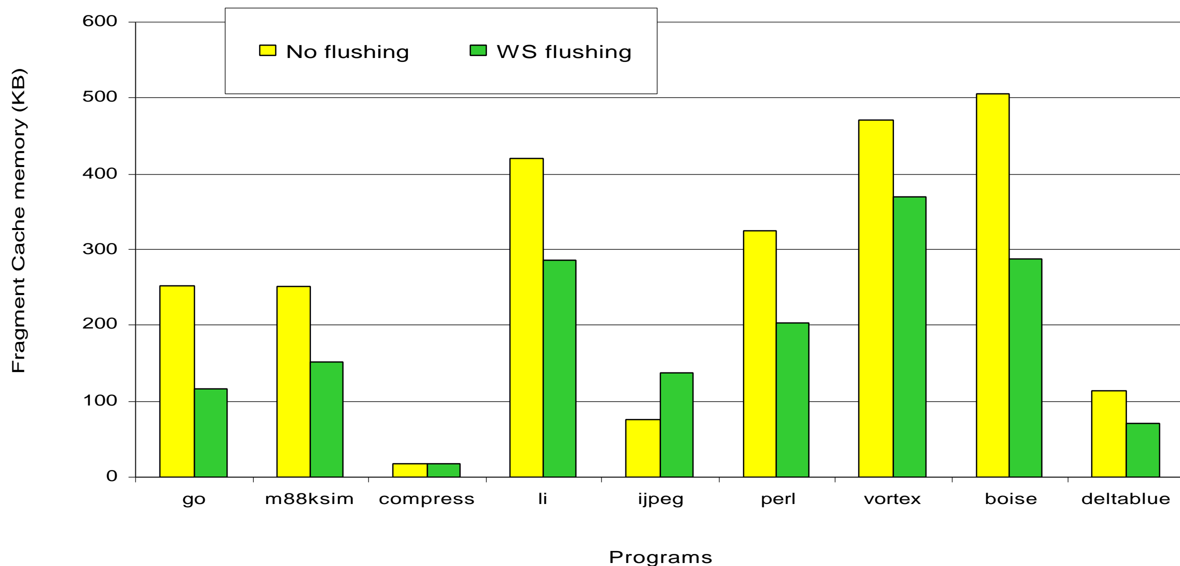


Figure 31: Fragment Cache memory usage with and without WS flushing

Figure 31 shows the reductions in maximum Fragment Cache memory usage with WS flushing. Each bar represents the maximum size of the Fragment Cache during the specified execution. The following programs bailed out: go, vortex, and ijpeg when WS flushing was not

used, and go, vortex, ijpeg, and boise when WS flushing was used. Average Fragment Cache memory usage is reduced from 270 KB to 182 KB. The reductions range from an increase of 61 KB for ijpeg to 219 KB for boise. ijpeg's behavior is due to the fact WS flushing causes the Fragment Cache size to fluctuate during the period before Dynamo bails out. ijpeg bails out at a different point in time when WS flushing is used, at which time the Fragment Cache is larger than when bailout occurs without WS flushing. In general, for programs that bail out both with and without WS flushing, it is indeterminate if the Fragment Cache will be larger or smaller with WS flushing. However, the data shows that for go and vortex, the Fragment Cache size is smaller. (The flush factor was chosen so that the average Fragment Cache memory usage across all programs is minimized, including cases where programs bail out.) For the five programs that do not bail out, WS flushing always yields a smaller maximum Fragment Cache size with an average reduction from 226 KB to 146 KB.

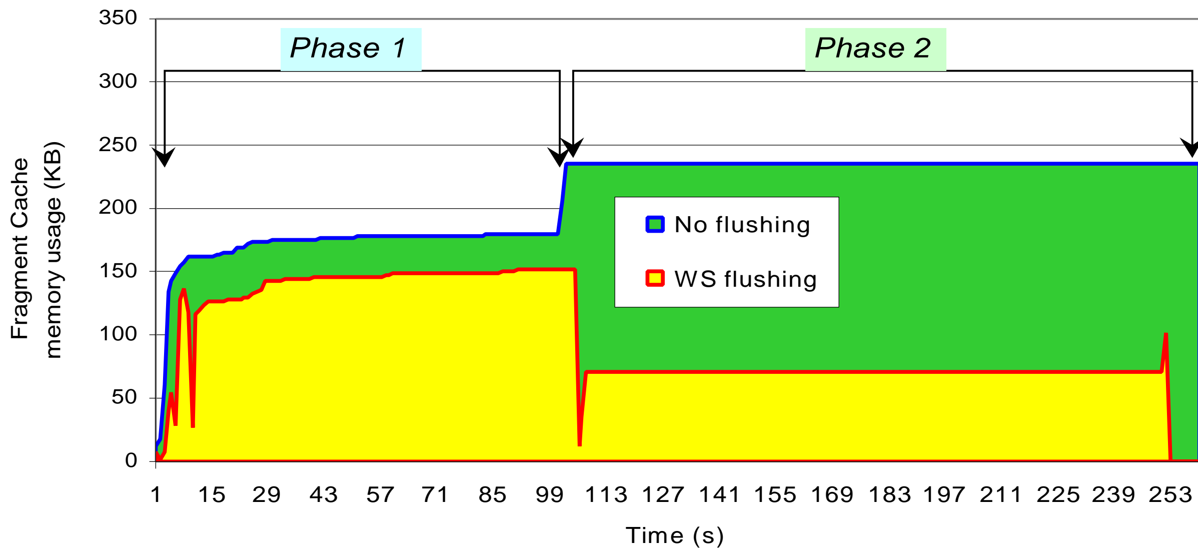


Figure 32: Dynamic changes in Fragment Cache memory usage with WS flushing in m88ksim

Figure 32 shows the dynamic changes in Fragment Cache memory usage for m88ksim, with and without WS flushing. In the absence of WS flushing, the Fragment Cache size increases monotonically as execution progresses. At the phase change between Phases 1 and 2 (around the 99s mark), the memory usage increases from 180 KB to 236 KB. With WS flushing, flushing occurs when the phase change is detected, and memory usage decreases from 151 KB to 70 KB. WS flushing causes multiple flushes at the beginning of program execution (as evidenced by the fluctuation in memory usage between the 1s and 15s marks). In doing so, Fragment Cache memory usage for Phase 1 is smaller when WS flushing is used – 146 KB versus 180 KB. This has the significant effect that WS flushing does not simply reduce the maximum Fragment Cache memory usage to the footprint of the largest working set. It can actually reduce the footprint of the largest working set itself. For m88ksim, reducing the maximum memory usage to the size of Phase 1's footprint (180 KB) yields a 23.7% reduction. WS flushing reduces the size of Phase 1's working set – and the overall Fragment Cache memory usage – to 146 KB, which reduces the maximum memory usage by 38.1%, a significant additional reduction.

Considering all programs in our benchmark suite, WS flushing reduced the maximum Fragment Cache memory usage from 506 KB (for boise) to 369 KB (for vortex, which bails out). The practical significance is that with WS flushing Dynamo can allocate a Fragment Cache that is 137 KB smaller. Restricting the analysis to the programs that do not bail out, the maximum Fragment Cache usage is reduced from 420 KB to 286 KB (both for li). Note that boise bails out

when WS flushing is used but does not when WS flushing is not used. When WS flushing is used, boise's fragment creation rate is similar to those for go, ijpeg, and vortex and meets the bail out criteria. Figure 33 shows the reductions in memory usage for all dynamic pools with WS flushing (this includes Fragment Cache memory). On average, memory requirements are reduced from 508 KB to 339 KB. The maximum dynamic pool memory usage is reduced from 932 KB to 646 KB, a reduction of 30.7%. As with Fragment Cache sizes, the practical significance is that with WS flushing Dynamo needs less memory – 286 KB less – for its dynamic pools.

The data shows that WS flushing enables significant reductions in memory usage. Although the reduction in maximum dynamic pool memory of from 932 KB to 646 KB is impressive, 646 KB could be an unreasonable amount of memory for some embedded devices. For this reason, WS flushing could be more applicable in devices where a small memory footprint is not a primary issue but is an important secondary concern, such as enterprise-level computer systems.

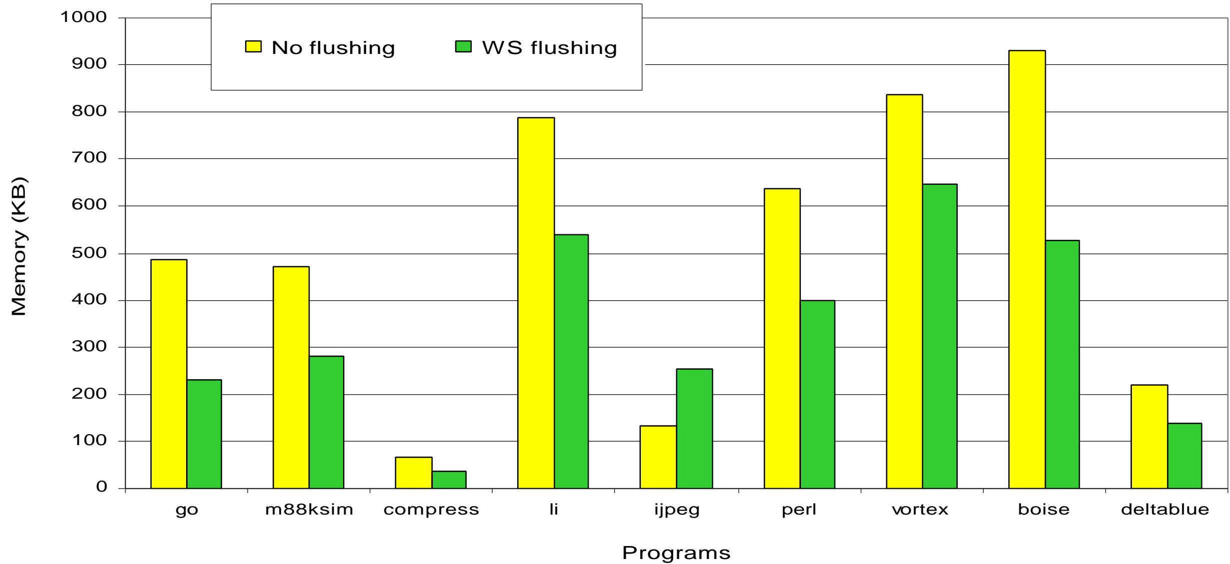


Figure 33: Memory usage in dynamic pools with and without WS flushing

10.2.4 Performance: speedups

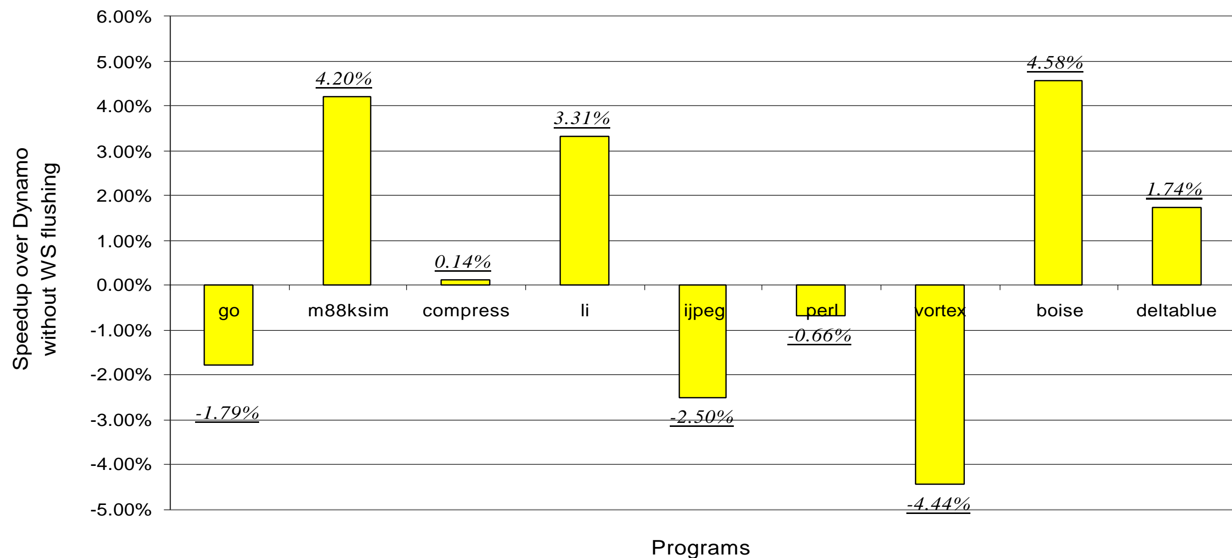


Figure 34: Run time performance of WS flushing compared to no WS flushing

This section discusses run time performance for WS flushing. Figure 34 shows the run-time performance for WS flushing versus not using WS flushing (not for WS flushing versus native execution). The interesting fact is that WS flushing yields a slight average speedup of 0.51%. Even though the Fragment Cache is flushed repeatedly – albeit intelligently – the average run time does not increase and in fact decreases slightly! For the five programs that do not bail out – m88ksim, compress, li, perl, and deltablue – the average speedup is 1.74%. boise experiences a speedup of 4.58% because it is able to bail out, which it cannot do without WS flushing. go, ijpeg, and vortex all bail out (just as without WS flushing) but do so at later points in time and so suffer slowdowns.

Figure 35 shows the number of times the Fragment Cache is flushed for the programs that do not bail out. Most programs can tolerate a large number of flushes. The Fragment Cache is flushed 46 times for m88ksim and li; both programs experience speedups in excess of 3%. The Fragment Cache is flushed 37 times for perl, which causes only a 0.66% slowdown.

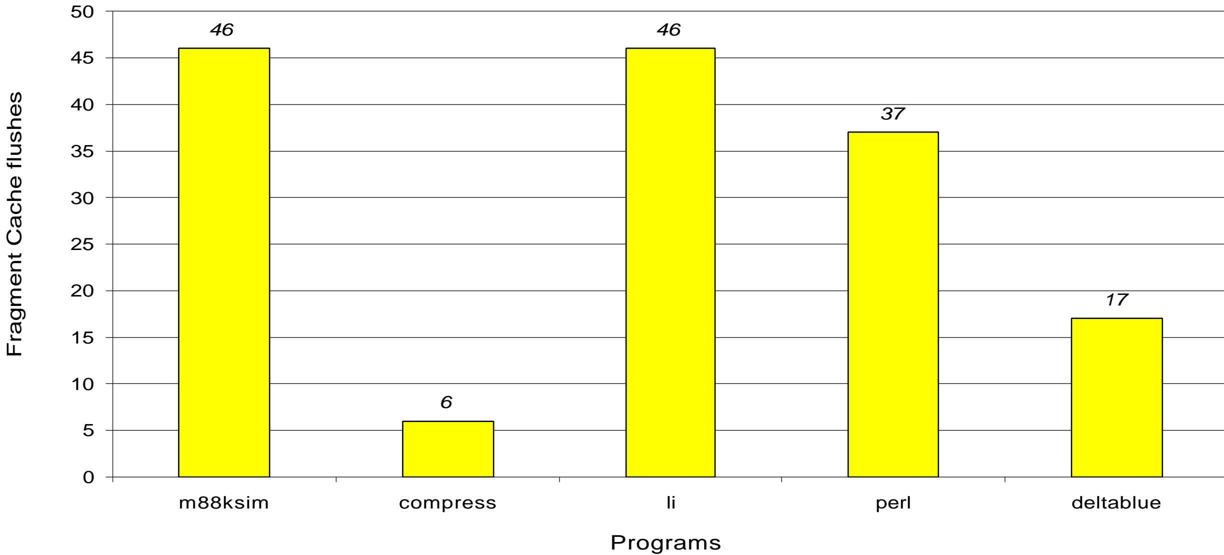


Figure 35: Number of flushes caused by WS flushing

The data in Figure 34 indicates that WS flushing can improve runtime performance. This leads to the issue of whether the timing of the flush is as important as the flush itself. To test the importance of timing, we configured Dynamo so that the Fragment Cache is flushed at regular intervals. Specifically, the Fragment Cache is flushed after every *N* fragments are formed. We term this *interval-based flushing*. For example, *N*=200 means the Fragment Cache is flushed after the first 200 fragments are formed, after the second 200 fragments are formed (400 fragments total), after the third 200 fragments are formed (600 total), and so on. The effect is to limit the Fragment Cache's capacity to *N* fragments. Bail out was disabled for this experiment. Results are presented in Figure 36. The data shows that for some combinations of program and *N*, interval-based flushing is competitive with or superior to WS flushing. However, the general trend is that WS flushing yields far better performance. The average speedups (compared to no flushing) for intervals of 200, 300, 400, 500, and 600 are –15.06%, -5.14%, -4.79%, -3.68%, and –3.74%, respectively, all of which are slowdowns. The data confirms that flushing can be harmful if not performed at appropriate times.

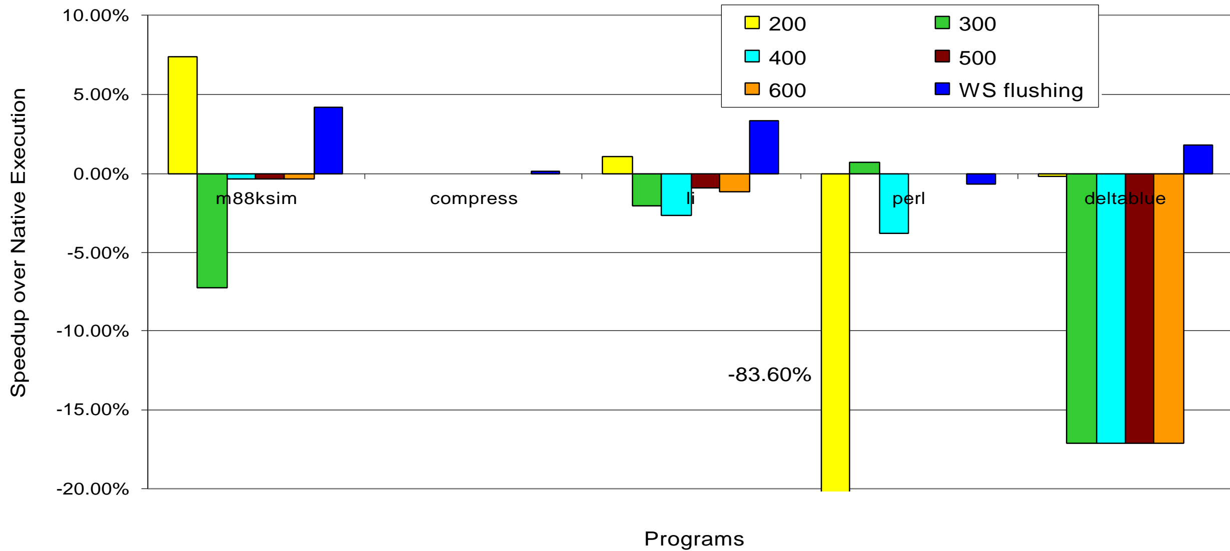


Figure 36: Run time performance of interval flushing versus WS flushing

WS flushing yields speedups for reasons outlined previously: reductions in VM pressure, branch execution, and instruction cache misses. Precisely measuring individual effects is difficult in our software-only prototype. For example, a change in TLB behavior cannot be measured precisely, since Dynamo does not have access to any hardware that monitors TLB behavior. However, we can estimate the behavior of certain aspects of the system using the Dynamo system profiler. (Details of the profiler are presented in Dynamo System Development Tools and Utilities chapter). Figure 37 presents data on the number of Fragment Cache pages referenced per second with WS flushing. Across all programs, fewer pages are referenced when using WS flushing. The reductions range from less than 1 page for compress to 5 pages for li and 6 pages for m88ksim. WS flushing can reduce the amount of branch execution. Dynamo does not have a non-invasive method for measuring the amount of branching that takes place within the Fragment Cache. However, the static metric of the amount of stub-based linking can be used as a coarse estimate. Since WS flushing causes the creation of more fragments and a greater absolute amount of fragment linking, comparing absolute amounts of linking is not illuminating. However, the average number of links per fragment normalizes the metric. The data is presented in Figure 38. Flushing significantly reduces the amount of stub-based links, with an average reduction of 62.7%. The absolute reductions range from 0.06 for compress to 0.55 for deltablue and 0.64 for li.

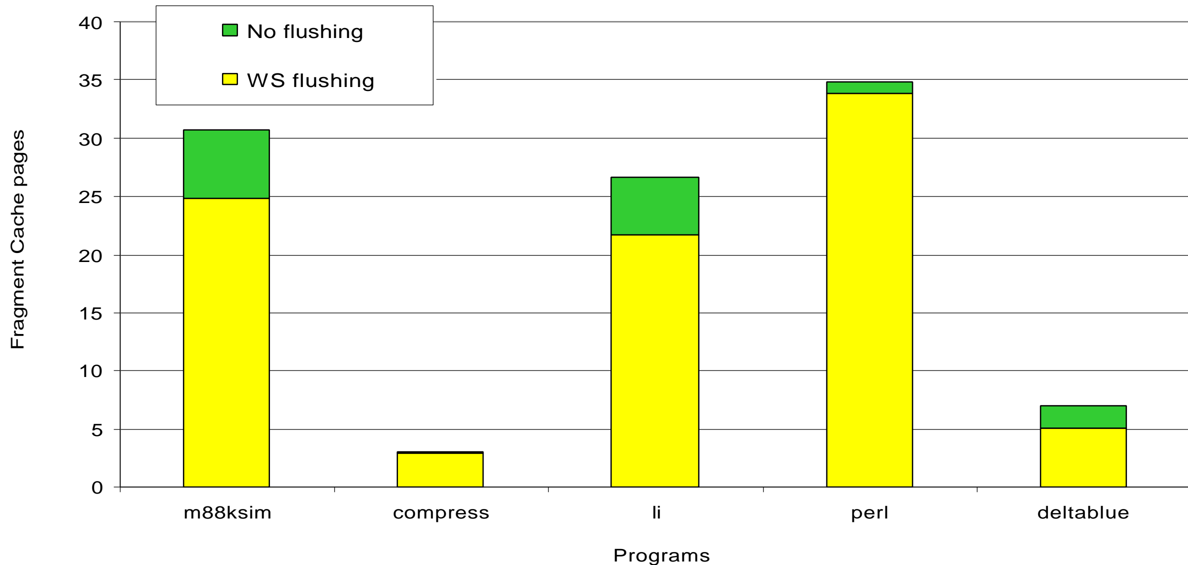


Figure 37: Fragment Cache page usage with and without WS flushing

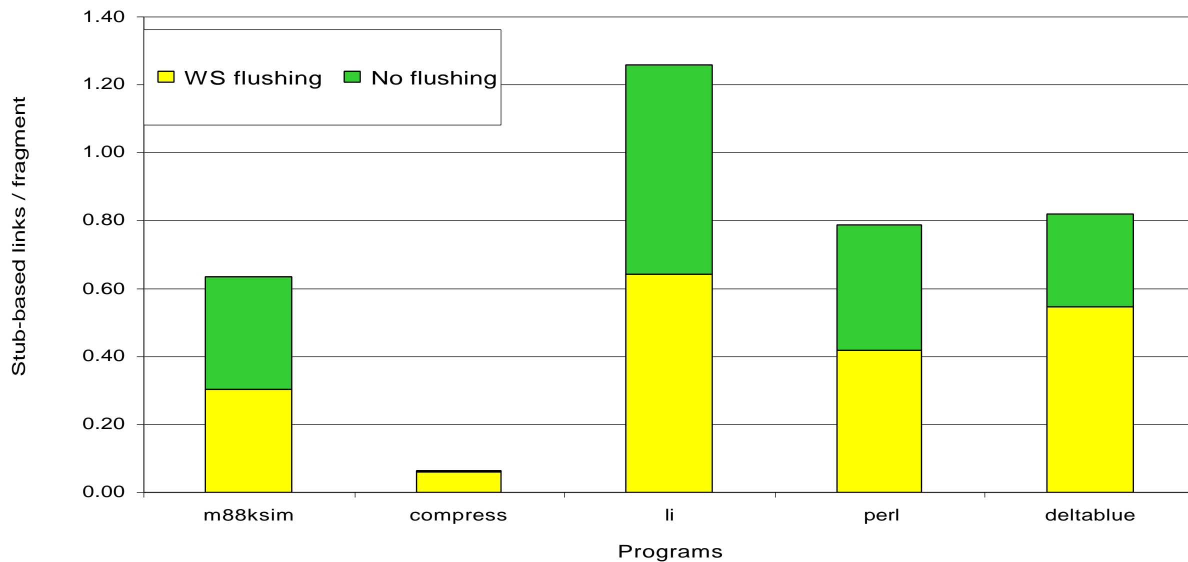


Figure 38: Stub-based linking with and without WS flushing

WS flushing reduces memory requirements and improves run-times. Given these beneficial effects, WS flushing could be a default Dynamo setting for non-memory constrained devices such as enterprise-level computer systems.

10.3 Alternative placement strategy

The previous section assumed that all fragments were placed into contiguous locations in the Fragment Cache. Such a placement strategy is appropriate for a trace selection scheme that creates only one type of fragment. This is the case for the Dynamo selection scheme and for static selection and basic block selection also. However, 2-level hierarchical trace selection (2-LEVEL) creates two fundamentally different types of fragments, static fragments and secondary fragments. (For details on 2-LEVEL selection, see the Trace Selector chapter.) The split cache

implements a code layout strategy for 2-LEVEL trace selection. The split cache leverages the fact that two different types of fragments are created to guide code placement decisions.

10.3.1 Functionality and behavior

A static fragment/trace component is formed to indicate that a branch target has become hot and is a candidate to be included in more frequently executed code sequences. A secondary fragment is formed from trace components when a secondary trace head meets a second set of criteria. Under these criteria, secondary fragments are by definition "hotter" than static fragments. The *split cache* is an allocation scheme that places static fragments and secondary fragments into different areas of the Fragment Cache memory space. Contrast this with the standard allocation scheme – termed the *unified cache* – in which space is allocated sequentially irrespective of any characteristics of the fragments. The split cache has several potential advantages over the unified cache, similar to those for WS flushing: a reduction in VM pressure, execution of fewer branch instructions, and a reduction in I-cache misses. The rationale for the advantages varies slightly, as explained below.

- *Reduction in VM pressure*: Since static and secondary fragments are not interspersed within the memory space, fewer code pages and instruction TLB (I-TLB) entries are required for secondary fragments. This can lead to a reduction in TLB pressure, just as with WS flushing.
- *Execution of fewer conditional branch instructions*: Since secondary fragments are stored in a more compact memory space, inter-fragment control transfers (between secondary fragments) can require the execution of fewer branch instructions, since fewer stub-based links will be required.
- *Reduction in I-cache misses*: The I-cache footprint of the secondary fragments is smaller since static fragments and secondary fragments do not share cache lines. This can reduce the number of I-cache misses.

The implementation of the split cache is no more complicated than the unified cache. For every space allocation request, the Fragment Cache is passed a parameter that indicates whether the fragment is static or secondary. The parameter guides the placement decision. In our implementation, static fragments are allocated upwards from low addresses and secondary fragments downwards from high addresses.

10.3.2 Performance

This section presents data that illustrates the performance of the split cache. The six programs that do not bail out – m88ksim, compress, li, perl, boise, and deltablue – are examined. Figure 39 shows speedups for a split cache over a unified cache. The split cache provides a consistent performance gain over the unified cache. The speedups range from 0.22% to 6.24%, with an average of 3.13%. The only program that does not improve performance is li, which suffers a slight slowdown (0.68%).

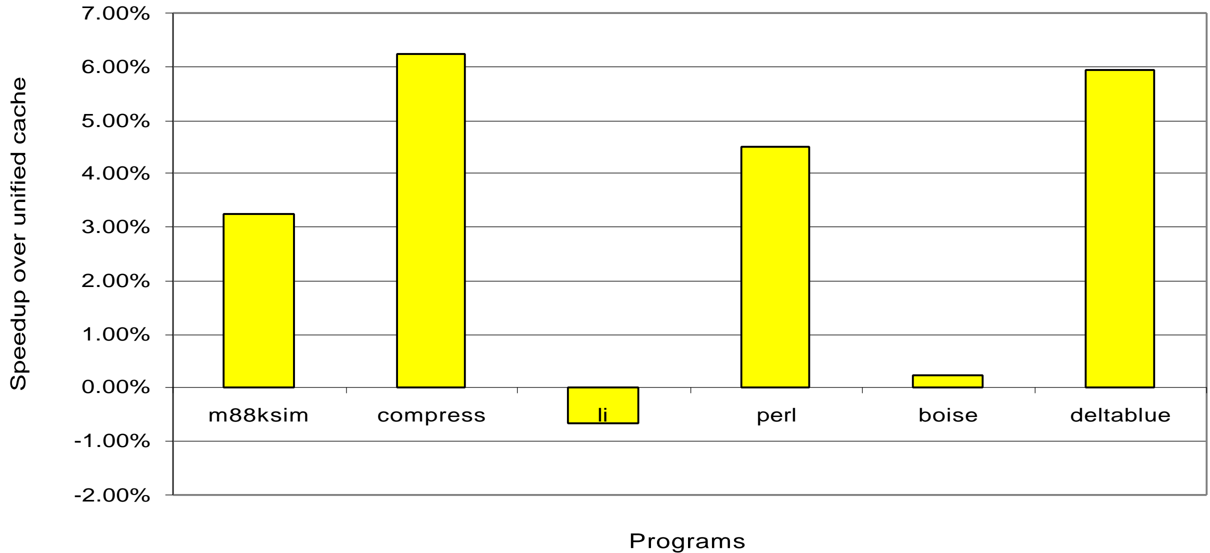


Figure 39: Run time performance of the split cache

Figure 40 shows the average number of Fragment Cache pages referenced per second for a split cache and a unified cache. For all six programs, fewer pages are referenced when using the split cache. The reductions range from almost 7 pages for perl to less than 1 page for compress and boise. Examination of Figure 39 and Figure 40 reveals that there is not a direct correlation between a reduction in Fragment Cache page usage and speedup. Page references by compress and deltablue are reduced by less than 1 page/s yet these two programs experience the greatest performance gains – 6.24% and 5.94%, respectively. Page references by boise are reduced by almost 4 pages/s yet boise experiences only a 0.2% speedup. For a program that places a lot of pressure on the VM system, a small decrease in page usage can lead to reduction in page and TLB misses, which in turn can cause a disproportionately large performance gain. For example, compress's data references cause a great deal of TLB pressure on the PA-8000 processor. A small reduction in page references can have a major impact on TLB pressure and thus improve performance [Hsu 1999].

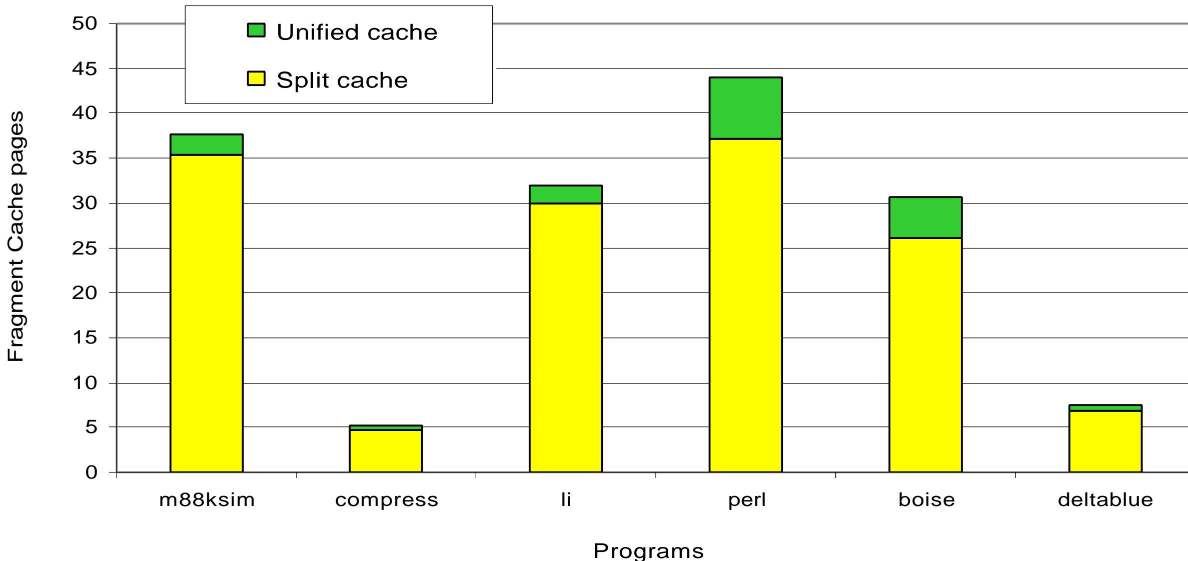


Figure 40: Fragment Cache page usage with and without the split cache

Another potential advantage of the split cache is a reduction in the number of conditional branches executed. Figure 41 show data on stub-based linking between secondary fragments. There is a significant reduction for most programs. The split cache reduces the number of stub-based links by an average of 13.4%, ranging from 3.7% for perl to 19.6% for m88ksim. The data shows clearly that fewer static branch instructions are required for inter-fragment transfers when using the split cache.

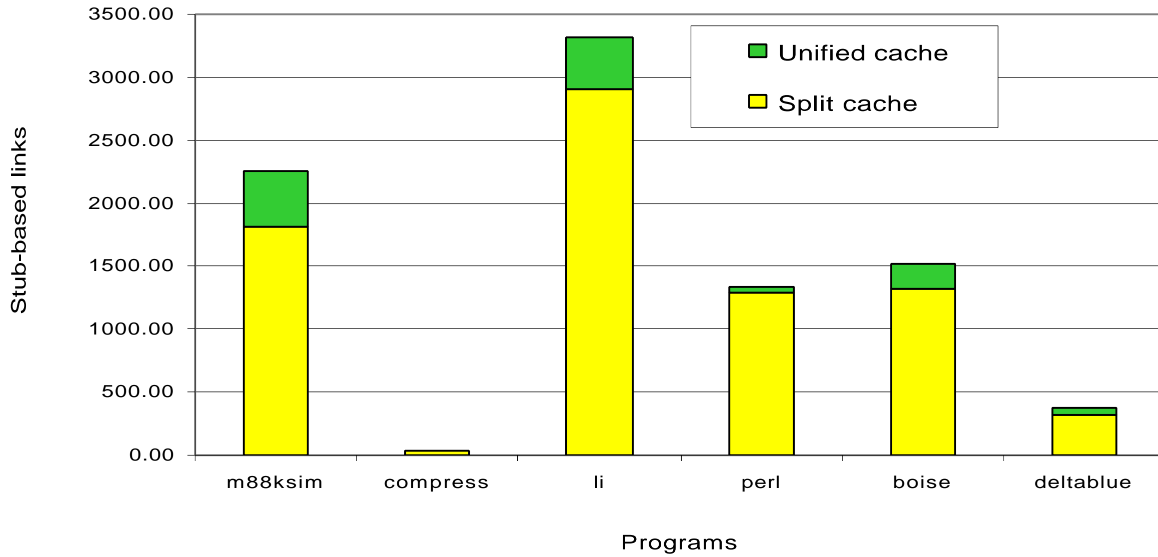


Figure 41: Stub-based linking with and without the split cache.

11 Signal Handler

Dynamo has to intercept all signals delivered to the program being dynamically optimized. Otherwise, if the program arms a signal with its own handler, the operating system will directly invoke that handler upon arrival of the signal, and Dynamo will lose control over the program. Signals that are armed by a program create a problem because the signal handler can examine or even modify the machine state (also called the *sigcontext*) passed to it. If a signal arrives when control is in the Fragment Cache, the machine state at the instant of the signal may not correspond to any valid state in the original program, since the code in the Fragment Cache was generated by Dynamo. The program's signal handler code may fail if it is passed an inaccurate *sigcontext*. The optimizations done by the Fragment Optimizer complicate this situation even further.

Dynamo's approach to signal handling is very similar to that used by the Shade dynamic translator [Cmelik and Keppel 1994]. Dynamo intercepts all system calls in the program code that install signal handlers (such as *sigaction*). The system call is suppressed, and instead Dynamo installs its own signal handler for the signal. The original program's handler address is recorded in a table maintained by Dynamo. The Signal Handler component in Figure 3 (page 20) is the master signal handler within Dynamo. All signals armed by Dynamo are handled by this component. When a signal arrives, the Signal Handler is automatically invoked by the kernel, instead of the program's signal handler. The program's signal handler code is then executed under Dynamo control, similar to other program code, so that Dynamo retains control upon return from the program's signal handler.

Signals can be classified as asynchronous and synchronous. Asynchronous signals arise from events like keyboard interrupts and timer expiry, and are generally not associated with a specific PC. This allows the *delivery* of such signals to the program code to be postponed. The Signal Handler takes advantage of this, and when an asynchronous signal arrives, it simply records its arrival in a FIFO queue and jumps back into the Fragment Cache to resume execution. When control eventually traps to Dispatch during the next Fragment Cache exit, Dispatch checks the FIFO queue of pending asynchronous signals, and processes them. When control is outside the Fragment Cache, the program's context in the context save area of Dynamo memory is always in a form that can be mapped back to the original program code in a straightforward way. In other words, Dynamo can guarantee a faithful *sigcontext* to the program's signal handler (as if the program were executing directly on the underlying processor) as long as control is not in the Fragment Cache.

In order to bound the latency of asynchronous signal handling, the Signal Handler unlinks all exit branches in the current fragment (the one that was executing when the asynchronous signal arrived). This prevents control from spinning within the Fragment Cache for an arbitrarily long period of time before Dispatch gets to process the pending signal queue. This is especially useful in environments where a soft real time response is necessary.

Synchronous signals arise from events like memory faults (like SIGSEGV for example). Such signals are associated with a specific faulting instruction PC, and cannot be postponed. Unfortunately, these signals can occur when control is at a point in the fragment where the program's context (currently in the underlying processor's registers) cannot be easily mapped back to the original. The optimizations done by the Fragment Optimizer further complicate this situation. Most programs do not arm synchronous signals since execution generally aborts in such cases, and even among those that do, the machine state at the instant of the signal is seldom examined, so delivering an unfaithful context is usually not catastrophic [14].

The need for precise exception delivery creates restrictions on the kind of optimizations that Dynamo can perform. The more aggressive the optimization, the harder it is to undo. However,

[14] An important exception is when the program is a debugger (debugging another program).

Dynamo has an important advantage over a static compiler in this regard. Dynamo's fragments are very simple straight-line code sequences where control can only enter at the top. The simplicity of control flow within a fragment makes it possible to undo many kinds of optimization that in the case of a static compiler would have been very difficult to undo. The impact of precise exception delivery on optimization was discussed in detail in the Fragment Optimizer section. To recap that discussion, when creating a fragment out of a hot trace, the Trace Selector keeps track of the mapping between fragment cache addresses and application program addresses for the instructions in the fragment. This mapping is used to reconstruct the application program PC corresponding to an exception.

The Fragment Optimizer provides two modes of optimization: a conservative mode and an aggressive mode. Conservative optimizations allow the original program context to be re-created at any point during a fragment's execution, but the performance wins from Dynamo may be lower. By default, the Fragment Optimizer operates in its aggressive mode, but this can be changed at Dynamo-initialization time. Another option is to start out in the aggressive mode, and watch the program code for situations that warrant the use of the conservative mode. If such a situation is detected, the contents of the Fragment Cache can be immediately flushed (causing Dynamo to re-populate it), and the Fragment Optimizer is switched to the conservative mode for the remainder of the run. Currently, information available in a PA-RISC binary is not sufficient to make such a detection. However, a Dynamo-aware compiler could easily communicate such information through annotations in the program binary [Buzbee 1998].

When programs arm synchronous signals in order to continue execution in spite of faults, they usually use *longjmp* to return to an earlier *setjmp* point, rather than a normal return. This causes the kernel to unwind the program's runtime stack to the environment present at the time of the *setjmp* call, and normal execution resumes from that point onwards. Thus, if a program *longjmp* call is allowed to execute directly in the Fragment Cache, Dynamo will lose control. The solution is to intercept the *setjmp* call, and modify its associated "jmp_buf" parameter so that the Dispatch entry point is the *longjmp* target. When a *longjmp* call is executed in the Fragment Cache, control will return to Dynamo once the program runtime stack has been unwound. Dispatch can then look up the original program jmp_buf information and resume executing the program code from the appropriate program PC.

12 Performance Summary

The previous sections presented data on various aspects of Dynamo: the characteristics and resultant behavior of several trace formation heuristics, the potential for optimizations on dynamically formed fragments, and performance aspects of the fragment cache. This section combines the data from the prior sections into a cohesive picture by presenting summary data.

12.1 Experimental framework

All experiments that were performed for the results in this report were conducted on standard, commercially available hardware and software. No modifications were made to the hardware, operating system, compilers or other systems software. An HP PA-8000-based workstation running HP-UX 10.20 was used. The processor is configured with direct-mapped 1 MB instruction and data caches.

Our benchmark suite consists of seven programs from the SPECint95 benchmark suite [SPEC] and two C++ programs. *boise* is commercial C++ code that is used within the rendering pipeline of a color printer. *deltablue* is a C++ implementation of the DeltaBlue incremental constraint solver [Sannella et al. 1993]. All programs were compiled using "off-the-self" HP C and C++ compilers. (g++ was used to compile deltablue as it produced code that executed faster than that produced by the HP C++ compiler.) All results presented to this point in the paper were based on programs compiled with Level 2 (+O2) optimizations and statically linked. Level 2 performs the following optimizations.

- FMAC synthesis.
- Coloring register allocation.
- Induction variable elimination and strength reduction.
- Local and global common subexpression elimination.
- Advanced constant folding and propagation.
- Loop invariant code motion.
- Store/copy optimization.
- Unused definition elimination.
- Software pipelining.
- Register reassociation.
- Loop unrolling.

Reference inputs were used for all experiments. boise used an input data set representative of an actual commercial workload. deltablue was configured to solve 350,000 constraints.

12.2 Experimental results

Figure 42 shows the runtime performance of Dynamo. The average speedup 7.37%. The three programs that bail-out (go, ijpeg, and vortex) are shown in a lighter shade. We refer to this set of experiments as our *baseline experiments*. 6

Figure 43 shows the overhead of Dynamo as a percentage of total program execution time under Dynamo. Again, the programs that bail out are shown in a lighter shade. The average overhead is 1.4%. The overheads are consistently lower for the five programs that experience speedups under Dynamo. The highest overhead among those five programs is 1.1% (li). go and vortex have very low overheads since Dynamo bails out very early; for the period of time before backout occurs, the overheads are 99% and 94% percent. boise's problematic behavior is highlighted by its overhead. A high overhead does not imply that the overhead itself is the cause of the performance degradation. Rather, it is a symptomatic of the fact that Dynamo is unable to successfully optimize the program. For example, boise has 3.3% overhead yet suffers a 9.5%

slowdown. When bailout is disabled, go has a 6.13% overhead yet suffers a slowdown in excess of 150%. Clearly overhead alone is not the reason for the slowdown.

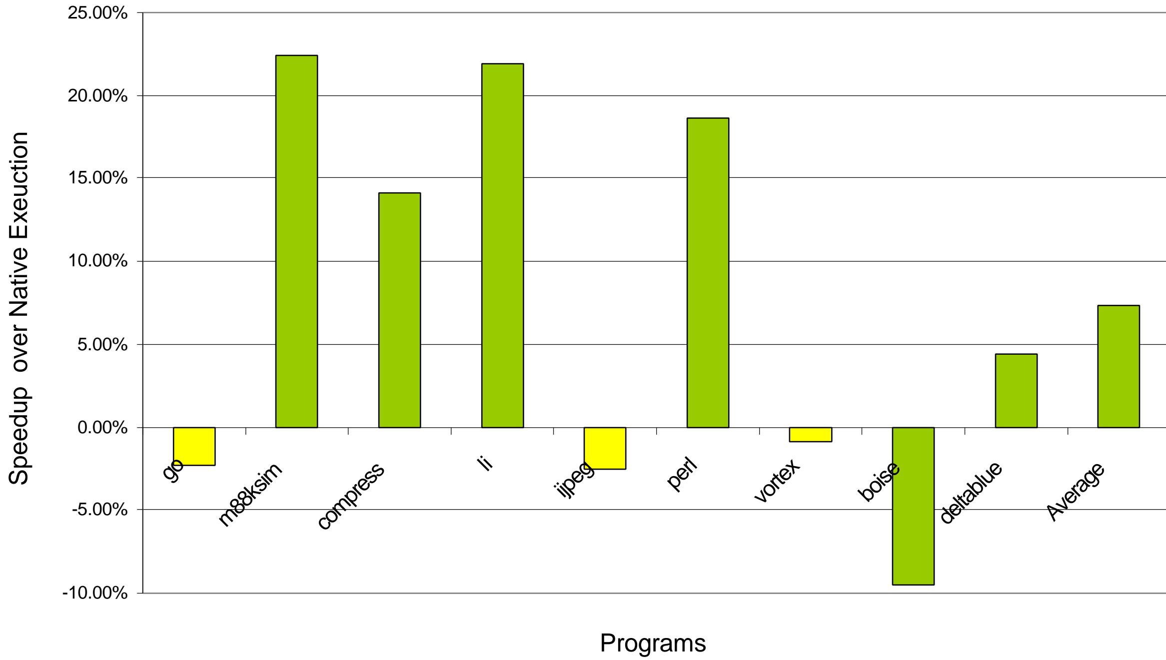


Figure 42: Dynamo performance (light/yellow bars indicate that Dynamo bailed-out)

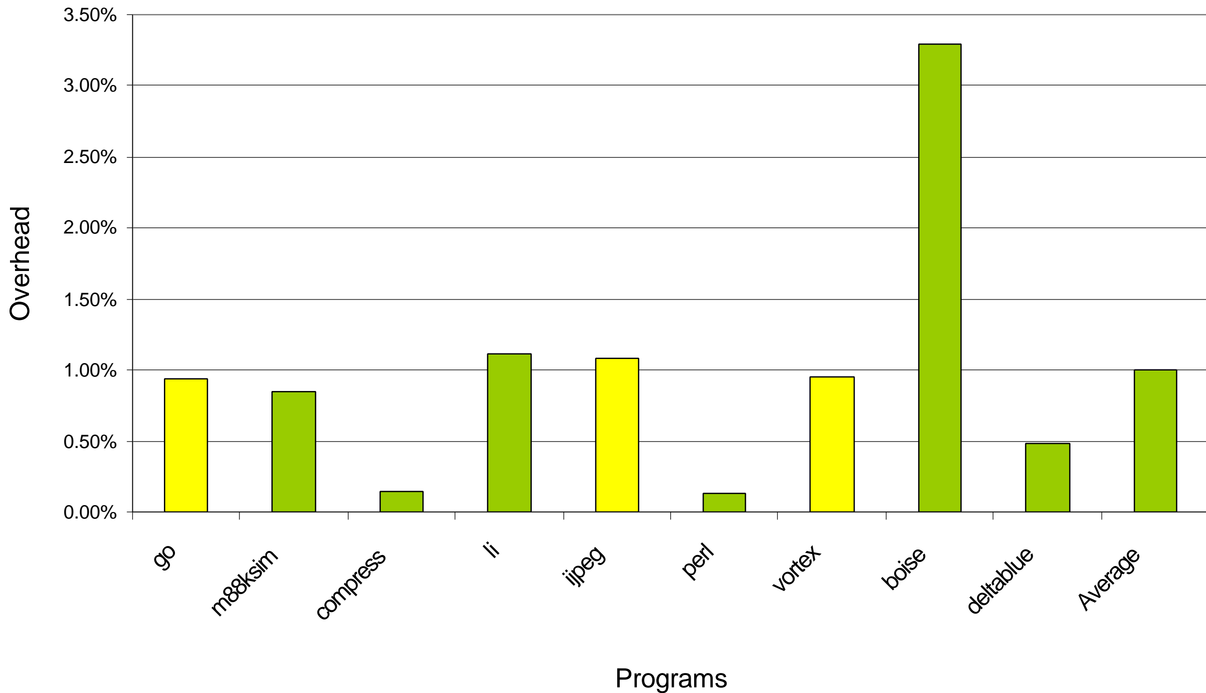


Figure 43: Dynamo overhead (light/yellow bars indicate that Dynamo bailed-out)

12.3 Dynamo code and memory footprint

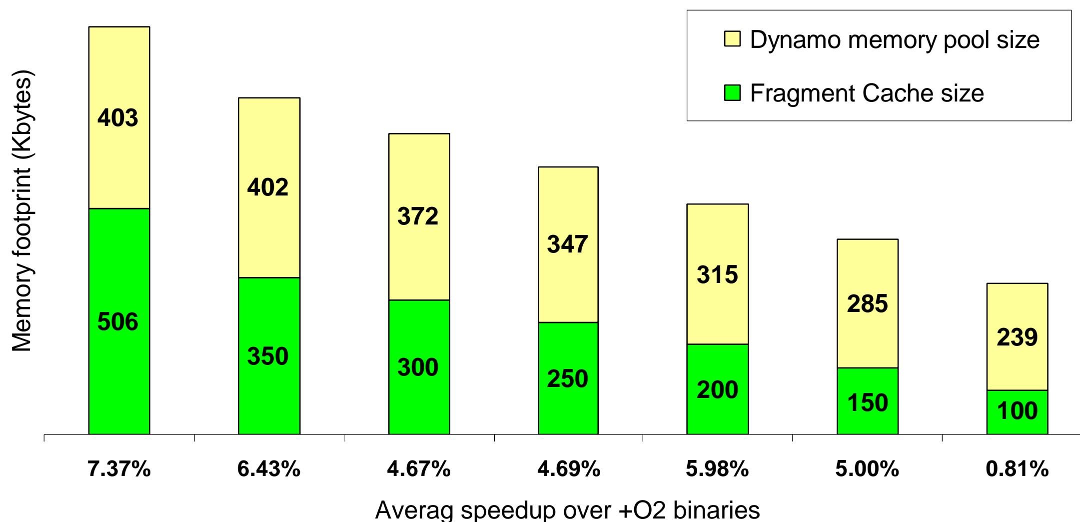


Figure 44: Fragment Cache and Dynamo memory pool footprint

Dynamo has been designed as an *efficient* means to deliver high performance. An important aspect of efficiency is Dynamo's small memory footprint design. Practically every design decision within Dynamo was made with careful attention to the balance between creating a high-performance optimizer framework and doing so with a minimum amount of memory usage. This section presents data on Dynamo's performance with a limited amount of memory. Dynamo's Fragment Cache can be configured at various sizes. With a limited Fragment Cache size several of the benchmark programs will experience capacity pressure in the Fragment Cache, causing cache flushes/fragment replacements.

Figure 44 shows the memory footprint for seven different Fragment Cache configurations. . For each configuration the average speedup over our benchmark set is shown along with a break down of the memory footprint into the Fragment Cache size and the total size of data structures that Dynamo allocates in its own dynamic memory pool. The leftmost bar shows the memory footprint that results if no size limit is imposed on the Fragment Cache. In this case, 506 KBytes in Fragment Cache space is utilized with an average speedup of 7.36%. For a Fragment Cache size of 100 KB, there is a slowdown in excess of 80%. A 100 KB cache is not large enough to hold some of the working sets resulting in thrashing. Increasing the Fragment Cache size to 150 KB is sufficient to prevent thrashing and results in an average speedup of 5%. The use of the Fragment Caches larger than 150 KB yields only incremental improvements. Overall, the 150 KB Fragment Cache provides the best combination of memory usage and run-time performance.

Figure 45 shows the breakdown in the Dynamo code footprint. The total code size is 265 KB. The bulk of the code is PA dependent, which is not surprising since the system operates at a very low level and much of it is dependent on the machine context and machine architectural features. The large fraction taken up by the decoder and interpreter indicate that the PA has a very complex instruction set. On a simpler machine, the footprint of these modules may be much smaller.

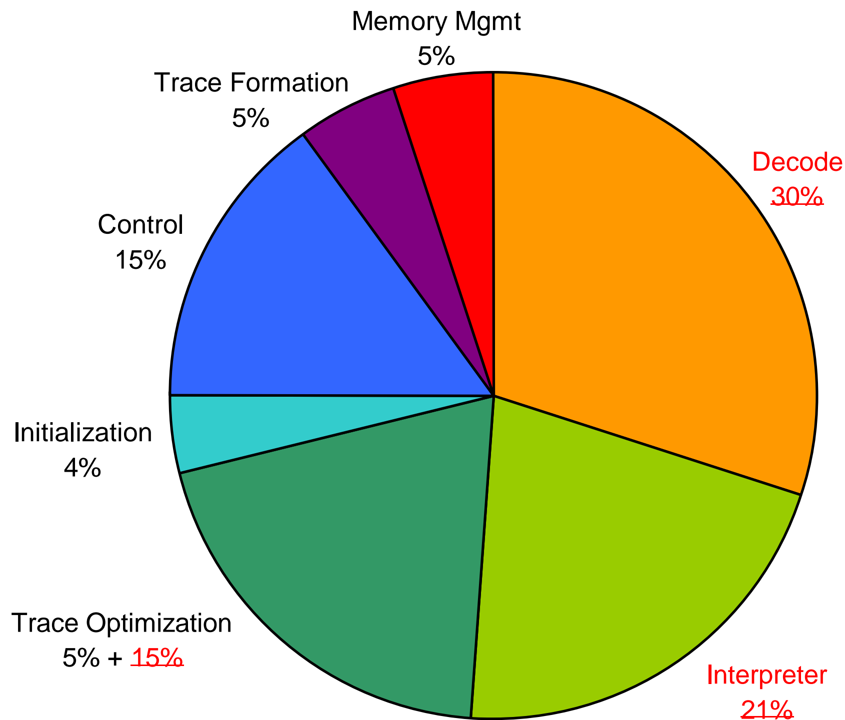


Figure 45: Breakdown of the Dynamo code footprint (underline indicates PA-dependent code)

12.4 Dynamo on highly optimized programs

To test the behavior of Dynamo on highly optimized binaries, we compiled our benchmark programs with optimizations Level 3 (+O3), Level 4 (+O4), and Level 4 with profiling (+O4 +P) optimizations and ran Dynamo on these highly optimized binaries. In addition to all of the Level 2 optimizations, Level 3 performs interprocedural optimizations and inlining within a single source file. Level 4 applies all Level 3 optimizations across all program source files (these are link-time optimizations). Level 4 also optimizes access to global and static data using data layout. Level 4 with profiling invokes profile-based optimization (PBO) which causes the compiler and linker to use profile information to guide optimizations. All programs were profiled and executed using the same input data sets[15]. This is an ideal situation for PBO as it generates 'perfect' profile data.

[15] The HP aCC C++ compiler was used to compile *deltablue*.

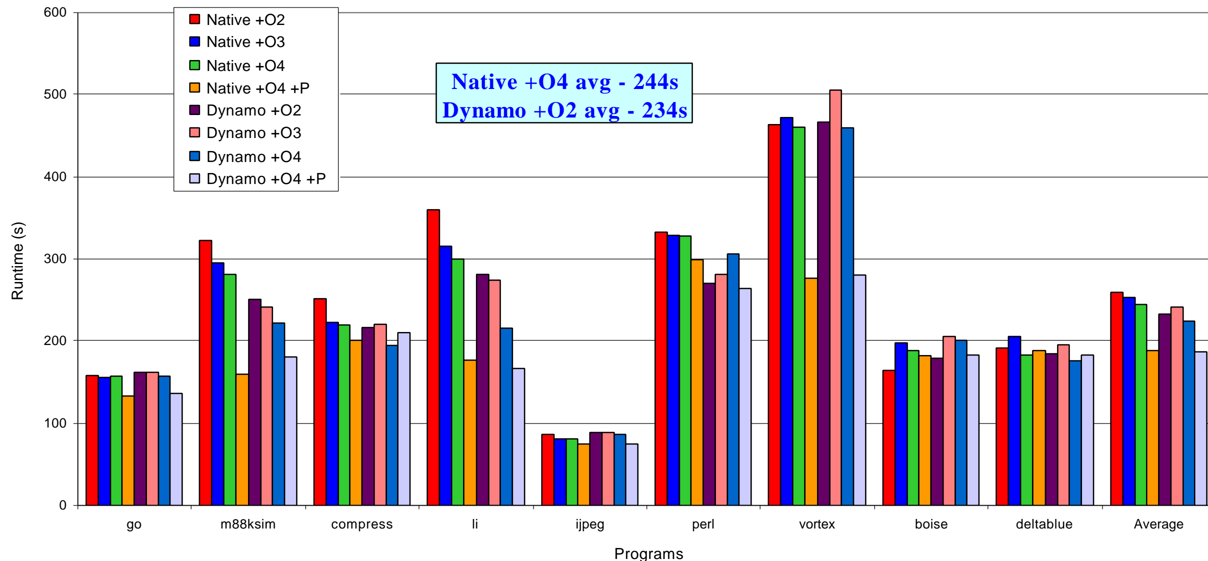


Figure 46: Performance comparison of Dynamo against native execution

Figure 46 shows the corresponding runtimes for all programs across all optimization levels for native execution and execution under Dynamo. Level 3 and Level 4 optimized programs experience significant performance gains with Dynamo, with average speedups of 3.55% and 6%, respectively. Level 4 is the highest non-PBO optimization level available in HP compilers and yields significant speedups over Level 2 as illustrated by the runtimes in Figure 46. Yet Dynamo is able to extract an additional speedup of over 20% on m88ksim and li and a speedup of over 10% on compress. At Level 4 with profiling, the average speedup is a slowdown of 0.2%. m88ksim, compress, and boise have slowdowns, and li, perl and deltablue have speedups. (go, ijpeg, and vortex cause Dynamo to bail out.) At Level 4 with profiling, profile-based inlining has been performed in the native binaries, which eliminates beneficial dynamic inlining opportunities for Dynamo. Among the non-profile-based optimization levels one would except Dynamo's benefits to decrease with higher optimization levels. Clearly program binaries offer significant sources for dynamic optimization even if compiled with Level 3 optimization. Somewhat surprisingly, the average Dynamo speedup for Level 3 optimized binaries is lower than the average speedup for Level 4. Level 3 performs non-aggressive interprocedural optimization including inlining within a single source file, which is likely to reduce some of the benefits of the partial dynamic inlining performed by Dynamo. Level 4 performs link-time optimization leading to a much more aggressive repositioning of the code across source files. The lack of dynamic or profile information in repositioning the code is likely to create new opportunities for Dynamo to improve locality and exploit remaining inlining opportunities in solely the hot areas of the repositioned code.

The data Figure 46 reiterates the fact that even at very high optimization levels, there are opportunities for improvements for Dynamo by observing run-time behavior. Perhaps the most compelling result is the fact that on average running Dynamo on Level 2 optimized programs yields runtimes that are slightly faster than Level 4 optimized programs running without Dynamo.

12.5 Machine specific performance: the PA-8000

This section examines the interactions between Dynamo and the microarchitecture of the PA-8000. The PA-8000 is a four issue, dynamically scheduled processor. The machine uses a 56-entry instruction reorder queue (IRQ) from which it issues instructions. Dynamic branch prediction is used, with 256 3-bit entries in a branch history table (BHT) and a 32 entry branch

target address cache (BTAC) that stores the target addresses of taken branches. The branch misprediction penalty is five cycles. Most indirect branches are not predicted at all and incur the misprediction penalty since the target address is not known until after the execute stage in the pipeline. The exception is procedure returns which are predicted using a call return register termed the Pop Latch, which effectively functions as a return address stack [Hunt 1999]. These traits characterize the PA-8000 as a highly speculative, deeply pipelined processor. Additionally, the processor has a 96 entry unified TLB and a four entry micro I-TLB located adjacent to the processor core. Our system was configured with 1 MB I- and D-caches.

12.5.1 Branch prediction

Dynamo does not perform transformations that directly affect an individual branch's predictability. However, effective trace selection can indirectly aid the PA-8000's branch prediction hardware. Dynamo transforms taken branches in a trace into fall-through branches in the fragment. An implicit goal is that the hot code in the Fragment Cache executes fewer taken branches than the equivalent code in the original binary. Direct branches that "take" but do not have a target address in the BTAC cause a two cycle bubble in the fetch pipeline. Therefore, it is important to utilize the BTAC as effectively as possible. If fewer taken branches are executed due to trace selection, the pressure on the BTAC can be reduced substantially. BTAC entries are less likely to be replaced, and the prediction hardware is more likely to find targets for predicted taken branches. This can reduce the number of times that taken branch penalty is incurred and improve instruction fetch performance. Since branch prediction is central to a dynamically scheduled processor, any improvement in predictor performance can help overall performance.

12.5.2 Indirect branches

Indirect branches can be a significant performance barrier on the PA-8000. Most are not predicted at all and incur the full branch misprediction penalty. Dynamo attempts to eliminate this penalty in two ways. Firstly, as explained in the Trace Selector chapter, Dynamo converts an indirect branch in the trace into a conditional direct branch in the fragment, possibly with a predicted inlined target. The PA-8000 can use its hardware predictor to predict the conditional branch. Secondly, when an indirect branch in a trace is possibly a return that corresponds to a procedure call appearing earlier in the same trace, it might be possible to eliminate the replacement branch (and inline the entire trace, through the call and the return).

12.5.3 Virtual memory

We are not able to compare the VM effects of a program executing under Dynamo and the program executing natively since we have no data for the former. But there are general trends we can speculate about that affect VM performance, particularly with respect to TLB behavior. The Trace Selector and Fragment Cache both perform code layout, the Trace Selector at the code block level and the Fragment Cache at the fragment level. They work in concert to attempt to place the hottest dynamic code regions in the program in close proximity to each other. These hot code regions may have been distributed across many pages in the original program. The Fragment Cache mapping of the hot code regions (manifested as fragments) may use fewer pages and TLB entries than is required by the program executing natively. This can reduce the number of page misses and TLB pressure for instruction accesses. Since the PA-8000 uses a unified I- and D-TLB, a reduction TLB pressure can also reduce TLB misses for data accesses.

12.6 Other programs and processors

We are intimately familiar with the behavior of Dynamo on our benchmark programs running on the PA-8000. In this section, we discuss how the behavior observed on our experimental platform might be extrapolated to other programs and machines.

12.6.1 Program traits

Programs that have well defined working sets or execute in well-defined phases are candidates for performance gains from dynamic optimization. In particular, a well-defined, stable hot spot is especially conducive to gains from trace selection. For example, the dynamic fragment creation rates for m88ksim and perl, shown in Figure 29 (on page 66) and Figure 47 respectively, indicate that these programs have such characteristics. These two programs experience significant speedups under Dynamo. On the other hand, a program that has a constantly changing working set, such as go, is not a good candidate for dynamic optimization. (witness its 153% slowdown under Dynamo).

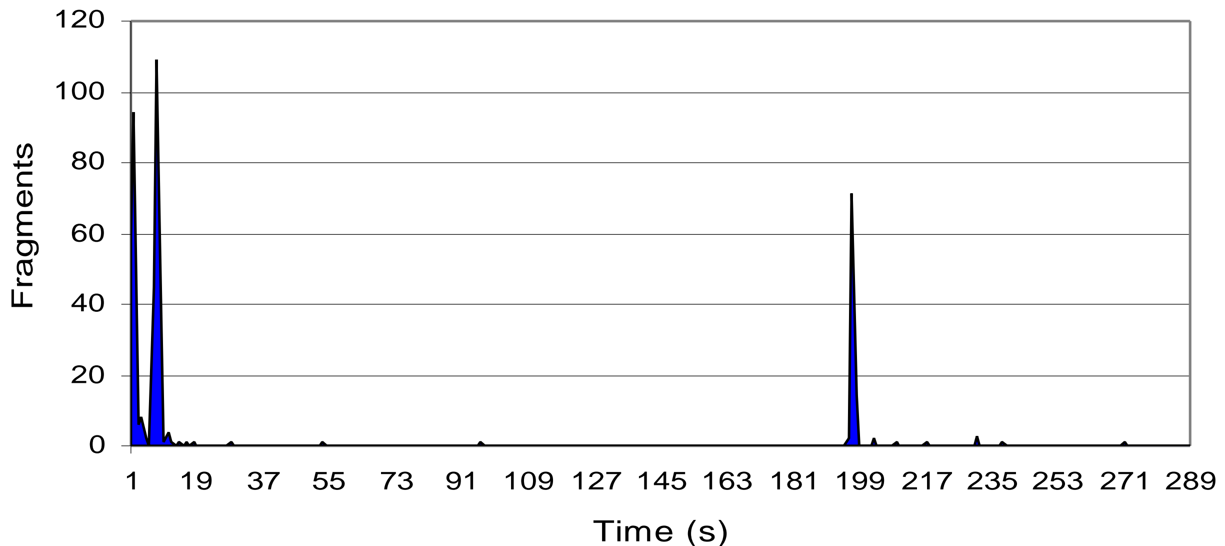


Figure 47: Fragment creation rate pattern for perl

When the Trace Selector is able to pick high-quality traces, the effects of the Fragment Optimizer's actions become more profitable. In particular, when traces are built across many branches and especially across procedure calls and returns, the Fragment Optimizer can dramatically reduce the number of instructions executed on the trace. The Trace Selector can form interprocedural traces only when program execution follows loops that include procedure calls. This highlights that a second indicator of a program that can be successfully optimized dynamically is one that contains loops containing call and returns. Interprocedural trace selection can be helpful without optimization also, since inlining can enhance locality.

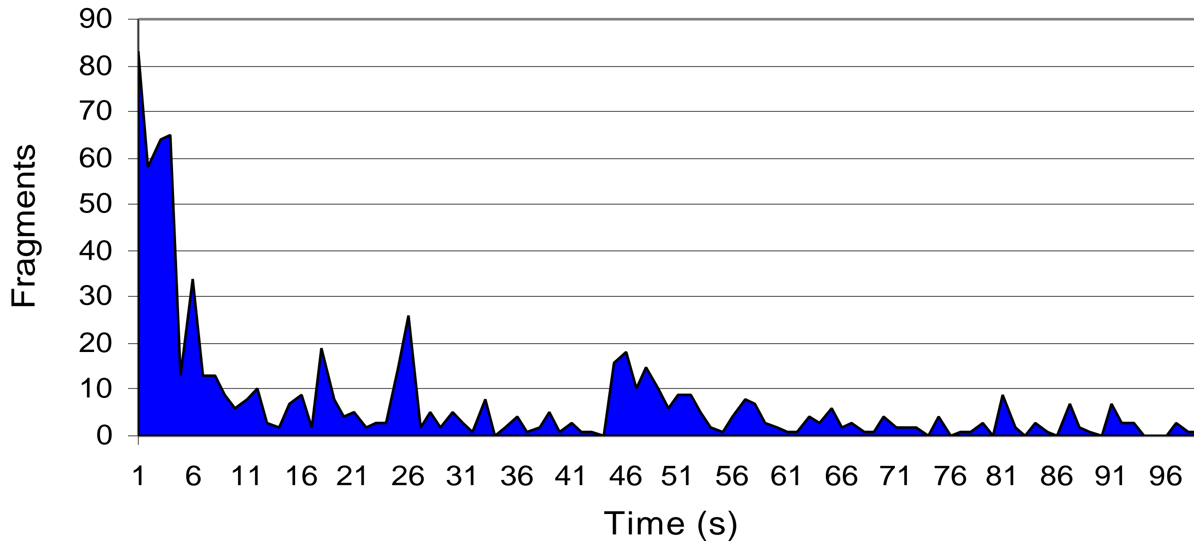


Figure 48: Fragment creation rate pattern for ijpeg

ijpeg is an example of a program that is resistant to dynamic optimization, for multiple reasons. Firstly, the program's working set changes quickly across its execution, as shown in Figure 48; this makes it difficult to capture the working set in the Fragment Cache. Secondly, its execution is dominated by intraprocedural loops, a trait which a static compiler can take advantage of, leaving little opportunity for the Fragment Optimizer. Lastly, ijpeg has a short execution time (<90s for native execution). It is difficult to enable sufficient re-use within the Fragment Cache for such a short running program for Dynamo to show any benefit.

12.6.2 Machine traits

Section 12.5 discussed Dynamo's performance on our PA-8000-based experimental platform. Clearly there are characteristics of the PA-8000 that the Trace Selector and Fragment Optimizer are able to exploit for performance gains. An example is the reduction and elimination of taken-branch and misprediction penalties caused by indirect branches. The question is how would Dynamo perform on processors with other characteristics?

The Trace Selector attempts to reduce or mitigate taken-branch penalties by converting taken branches on a trace into fall through branches. This works well on the PA-8000 since 1) a taken branch can cause a two cycle bubble and 2) the processor contains a small amount of hardware resources to fetch predicted-taken branch targets. However, this may not be as important on a machine that has more resources, e.g., a larger BTAC, to handle predicted-taken branches. The removal of indirect branches is highly beneficial on the PA-8000. This may not be as important on a machine that predicts indirect branches well. However, even a slight improvement in branch predictor improvement performance can be significant on a dynamically scheduled machine, since branch prediction is the backbone of the dynamic scheduling engine.

Branch prediction effects could differ on a statically scheduled machine. For a statically scheduled processor that does not support static speculation, i.e., in-order issue superscalars such as the HP PA-7200 [Gwennap 1994], an improvement in branch predictor performance will aid the instruction fetch engine but by definition cannot improve scheduling within the processor core. For a statically scheduled processor that does support static speculation (such as a VLIW machine [Fisher 1983]), an improvement in branch predictor performance cannot improve scheduling-related performance. For this class of machine, the static compiler performs scheduling, usually based on compile-time heuristics or profile information. The processor core

does not modify the statically generated schedule and so improvements in branch predictor performance cannot improve scheduling. One possibility is for the Fragment Optimizer to perform [re]scheduling in an attempt to improve the static schedule.

One of the desirable effects of trace selection is that the hottest code regions are placed in close proximity in the Fragment Cache memory. This can reduce the number of pages occupied by the hottest code and thus reduce page and TLB misses. This could be important when the combination of machine resources and program behavior causes a high degree of VM pressure. In general, the smaller the TLB, the greater the benefits from the Trace Selector's code re-layout. Since the hot code is stored in a contiguous set of addresses in the Fragment Cache, it is not as widely distributed as within the original binary (and possibly dynamically linked libraries). The contiguous placement means a contiguous mapping into the I-cache, which can reduce the frequency of mapping conflicts. This is particularly useful on machines with low associativity I-caches such as direct-mapped caches.

13 System Development Tools and Utilities

This section discusses two components that are central to developing and analyzing Dynamo. The system profiler, sprof, allows us to observe the behavior of Dynamo components as well as the optimized code dynamically generated by Dynamo. Sprof is composed of three logical portions that work in unison to gather data on Dynamo's behavior. Dynamo's memory allocators are central to the design of Dynamo as a memory-efficient system. Two different memory allocators are used: a zero-overhead allocator based on an algorithm first proposed by Knuth and an enhanced malloc-style allocator.

13.1 The system profiler - sprof

The Dynamo system profiler, sprof, is responsible for reporting data on the internal behavior of all other components of Dynamo, including the Fragment Cache. sprof is composed of three logically distinct pieces, each of which performs a distinct sub-task.

13.1.1 Specifying what to profile: the profiling registry

We wanted Dynamo to be "self-profiling". That is, we wanted each Dynamo component to specify to the profiling system what characteristics should be observed or what actions tracked about that component. Additionally, each component should maintain any counters associated with these actions. To accomplish this end, a *registry service* of profiling functions was developed. The registry works as follows. At Dynamo initialization, any component can request to have a profiling function registered with the registry service. A request is composed of a pointer to the function, an integer value n indicating the number of counter values the function will report, and n strings each describing one of the counters. A registry request is a query to the profiling system asking if it has sufficient internal resources to store the n counters. The registry service responds to each request with an "accept" or "deny" based on its capacity at the time of the request; functions accepted for registration are kept in an internal list.

An acceptance of a registry request means the following things.

- The Dynamo component that initiated the registry (the "parent component") is expected to increment the n counters as it deems appropriate. Each counter probably represents some specific action within the parent component, and it is expected that the parent component will increment the counter when the action occurs. (We say "probably" because the semantic of a specific counter is not pertinent to the registry's function. The registry is oblivious to if and when counters are incremented.)
- The registered function will be invoked at profiler-specified time intervals, at which time the profiling system will issue a query request to the function. The function is expected to respond by returning a pointer to a list of counters maintained by the parent component.
- The profiling system will copy the values in the counters into an internal buffer, which holds a history of counter values. The profiling system will then reset the values of all counters to zero.

The registry system is a flexible, modular interface. It enables a decoupling of determining what data is interesting to report from the mechanics of reporting the data.

13.1.2 Gathering profiling data: timer interrupts

The second of the three sprof components, the interrupt handler (IH), uses the registered functions to gather profiling data. At initialization, Dynamo specifies that a virtual timer interrupt is to occur 100 times per second, and the IH is specified as the signal handler for timer interrupts. This is similar to the functionality provided by the `profil()` routine on HP-UX systems

[Hewlett Packard 1996]. When a timer interrupt (also referred to as a timer hit) occurs, the IH receives control and performs several actions.

- Each function in the profiling registry is invoked. The function is expected to respond by returning a pointer to a list of counters maintained by its parent component.
- The IH copies the values in the counters into a shared memory buffer. The IH then resets each counter to zero.
- The IH inspects the program counter (PC) value it is passed at the time of the interrupt to determine if execution was within a Fragment Cache-resident fragment. If execution was within a fragment, the IH gathers information about the fragment by inspecting various Dynamo data structures and querying various Dynamo components. This information is placed into the shared memory buffer.
- The IH writes the PC value into the shared memory buffer.

The technique of gathering profile data at regular intervals as Dynamo executes is termed *snapshotting*, since at each timer interrupt the profile data provides a snapshot of Dynamo's state at that instant.

In summary, the IH is the signal handler for timer interrupts and gathers the profiling data produced by Dynamo. The profiling data consists of values returned by registered functions, the PC at the time of the interrupt, and if the PC corresponded to a fragment in the Fragment Cache, information about the fragment.

13.1.3 Reporting profiling data: the back-end

The two profiling components described thus far are responsible for allowing Dynamo to specify what profile information is to be gathered and gathering the profile information at regular intervals (100 times/s). The third sprof component, the back-end profiler, is the portion that processes and reports the data. The back-end executes as a separate process and consumes the data placed into the shared memory buffer by the IH. (Semaphores are used to coordinate buffer access.) By default, the back-end produces the following types of data.

- A Fragment Cache hit rate. The back-end receives a stream of PC values from the IH annotated with extra information for those PCs that correspond to Fragment Cache-resident fragments. (During Dynamo execution, control can be in one of three places: the Dynamo control logic, the Fragment Cache, or "other". Other is used to classify execution that is external to Dynamo, such as operating system routines. These three categories can be simplified to two: within the Dynamo fragment cache or external to it.) The Fragment Cache hit rate is simply the ratio of the number of PC values that represent fragments to the total number of PC values. Since the IH is triggered 100 times/s, the number of fragment hits during each second is reported as the hit rate*, i.e.,* 98 hits during 1 second equals a hit rate of 0.98.
- A breakdown of time spent executing in Dynamo control logic. At startup, the back-end reads the symbol table from the Dynamo shared library image and creates a pseudo symbol table that lists the addresses of all Dynamo routines. At the first profiling timer interrupt, the IH passes to the back-end the virtual addresses where the shared library is loaded. The back-end adjusts the offsets in its pseudo symbol table using these addresses. From the stream of PC values provided to the back-end by the IH, every PC that does not represent a fragment is examined to see if it lies within the shared library boundaries. If it does, a counter for the corresponding Dynamo routine is incremented. This provides coarse-grained information about the execution frequency of individual routines. The sum of these counters is used to measure Dynamo overhead.
- A summary of the data reported to the IH by registered functions. Each time the back-end reads the shared memory buffer, counter values from the registered functions are copied into a private buffer maintained by the back-end. The private buffer is sized large enough to hold

all counter values over an entire Dynamo execution. The back-end prints out the average of each individual counter (along with its corresponding descriptive strings) at the end of Dynamo execution.

The back-end also has the ability to report a plethora of optional information.

- A list of the most frequently executed fragments. This is derived from the fragment information that the IH passes to the back-end for PC values that hit in a Fragment Cache-resident fragment. The back-end processes this information at the end of Dynamo execution – or at a limited set of other points, such as just after a Fragment Cache flush – and reports the most frequently executed fragments. The following data (as gathered by the IH) is also reported for each fragment.
  - The size of the fragment.
  - Other fragments that can be reached through conditional branch exits from this fragment.
  - If any of the following is true for this fragment: contains an inlined procedure; is the target of an indirect branch; contains an indirect branch; was reformed after the preceding Fragment Cache flush, if there was a flush.
  - A disassembly of the fragment's instructions.
- A dynamic view of the fragment working set across the entire execution including the size of the working set, the number of Fragment Cache pages within the working set, and the number of fragments that comprise the working set. This is computed using the PC values and fragment data gathered by the IH. The working set size is the sum of the sizes of all fragments seen during the specified time interval (the back-end uses a time interval of 1s for all dynamic statistics). The number of pages is computed from the fragment PCs by converting the virtual PC values into virtual page numbers. The number of fragments is the number of distinct fragment entry points seen during the interval.
- A dynamic view of the Fragment Cache hit rate and all counter values produced by registered functions. As described earlier, the back-end maintains an internal buffer to hold all counter values supplied by the IH. An additional buffer is allocated to store Fragment Cache hit rates.

The back-end does not actually display a graph of the dynamic trend. Rather, it produces a file that lists all of the necessary data in a form that can be imported into a spreadsheet program and then charted.

13.1.4 Examples of profiling data

```
Cache hit rate    91.89%      (9243)
Overhead          8.11%       (815)
Other             0.00%       (0)

Total hits  10058
...
```

Figure 49: Fragment Cache hit rate and breakdown of Dynamo overhead

This section contains examples of the data that is reported by sprof and registered functions. All of the data shown was gathered for a Dynamo execution of the SPECint95 program `ijpeg` on ref inputs without bail out. Most of the data is truncated from the entire output for brevity's sake. Figure 49 shows the summary Fragment Cache hit rate and Dynamo overhead. Detailed output is also available that shows the percentage of total execution time spent in individual Dynamo routines along with the absolute number of times the IH encountered a PC in the routine's address space. We have used this information to determine what routines are most profitable to optimize and tune.

```
User-defined counters
---------------------
                  cache-size (KB) - 295     /s (29499.00)
               fragment_pool (KB) - 109     /s (10873.35)
                 lookup_pool (KB) - 323     /s (32323.95)
                     ir_pool (KB) - 26      /s (2589.06)
                   link_pool (KB) - 141     /s (14127.32)
               dispatch_pool (KB) - 259     /s (25857.58)
              total mempools (KB) - 858     /s (85771.25)
```

Figure 50: Counter values from registered functions

Figure 50 shows counter values gathered by the IH from registered functions and passed to the back-end. For each counter, the back-end prints the descriptive string, the average over all values seen for the counter, and the sum of all values seen. The average can give a quick check of if the value is in an expected range. For additional details the dynamic view data can be dumped and graphed using a spreadsheet package.

```
Fragment profiling
==================
# fragments touched 177   Avg. instr cnt 51   Highest id 2360
%-age cache exec time 99.33

# fragments in 90.0000% threshold 46   Avg. instr cnt 71
      %-age cache exec time 90.21
%-age cache exec time 90.21

Key: ib=has indirect branch, it=ib tgtm ip=has inlined proc

Summary output

                                                          Cond br exits
%-age  # hits  Id                   [orig addr]   i-cnt    Offset, id (%-age)
-----  ------ ------------------------------- -----    ------------------
 9.87   978    94                   [   0x160b8]    40
 8.48   840   109                   [   0x19a50]   126     125, F188    (1.07)
 8.27   820   110                   [   0x19c68]   126     125, F189    (0.14)
 7.88   781    98                   [   0x17290]    28     27, F324     (0.01)
 7.08   702   522                   [   0x2706c]    56     55, F559     (0.01)
 5.46   541   102                   [   0x16f14]   49      48, F304     (0.01)
 5.27   522   115                   [   0x17c48]     9     2, F125      (0.57)
                                                           8, F204      (0.02)
 2.17   215   442                   [   0x206cc]   209     185, F446    (1.08)
                                                           206, F441    (1.98)
                                                           208, F462    (5.03)
 2.13   211   124         (ib)      [   0x17b2c]    38     16, F117     (2.05)
```

Figure 51: Fragment profiling data

Figure 51 shows the data on fragments. The legend at the top of the output gives an overview of the detailed data to follow. The total number of fragments seen and the average instruction count per fragment is displayed. In this output, the top 99.33% of Fragment Cache execution is accounted for by 177 fragments which have an average of 51 instructions. Furthermore, 90.21% of Fragment Cache execution was accounted for by 46 fragments which averaged 71 instructions each. Following these summary statistics is a legend for the fragment-

specific data that follows and the fragment-specific data itself. The most popular fragment is F94 with an entry point 0x160b8. (The "94" signifies that it was the 94th fragment created. This is useful for debugging but is not used for other purposes within Dynamo. "F94" can be considered a *fragment id*.) It accounted for 9.87% of Dynamo execution which corresponds to 978 IH timer hits. F94 has 40 instructions, not including the linker stubs associated with its exit branches. F94 is not linked to any other fragments through conditional branch exits. However, the 7th most-heavily heavly executed fragment, F115, contains two linked conditional branch exits. They are listed in ascending order of their offset in number of intructions from the top of the fragment along with the target fragment id; also, if the target fragment is included in the detailed per-fragment output, the percentage of time it executes is listed in parentheses. For example, the last linked conditional branch exit from F115 is located 8 instructions from the top of the fragment and is linked to F204, which accounts for 0.02% of Dynamo execution. (Information on F204 is not shown in the truncated output.)

13.2 Memory allocators

One of the design goals for Dynamo was memory compactness. One decision that was made early in the design stage was to not use standard memory management routines such as malloc and free. The reason was that the standard implementations could consume a large amount of memory since they do not limit the amount of memory allocated. Additionally, we wanted the ability to quickly free all allocated memory. One option was to code our own versions of malloc and free using configurable, fixed-size memory pools. However, when a request is made to malloc *n* bytes, malloc returns a pointer to *n* bytes of free space but internally has allocated a few additional bytes to maintain a single list of allocated and free blocks. When the objects being allocated are very small, the space overhead of malloc is substantial.

13.2.1 The mempool allocator

We decided to code our own allocator and named it mempool. Mempool is similar to an algorithm that to our knowledge was first introduced by Knuth [Knuth 1973]. Mempool uses a fixed user-specified amount of memory and supports basic allocation and de-allocation functions and fast reset that frees all allocated memory. A first-fit search is used for allocation. Unlike malloc, mempool does not use additional bytes to maintain a list of allocated blocks and is therefore a *zero-overhead allocator*. The disadvantage of not maintaining a single allocated and free block list is that coalescing of free blocks becomes substantially more expensive. Coalescing is important when a memory allocator supports allocation of multiple sized objects (like malloc) so that neighboring free blocks can be combined into a single larger free block. This reduces fragmentation and enables the allocator to satisfy more requests.

The issue for mempool was how to prevent fragmentation given that coalescing was an expensive operation. If the number of different sizes allocated from a pool is restricted, the effects of fragmentation can be reduced. If every allocated object is the same size, an allocation request will be denied only when there is truly no free space remaining in the pool[16]. Therefore, one way to control fragmentation is to use a separate mempool for every different sized object. Since Dynamo allocates dynamic objects of many types and sizes, this is not a practical approach. We opted for a compromise: each major Dynamo component is given its own private mempool. It turns out that most components allocate objects of only one or two different sizes or types. For example, the linker allocates only two different types of objects, link records and fragment lookup table entries. The fewer the number of sizes allocated from a pool, the smaller the probability that fragmentation will cause an allocation request to fail.

[16] This assumes that the pool size is a multiple of the size of the objects being allocated.

Actually, fragmentation can occur only if objects are de-allocated. In general, the only time when Dynamo de-allocates memory is when a fragment is deleted. There are only two conditions under which a fragment is deleted: 1) a Fragment Cache flush or 2) when an individual fragment is deallocated. When the Fragment Cache is flushed, all allocated memory in all mempools can be reclaimed using a fast reset function. For 2-level hierarchical trace selection, fragmentation occurs when static fragments are freed but is minimized by the fact that every pool allocates memory in chunks of only one or two different sizes. For the other trace selection schemes, there is no fragmentation since objects are not deleted. For the latter three schemes, the only time an allocation request cannot be satisfied is when there is not enough memory available.

13.2.2 The poolmalloc allocator

The mempool allocator provides a fixed-size memory pool for zero-overhead allocations. However, for a Dynamo component that allocates and de-allocates objects of many different sizes, mempool may not appropriate due to its potential for fragmentation. The only such Dynamo component is the Fragment Cache when single-fragment replacement is used. A malloc-style allocator is more appropriate for the Fragment Cache, but the standard malloc implementation does not provide a way to limit its memory usage. Therefore, we created poolmalloc, a malloc-style allocator built using mempool. Like mempool, poolmalloc uses a fixed amount of storage and provides a fast reset function. Like malloc, poolmalloc maintains a list of allocated and free blocks. The ability to traverse the unified list is useful when the Fragment Cache has to make a replacement decision due to an out-of-space condition. If the Fragment Cache did not have a map of how fragments are placed in its memory space, it could not make an intelligent decision on how many contiguous fragments need to be displaced to satisfy the outstanding space request. poolmalloc's unified allocated and free block list provides such a map and permits the Fragment Cache to implement a single-fragment replacement policy.

13.2.3 Memory usage in Dynamo

Now that the types of memory allocators used within Dynamo have been presented, we can discuss how the allocators are used. There are five Dynamo components that have exclusive use of their own private memory pools: Dispatch, the Trace Selector, the Fragment Optimizer, the Fragment Linker, and the Fragment Cache. Dispatch allocates the fragment lookup table from the Lookup Pool. The Trace Selector allocates memory from the Fragment Pool to create a compact representation of the code blocks in each trace. The Fragment Optimizer allocates memory from the IR Pool to form the IR for fragment formation and optimization. The Fragment Linker allocates memory from the Link Pool for link records and the fragment lookup table entries, and the Fragment Cache allocates memory from the Cache Pool for fragments. All memory pools except for the Cache Pool use mempool. All memory pools are individually configurable for a specific amount of storage.

We chose to use separate memory pools for the different components because 1) it was simple to engineer and 2) it permitted us to easily measure the memory usage of individual Dynamo components. An example of the need for engineering simplicity is the use of the IR Pool. Memory for the IR is allocated as the IR is constructed by inspecting the code blocks provided to the Fragment Optimizer by the Trace Selector. Once fragment optimization is complete, the IR is no longer needed. Therefore, all memory allocated from the IR Pool can be freed using mempool's reset function. If the IR were allocated from another memory pool, i.e., the Fragment Pool or the Link Pool, the fast reset function could not be used since all memory does not need to be freed, only those blocks used for the IR. Also, individually freeing the potentially non-contiguous blocks could causes fragmentation.

The IR Pool is unique in that objects allocated from it are not persistent, that is, they are required only during fragment optimization. Objects allocated from the other four pools are

persistent; once an object is allocated, it is usually freed in conjunction with the deletion of an associated fragment. However, as alluded to in the Fragment Cache chapter, there are differences in how memory is allocated from the four remaining pools. The fragment lookup table is allocated at Dynamo initialization, and their sizes remain constant during an entire execution. No more objects are allocated from the Lookup Pool for the remainder of Dynamo execution. For this reason, the Lookup Pool is classified as a *static pool*. Since the Link Pool is used for the allocation of link records, its memory usage varies during execution, as does the memory usage of the Fragment Pool and the Cache Pool. These three pools are classified as *dynamic pools*. (The IR Pool is dynamic in that the IR for different fragments could require different amounts of memory. However, since the IR Pool is not persistent storage we do not classify it as dynamic nor as static.)

Since most of the trace selection schemes do not delete any objects and do not cause fragmentation, the amount of memory used within a memory pool accurately captures the amount of memory required for that pool. That is, if 120 KB of memory was allocated in the Link Pool, then the Link Pool needs to be sized at 120 KB to avoid an out-of-space condition. To measure Dynamo's memory efficiency, memory usage is measured since it provides an accurate metric of the amount of memory truly required.

13.2.4 Retrospective

Poolmalloc was implemented specifically for the Fragment Cache, to support allocation and de-allocation of multiple-sized objects in support of single-fragment replacement. However, the replacement policy of choice for Dynamo is to flush the entire cache. Since a cache flush eliminates fragmentation issues, a poolmalloc-style allocator is not necessary. However, the Fragment Cache is implemented using poolmalloc since we determined that the malloc overhead was minimal and did not warrant a re-implementation.

Private fixed-size memory pools are used by each of the Dynamo components. This was done to simplify the engineering of the research infrastructure and makes it easy to measure the memory usage of a specific component. However, for a more efficient, realistic implementation, fewer memory pools would probably be used. For trace selection schemes that do not delete individual fragments use of fewer pools would not cause fragmentation. Use of fewer pools does not impose hard limits on the memory available for specific types of objects as does our current approach.

14 Conclusion

Dynamic optimization offers a new way of thinking about cost-effective performance delivery in computer systems. Today, we divide the performance delivery task between a software compiler and hardware CPU. But several market trends are complicating this traditional method of performance delivery. The increasing use of runtime binding in modern software is one example. Most shrink-wrapped software is shipped as collections of dynamically linked libraries (DLLs), rather than as a monolithic binary. Object-oriented programming languages like C++ and Java that emphasize delayed binding are increasing in popularity. All of this creates obstacles to static compiler optimization, and continues to place the bulk of the performance burden on the processor hardware.

Even if advanced optimization levels can overcome such obstacles, one must still rely on the software vendor to enable them. This places computer system vendors in an awkward position. They have to convince customers that their hardware offers a better cost/performance than the competition, while being at the mercy of the software vendors to make good on this promise. Independent software vendors, for whom time-to-market and robustness of the software are paramount issues, are often reluctant to compile at high optimization levels for a variety of reasons.

Dynamo offers a radically different performance delivery technology: performance optimization is done at the time the bits are executed on the client machine, rather than at the time the bits are created at compile time. The advantage of this approach is that one need not rely on the software vendor to enable the optimization level; it is a client-side performance delivery mechanism. Our prototype has already demonstrated that significant performance wins can be achieved, even on integer benchmarks compiled at high static optimization levels. In addition, delayed binding and dynamic linking actually work in our favor, because these do not present obstacles to a dynamic optimizer that operates after such objects have been bound at runtime. In fact, one can view Dynamo as an enabler of more abstract programming methodologies at the source level, because its runtime optimization effectively negates the adverse performance impact that is often associated with techniques like object oriented design.

Dynamo is also unique in its de-coupling of the translation component from the target-specific optimization component. Dynamo technology only focuses on the native-to-native optimizer half of the picture. Instruction set translation is viewed as being done outside of Dynamo. The rationale is simple. The ideal granularity of translation is often different from the ideal granularity of optimization (for example, a larger translation unit requires lower profiling overhead, but greater optimization time). Doing optimization at translation-time (the approach followed by most Java JITs today) forces one to make a compromise on the granularity, which could end up hurting both the translation and the optimization component. Furthermore, when dynamically compiling a non-native program, the entire program code *has* to be translated, whereas only a few hot spots need to be optimized. Separating the translator from the native optimizer allows each to play a more effective role. For instance, a JIT can be implemented using a lightweight, quick-and-dirty "Dynamo-aware" translator, with the underlying dynamic optimizer doing a more focussed optimization of dynamic hot spots in the native code.

Dynamic optimization should not be viewed as a competing technique to static compiler optimization. Rather, the two can complement one another quite effectively. For instance, the static compiler can assist the dynamic optimizer through hints embedded in the binary. We saw one example of this in the Signal Handler section, where compile-time information could guide the dynamic optimizer in determining whether to do aggressive or conservative optimization. Another example was mentioned in the Fragment Optimizer section, where knowledge of volatile accesses and adherence to calling conventions would allow the dynamic optimizer to perform optimizations that are more aggressive. In return, the dynamic optimizer can provide much richer

feedback to the static compiler than conventional program profiling schemes. For example, it can provide a disassembly of the hottest program traces (that may even go through dynamically linked code). This information can be used by the static compiler to focus its optimizations better, and possibly prime Dynamo's code cache for a later run.

It is also attractive to think of a tighter coupling between the dynamic optimizer software and the microprocessor hardware. The transparent nature of Dynamo's approach makes this vision a practical one to pursue. Hardware support for a dynamic optimizer is also potentially simpler than that for a dynamic translator, because in the case of the latter, the hardware has to support two different machine contexts, a native one and a non-native one. Supporting a translator in hardware also binds the hardware implementation to a particular translator, and hence the translator's input non-native instruction set [Kelly et al. 1998]. On the other hand, supporting a system like Dynamo in hardware allows the same dynamic optimization technology to be applied to a variety of different translators, without changing the hardware. Moreover, keeping the dynamic optimizer in software lets the hardware cost remain low, while providing the flexibility of customizing the optimizer's behavior to specific application domains if necessary.

Another intriguing possibility is to think of dynamic optimization as a service that is transparently provided by the operating system. The operating system controls the loading of a program into memory as well as its actual execution on the underlying machine. The software code cache necessary for Dynamo's operation could be allocated in a special area of system memory. Dynamo code would also have access to privileged machine state as part of the operating system, allowing much of the control logic to be simplified and made more robust.

The Dynamo prototype has shown that it is possible to get significant speedups through dynamic optimization, even from legacy binaries that were created using high levels of static compiler optimization. It has also demonstrated the feasibility of engineering a realistic transparent dynamic optimizer, in spite of the severe constraints of operating at runtime. In doing so, it has broken new ground in the space of performance optimization technologies. Yet, Dynamo may only have opened the floodgates: there is a whole spectrum of open questions that this technology raises, ranging from programming language implementation and hardware design.

15 Acknowledgements

The original Dynamo concept was conceived during the fall of 1995. After some initial pilot studies in the context of the Multiflow VLIW compiler, a memo detailing the project proposal was presented in April 1996 [Bala and Freudenberger 1996]. Since then, a lot of people have influenced our thinking, and helped shape the technology to what it is today. We would especially like to thank Bill Buzbee for numerous discussions on low-level engineering issues in the early stages of the project, Josh Fisher for his constant encouragement even when this seemed like a crazy idea, and Wei Hsu for being our reality check at every turn. We also thank our colleagues Geoffrey Brown, Giuseppe Desoli, Paolo Faraboschi, Stefan Freudenberger, Vineet Soni, Gary Vondran, and Mon-Ping Wang for their comments and suggestions, and Glenn Ammons, our ever reliable summer intern student.

16 References

Ammons, G., Ball, T., and Larus, J.R. 1997. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices 32*, June. 85-96.

Anderson, L., Berc, M., Dean, J., Ghemawat, M.R., Henzinger, S., Leung, S., Sites, L., Vandervoorde, M.T., Waldspurger, C.A., and Weihl, W.E. 1997. Continuous profiling: Where have all the cycles gone. In *Proceedings of the 16th ACM Symposium of Operating Systems Principles*. 1-14.

Auslander, J., Philipose, M., Chambers, C., Eggers, S.J., and Bershad, B.N. 1996. Fast, effective dynamic compilation. In *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI'96*).

Ayers, A., De Jong, S., Peyton, J., and Schooler, R. 1998. Scalable cross-module optimization. In *Proceedings of the SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI'98),* Montreal. 301-312.

Bala, V. and Freudenberger, S. 1996. Dynamic optimization: the DYNAMO project at HP Labs Cambridge (project proposal). *Hewlett Packard Laboratories internal memo, Feb 1996*.

Ball, T. and Larus, J.R. 1993. Branch prediction for free. In Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation. 300-313.

Ball, T. and Larus, J.R. 1996. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29)*, Paris. 46-57.

Banerjia, S., Sathaye, S.W., Menezes, K.N., and Conte, T.M. 1998. MPS: miss-path scheduling for multiple-issue processors. *IEEE Trans. Computers 47*, 12 (Dec.).

Bedichek, R. 1995. Talisman: fast and accurate multicomputer simulation. In *Proceeding 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*.

Bodik, R., Gupta, R., and Soffa, M.L. 1998. Complete removal of redundant expressions. In *Proceedings of the SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI'98),* Montreal. 1-14.

Buzbee, W.B. 1998. System and method of using annotations to optimize dynamically translated code in the presence of signals. *U.S. Patent 5,838,978. Nov 1998*.

Chernoff, A., Herdeg, M., Hookway, R., Reeve, C., Rubin, N., Tye, T., Yadavalli, B., and Yates, J. 1998. FX!32: a profile-directed binary translator. *IEEE Micro, Vol 18, No. 2, March/April 1998.*

Cmelik, R.F. and Keppel, D. 1993. Shade: a fast instruction set simulator for execution profiling. *Technical Report UWCSE-93-06-06, Dept. Comp. Science and Engineering, Univ. Washington.*

Cmelik, R.F. and Keppel, D. 1994. Shade: a fast instruction set simulator for execution profiling. *ACM Sigmetrics Conference on the Measurement and Modeling of Computer Systems*.

Cohn, R., Goodwin, D.W., and Lowney, P.G. 1997. Optimizing Alpha executables on Windows NT with Spike. *Digital Technical Journal 9*, 4.

Cohn, R. and Lowney, P.G. 1996. Hot Cold Optimization of Large Windows/NT Applications. In *Proceedings of the 29th Annual Internation Symposium on Microarchitecture (MICRO-29),* Paris. 80-89.

Consel, C. and Noel, F. 1996. A general approach for run-time specialization and its application to C. In *Proceedings of the 23th Annual Symposium on Principles of Programming Languages.* 145-156.

Conte, T.M., Patel, B.A., Menezes, K.N., and Cox, J.S. 1996. Hardware-based profiling: An effective technique for profile-driven optimization. *International Journal of Parallel Programming 24*, April. 187-206.

Conte, T.M. and Sathaye, S.W. 1995. Dynamic rescheduling: a technique for object code compatibility in VLIW architectures. In *Proceedings of the28th Annual International Symposium on Microarchitecture (MICRO-28).* 208-218.

Cooper, K., Hall M.H., and Kennedy, K. 1993. A methodology for procedure cloning. *Computer Languages 19*, 2 (April). 105-117.

Cramer, T., Friedman, R., Miller, T., Seberger, D., Wilson, R., and Wolczko, M. 1997. Compiling Java Just In Time. *IEEE Micro*, May/Jun 1997.

Duesterwald, E., Gupta, R., and Soffa, M.L. 1993. A practical data flow framework for array reference analysis and its use in optimization. In *Proceedings of the SIGPLAN '93 Conference on Programming Languages Design and Implementation*. 68-77.

Duesterwald, E., Gupta, R., and Soffa, M.L. 1995. Demand-driven computation of interprocedural data flow. In *Proceedings of the 22nd Symposium on Principles of Programming Languages.* 37-48.

Ebcioglu K. and Altman, E.R. 1997. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*. 26-37.

Engler, D.R. 1996. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI'96*).

Fisher, J.A. 1983. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*. 140-150.

Friendly, D.H., Patel, S.J., and Patt., Y.N. 1998. Putting the fill unit to work: dynamnic optimizations for trace cache microprocessors. In *Proceedings of the 31st Annual Internation Symposium on Microarchitecture (MICRO-31),* Dallas. 173-181.

Griswold, D. 1998. The Java HotSpot virtual machine architecture. *Sun Microsystems, Mar. 1998. Available from http://java.sun.com/products/hotspot/whitepaper.html.*

Gwennap, L. 1997. Intel and HP make EPIC disclosure. *Microprocessor Report 11*, 14 (Oct.).

Gwennap, L. 1994. PA-7200 enables inexpensive MP systems. *Microprocessor Report 8*, 3 (Mar.).

Herold, S.A. 1998. Using complete machine simulation to understand computer system behavior. *Ph.D. thesis, Dept. Computer Science, Stanford University*.

Hewlett Packard 1996. HP-UX Release 10.20 Reference Manual. (Jul).

Hohensee, P., Myszewski, M., and Reese, D. 1996. Wabi cpu emulation. In *Proceedings Hot Chips VIII.*

Holzle, U. 1994. Adaptive optimization for SELF: reconciling high performance with exploratory programming. *PhD thesis, Computer Science Dept., Stanford University, available as Technical Report STAN-CS-TR-94-1520. Also available as a Sun Microsystems Lab technical report.*

Hookway, R.J, and Herdeg, M.A. 1997. Digital FX!32: combining emulation and binary translation. *Digital Technical journal, Vol 9, No. 1, 1997, pp 3-12.*

Hunt, Doug 1999. Private communication.

Hsu, W. 1999. Private communication.

Hwu, W.W., Mahlke, S.A., Chen, W.Y., Chang, P. P., Warter, N.J., Bringmann, R.A., Ouellette, R.Q., Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G., and Lavery, D.M. 1993.The superblock: an effective structure for VLIW and superscalar compilation. *The Journal of Supercomputing 7*, (Jan.). 229-248.

Insignia 1991. SoftWindows for Macintosh and UNIX. http://www.insignia.com.

Keller, J. 1996. The 21264: a superscalar Alpha processor with out-of-order execution. Presented at the *9th Ann. Microprocessor Forum*, San Jose, CA.

Kelly, E.K., Cmelik, R.F., and Wing, M.J. 1998. Memory controller for a microprocessor for detecting a failure of speculation on the physical nature of a component being addressed. *U.S. Patent 5,832,205, Nov. 1998.*

Knoop, J., Ruething, O., and Steffen, B. 1994. Optimal code motion: theory and practice. *ACM Transactions on Programming Languages and Systems 16*, 4. 1117-1155.

Knuth, D.E. 1973. The Art of Computer Programming: Volume 1 Fundamental Algorithms, *Reading: MA: Addison-Wesley Publishing Co.*

Kumar, A. 1996. The HP PA-8000 RISC CPU: a high performance out-of-order processor. In *Proceedings Hot Chips VIII*, Palo Alto, CA.

Leone, M. and Dybvig, R.K. 1997. Dynamo: a staged compiler architecture for dynamic program optimization. *Technical Report #490, Dept. Computer Science, Indiana University.*

Leone, M. and Lee, P. 1996. Optimizing ML with run-time code generation. In *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation.* 137-148.

May, C. 1987. Mimic: a fast System/370 simulator. *ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive techniques.*

Morel, E. and Renvoise, C. 1979. Global optimization by suppression of partial redundancies. *CACM 22*, 2. 96-103.

Nair, R. and Hopkins, M.E. 1997. Exploiting instruction level parallelism in processors by caching scheduled groups. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, Denver. 13-25.

Pan, S., So, K., and Rahmeh, J. 1992. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proceedings of ASPLOS-V*. 76-84.

Papworth, D. 1996. Tuning the Pentium Pro microarchitecture. *IEEE Micro*, (Apr.). 8-15.

Peleg, A. and Weiser, U. 1994. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. U.S. patent 5,381,533.

Poletta, M., Engler, D.R., and Kaashoek, M.F. 1997. tcc: a system for fast flexible, and high-level dynamic code generation. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*. 109-121.

Rotenberg, E., Bennett, S., and Smith, J.E. 1996. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29),* Paris. 24-35.

Sannella, M., Maloney, J., Freeman-Benson, B., and Borning, A. 1993. Multi-way versus one-way constraints in user interfaces: experiences with the DeltaBlue algorithm. *Software – Practice and Experience 23*, 5 (May). 529-566.

Sites, R.L., Chernoff, A., Kirk, M.B., Marks, M.P., and Robinson, S.G. Binary Translation. *Digital Technical Journal, Vol 4, No. 4, Special Issue, 1992.*

Song, S.P. Denman, M., and Chang, J. 1995. The PowerPC 604 microproccesor", *IEEE Micro*, (Oct.). 8-17.

SPEC, *Information on SPEC programs are available from the SPEC Web site at http://www.spec.org.*

Stears, P. 1994. Emulating the x86 and DOS/Windows in RISC environments. In *Proceedings Microprocessor Forum*, San Jose, CA.

Wall, D.W. 1986. Global register allocation at link-time. In *Proceedings if the SIGPLAN'86 Symposium on Compiler Construction.* 264-275.

Wall, D.W. 1992. Systems for late code modification. *Research Report 92/3, Digital Equipment Corp. Western Research Laboratory, 250 University Ave, Palo Alto, CA 94301.*

Witchel, E. and Rosenblum R. 1996. Embra: fast and flexible machine simulation. In *Proceedings of the SIGMETRICS '96 Conference on Measurement and Modeling of Computer Systems*. 68-78.

Yeager, K.C. 1996. MIPS R10000 superscalar microprocessor. In *IEEE Micro*, Apr. 1996.

Young, C. and Smith, M.D. 1994. Improving the accuracy of static branch prediction using branch correlation. In *Proceedings of ASPLOS-VI*. 232-241.

Zhang, X., Wang, Z., Gloy, N., Chen, J.B., and Smith, M.D. 1997. System support for automatic profiling and optimization. *In Proceedings of the 16th ACM Symposium on Operating Systems Principles*. 15-26.