Improving Instruction Locality with

Just-In-Time Code Layout

J. Bradley Chen and Bradley D. D. Leupen

Division of Engineering and Applied Sciences

Harvard University

bchen@eecs.harvard.edu

Abstract

This paper describes Just-In-Time code layout (JITCL), a new method for improving the locality of an instruction reference stream by selecting the order of procedures in the text segment during program execution. By determining procedure placement dynamically, this method provides an optimized procedure layout without requiring profile data. For UNIX-style workloads, JITCL provides improvements in instruction cache performance comparable to profile-based layout strategies, while avoiding the requirement of profile data. The late nature of this optimization makes it possible to implement procedure orderings across executable and DLL boundaries, overcoming a limitation of current profile-based techniques. Simulations using Etch [RVL97] on Windows NT show that inter-module JITCL commonly reduces the memory footprint of executable text by 50%.

1. Introduction

Just-In-Time Code Layout (JITCL) is a new method for improving the locality of an instruction reference stream. It achieves similar benefits to profile-based code layout while avoiding the separate profiling step. Current popular methods for procedure layout, such as that described by Pettis and Hansen [PE90], compute an optimized procedure ordering using profile data. Although these schemes can be effective in terms of eliminating instruction cache misses, they share the problems common to profile-based optimization, including profile management and the difficulty in obtaining realistic profiles. JITCL uses a new heuristic which does not require profile information. With this heuristic, an optimized procedure ordering can be computed and applied while the program is running.

An important characteristic of modern Windows applications is their extensive use of dynamically loaded libraries (DLLs). DLLs impose limits on current code-layout optimizations, which cannot typically implement procedure layouts that cross the boundaries of executable files. With JITCL, the optimized procedure ordering is implemented dynamically, with procedures copied into the text segment of the application as it runs. The dynamic nature of JITCL makes it realistic to implement procedure orderings that cross the boundaries created by executable files, introducing the potential for improved instruction cache behavior and smaller working set size for applications that make extensive use of DLLs. This paper describes JITCL and documents its positive impact on instruction cache performance when applied to UNIX and Win32 workloads.

In the next section we describe the JITCL heuristic and compare it to the heuristic described by Pettis and Hansen. We then explain how the JITCL heuristic can be implemented. In Section 3 we present results for JITCL optimization applied within an executable module. Our UNIX results demonstrate that JITCL provides improvements in instruction locality comparable to those commonly achieved with profile-based code layout algorithms for UNIX workloads. Our simulations of JITCL on Windows NT show that it can improve cache performance, avoid bad interactions between modules that occur with the Pettis and Hansen scheme, and substantially reduce text memory requirements.

2. A New Algorithm for Code Layout

2.1 A New Heuristic

In this section we describe the JITCL procedure layout algorithm and how it can be implemented on current hardware. Our algorithm is based on a new heuristic for procedure ordering, which we will present first. The heuristic, which we will call *Activation Order* (*AO*), can be stated simply as:

> ***Heuristic (AO): Co-locate procedures that are activated sequentially.***

With AO, procedures are placed in memory in the order they are first invoked. To get some intuition as to why AO is effective and how it compares to current practice, we compare it to the heuristic used by the popular Pettis and Hansen (*P&H*) procedure layout algorithm:

> *Heuristic (P&H): Co-locate procedures that call each other frequently.*

P&H is applied by generating a weighted call graph, which provides information on the frequency with which procedures invoke each-other, then greedily selecting the heaviest edge from the graph and collapsing the two nodes it connects until the graph is reduced to a single node. The order of procedures in memory is determined by the order in which the nodes are collapsed. Because P&H requires a weighted call graph, it must be computed off-line. In contrast, it is simple to apply AO dynamically, by loading procedures from the executable file into instruction memory in the order in which they are invoked.

Code	P&H	AO
`void main() { Initialize(); for (100 interations){ foo(); bar(); } }`	foo() main() bar() Initialize()	main() Initialize() foo() bar()

Figure 1. A simple example program.

A simple example (Figure 1) illustrates the similarities and the differences in the procedure orderings that result from the two heuristics. For this example, assume that the bulk of the computation for this program occurs in the for loop. The greedy algorithm used by P&H will start by clustering foo() and bar() around main(), as these are the heaviest edges in the weighted call graph. AO orders the procedures according to the order in which they are invoked. Both heuristics provide the desirable property that the four procedures are clumped together in instruction memory, providing better spatial locality. If the instruction cache is too small to hold all four procedures simultaneously, but is large enough to hold three of them, then the P&H order will generate fewer cache misses than AO because it avoids conflicts between main() and foo(). If the instruction cache is too small to hold three procedures simultaneously, but is large enough to hold foo() and bar(), then AO can give better results. This will occur if the overlap in the instruction cache between foo() and bar() for P&H is larger than the overlap for AO between the code in the for loop of main() and the other two procedures. The effectiveness of each heuristic depends on the frequency with which such phenomena occur in real programs. We explore this issue experimentally in Section 3.1.

2.2 Implementing JITCL

In an on-line implementation of the AO heuristic, procedures are copied from the executable file into instructioeuristic, procedures are copied from the executable file into instruction memory the first time they are invoked. Our implementation was designed to meet the following goals:

- *It should require no special hardware support.*
- *It should require minimal changes to the operating system.*
- *It should generate minimal system overhead.*

To implement procedure copying, we replace each procedure call in the original (i.e. conventionally compiled) program with a call to a thunk routine. One thunk is required for each procedure in the program. The task of a thunk is to check if the corresponding procedure has been copied into executable memory (typically from the file-system cache), copying it if it hasn?t, and then to patch the call site so that the next time the procedure will be called directly, rather than going through the thunk. A thunk routine finishes by transferring control to the real implementation of the procedure. Figure 2 gives a schematic version of an executable file that has been loaded for JITCL. In the example, the entry point of a program is __start, which after some initialization calls main().

JITCL introduces several new sources of overhead during program execution. One is overhead from maintaining cache-consistency. Note that programs loaded with JITCL are self-modifying. As a result, care must be taken to insure that the instruction cache contents remain consistent with other caches and main memory. Most current machines (including all PCs) implement hardware cache-consistency between the instruction cache and data cache, so no extra operations are required to maintain consistency. When hardware cache consistency is not provided, the routines that patch the call site and copy procedures into the executable segment must provide for instruction cache consistency. In this case, a user-level routine to flush a cache line or set can be implemented with minimal operating system support [CL97]. Just-in-time compilers, as have been proposed for languages such as Java [GJS96], are likely to encounter similar problems.

```
// This is the entry point of the executable image.
__start:
  perform initializations
  call thunk_main
   . . . // the rest of start

// The first thunk. Thunks reference an array ProcPointers[]
// which has one element for each procedure in the  module.
// Thunks use constants (such as INDEX_main and
// LENGTH_main) which are specific to each thunk.
thunk_main:
  if (ProcPointers[INDEX_main] == NULL) {
    copy main into text segment
    ProcPointers[INDEX_main] =
      new address of main;
  }
  PatchCallSite(RA, ProcPointers[INDEX_main]);
  jmp ProcPointer[INDEX_main];

// thunk code for foo and other procedures.
thunk_foo:
?

// Procedures are copied into the text segment starting here.
InstructionMemory:
?
```

Figure 2: Schematic text segment for a program loaded for JITCL. The executable file includes a text segment as appears above with write+execute attributes. The instructions which implement the actual procedures are found in a separate read-only code segment.

Each procedure that is referenced dynamically is copied exactly once into instruction memory before being invoked. JITCL relies on file-system caching to minimize the number of operations to disk when copying procedures from the executable file into instruction memory. Assuming reasonable pre-fetching and file-system cache performance, the overhead for procedure copying in JITCL should be comparable to overhead that occurs in systems with demand-loaded text.

Copy overhead is related to the number of distinct bytes of code used by a program. In contrast, the overhead from thunk routines is proportional to the number of distinct locations in code and data that reference procedures and are used during a program run. To gain intuition on these overheads, it is useful to group program runs into several classes:

- Trivial programs: For programs that are I/O bound, have small text segments, or that have very short execution times, the impact of any code layout optimization will be small. JITCL will have a small negative impact, and should not be used in this case.

- Large repeating programs: For these programs, the benefit of code-layout will often be substantial, and the overhead of copying and thunk-invocation in JITCL will be amortized over a large number of procedure calls. In this case JITCL is beneficial.

- Large non-repeating programs. Programs with no locality relative to the instruction cache or memory size will have large instruction reference penalties with or without procedure layout optimizations.

In the next section we demonstrate that the overhead of copying procedures and invoking thunk routines is negligible as compared to the benefit of good procedure layout.

2.3 Refinements

Given the basic framework for JITCL, there are a number of refinements that can be made to improve the basic algorithm. When the text segment is loaded, the invocation order for the first few procedures can be determined statically. For example, given the entry point of __start, it may be possible to determine statically that main() will be the next procedure to be called. In this case main() can be copied statically into the program text segment, instead of relying on a thunk for main() to do the copy at runtime. We expect that the impact of this optimization will be negligible in most cases.

Other refinements involve the use of dynamic information collected during one program run to benefit subsequent runs. For example, the order of invocation of procedures during one run could be used to update the order of procedures in the executable file. In this way, the system could reduce the amount of I/O activity required to copy procedures into memory during later runs.

Once an improved procedure layout has been identified, the executable file could be updated to use the improved ordering statically. In our experience we have found the overhead of procedure copying to be negligible; it is not clear that this refinement will be useful.

A final refinement relates to shared libraries. If library boundaries are ignored when applying JITCL, the result is a system that optimizes procedure layout across executable file boundaries. We explore the impact of cross-module JITCL in our Win32 experiments.

3. Methodology

We used two sets of experiments to evaluate the effectiveness of JITCL. In our UNIX experiments we compared JITCL to Pettis and Hansen code layout. We made this comparison using a memory system model that corresponds to a typical RISC-style system, and UNIX-style benchmarks. We chose this platform to be consistent with the context in which earlier code layout optimizations have been evaluated. The UNIX experiments include a detailed evaluation of the interaction between instruction and data caches, and the overhead of procedure copying and thunk-routine invocation. Our Win32 experiments build on the UNIX results by exploring the behavior of JITCL for large Windows applications that use many DLLs.

3.1 UNIX Methodology

We evaluated our UNIX workloads on Digital UNIX and the Digital Alpha microprocessor, using the Atom [SE94] instrumentation system to implement our simulator. We instrE94] instrumentation system to implement our simulator. We instrumented basic blocks, loads and stores in the executable program, inserting calls at these points to our JITCL simulator. Within the simulator, we maintained the state of the caches. We also maintained the additional data structures required by JITCL, and tracked the overhead required to maintain them. Sources of overhead include copy overhead for writing procedures through the data cache, thunk invocation, and data cache traffic from call-site updates.

We implemented a memory system simulation to estimate both instruction and data cache-miss penalties. Our simulated Alpha/UNIX memory system has a split L1 cache with single-cycle latency and no L2 cache. We modeled a 64-bit memory bus operating at 1/3 processor speed, with eleven memory busy cycles to read a 32 byte line. The simulation also used 8K byte coherent direct mapped instruction and data caches, with 32-byte lines. Based on these figures we used a cache read miss penalty of 33 CPU cycles, which corresponds to 11 memory bus cycles. We do not model bus contention or write buffer traffic. JITCL will tend to decrease cache read misses and hence memory traffic. In this respect the results we report are conservative.

Benchmark	Description	Text (KBytes)	Time (sec)
Compress	file compression	112	2
Gcc	The GNU C compiler	1552	60
m88ksim	Simulation of Motorola 88K	160	10
Perl	The perl scripting language	376	104
Raytrace	Image rendering	192	18
Xanim	MPEG player	2024	67

Table 1: Unix workloads. All workloads were statically linked.

Benchmark	Instructions (x1000)	L1 I-Cache Miss Rate	L1 D-Cache Miss Rate
Compress	54786	0.0001	0.0204
Gcc	1384160	0.0580	0.3457
M88ksim	542413	0.0441	0.0118
Perl	2157993	0.0442	0.0275
Raytrace	728778	0.0436	0.0119
Xanim	7085956	0.0204	0.0052

Table 2: Summary statistics for UNIX workloads. Instruction counts are in thousands.

Table 1 shows the experimental workloads we used for our UNIX simulations. Many workloads we originally considered did not have interesting instruction cache behavior and were excluded for that reason. We did include one workload, *compress,* with a small instruction cache miss rate. This gives an indication of the impact of the overhead of JITCL in a case where instruction cache behavior cannot be improved.

Table 2 gives summary statistics for the UNIX workloads. We use several metrics to evaluate the benefit of JITCL: instruction counts, delay cycles, and cache miss rates. Table 2 gives instruction counts for execution of the workloads without optimization. JITCL increases the dynamic instruction count for a given workload, and we use our UNIX workloads to evaluate this overhead in the next section. The benefit of JITCL will be in reducing the instruction cache miss rate, but it also tends to increase the data cache miss rate. For comparison, we provide baseline figures for instruction and data cache misses.

To evaluate JITCL with respect to the best known procedure-layout schemes, we compare results with JITCL layout to results for procedure layout using the Pettis and Hansen algorithm. As JITCL does not use profile information, there is no training input. For the Pettis and Hansen experiments, we trained and tested on the same input. This tends to give a high estimate for the potential benefit of the profile-based optimization; conventional methodology prescribes that different inputs should be used for training te for the potential benefit of the profile-based optimization; conventional methodology prescribes that different inputs should be used for training and testing [FF92]. The results in Section 4 show that JITCL compares favorably with profile-based layout schemes, in spite of this optimistic estimate of the benefit of profile-based layout.

3.2 Win32 Methodology

We evaluated our Win32 workloads using Windows NT 4.0 on an Intel Pentium-Pro based PC. We used the Etch [RVL97] instrumentation and optimization system to implement an instruction cache simulator. For these experiments, we modeled an on-chip 8K byte 2-way set associative instruction cache, backed by a 512K byte direct-mapped off-chip cache. Both caches used 32-byte lines. We choose to simulate an associative first-level cache as most systems that run Win32 applications use an associative first-level cache. Due to resource issues and our prior investigation of JITCL overheads in the UNIX simulations, we did not include data reference activity in these simulations.

Table 3 shows the experimental workloads used for the Win32 experiments. The Win32 applications are large relative to common UNIX applications. Although the size of these applications suggests that code layout optimizations might help them more than the UNIX applications, our analysis will show that other factors, such as cache associativity and interactions between modules, make standard profile-based optimization ineffective for our Windows benchmarks.

Benchmark	Description	Text (Kbytes)
Mazelord	Maze game	1445
Window NT perfmon	Display system performance info.	2805
Lotus Wordpro 96	Document preparation	5148
Microsoft Word 7	Document preparation	7694
Microsoft Internet Explorer 3.02	Web browser	4990

Table 3: Win32 workloads. Text size is the total of code size for the executable and all DLLs used directly by the application. As these workloads are interactive we do not give execution times.

Benchmark	Instructions (x1000)	I-Cache Miss Rate	
		L1	L2
Mazelord	5600	0.0239	0.0015
Window NT perfmon	9000	0.0119	0.0006
Lotus Wordpro 96	170000	0.0361	0.0017
Microsoft Word 7	400000	0.0266	0.0007
Microsoft Internet Explorer 3.02	5000	0.0153	0.0014

Table 4: Summary Statistics for Win32 workloads

Table 4 gives summary statistics for the Win32 workloads. As with the UNIX workloads, the instruction counts in Table 4 do not include overhead instructions introduced by JITCL. The use of the associative rather than direct mapped cache for the Win32 experiments precludes a comparison between the UNIX and Win32 workloads in terms of cache miss rates. Such a comparison is beyond the scope of this paper.

4. Results

4.1 Evaluation of the AO Heuristic

	Miss Rate		Improvement			
Benchmark	P&H	AO	P&H	AO		
Compress	0.00013	NT>	0.00013	0.00019	(0.00003)	(0.00009)
Gcc	0.05611	0.05474	0.00193	0.00330		
m88ksim	0.02172	0.03132	0.02234	0.01274		
Perl	0.03701	0.02829	0.00716	0.01588		
Raytrace	0.01084	0.01212	0.03272	0.03144		
Xanim	0.00292	0.00577	0.01744	0.01459		

Table 5: Instruction Cache miss rates (misses/instruction) for the P&H and AO heuristics. Improvement is computed by subtracting the miss rates in the 2nd and 3rd columns from miss rates in Table 2.

To evaluate the effectiveness of our procedure-layout heuristic, Table 5 compares cache miss behavior for the UNIX workloads and three procedure orderings: the original ordering, an optimized procedure ordering using the Pettis and Hansen algorithm, and procedure ordering using the AO heuristic. Table 5 gives the miss rates for the optimized layouts, as well as the improvement in the miss rate over the original layout.

Table 5 shows that the AO heuristic is effective in improving instruction cache miss rates. AO provides a significant reduction in cache miss rate, on the same order as that for P&H, for all workloads except *compress*. In the case of *compress*, the miss rate is already very low and does not benefit from either AO or P&H. In three of the other five cases, P&H achieves a larger reduction in cache miss rate than AO. These indicate situations where P&H can make effective use of the additional information provided by the profile.

Overall, the results in Table 5 indicate that the AO heuristic can be effective in improving cache miss rates in situations where instruction cache behavior is a significant problem. However, the reduction in cache misses will be beneficial only if they are greater than the overhead of procedure copying and thunk invocation. In the next section we evaluate this overhead.

4.2 Run-time Overhead

Table 6 gives UNIX simulation results for overhead introduced by JITCL. For all the workloads, the number of procedure calls is much higher than the number of call sites (stub calls) or bytes of code copied to support JITCL. Even for *compress*, the overhead of JITCL is amortized over a sufficiently large period of activity that it less than 0.05%.

JITCL also generates additional data cache traffic, as procedures are copied to instruction memory through the data cache, and instruction cache traffic, due to activity from thunk and copy routines. Table 7 quantifies these effects, giving the change in the instruction and data cache miss counts for our experiments. Table 7 also gives the increase (or decrease) in cache miss rate that occurs due to JITCL. In all cases the increase in data cache miss rate is very small (less than 0.001). Table 7 also shows the positive impact that JITCL can have on instruction cache performance.

	Calls (x1000)	Stub Calls	Bytes Copied	Instruction Overhead	Overhead (%)
Compress	923	85	44672	17000	0. 031
Gcc	19647	6644	1091856	1328800	0. 096
m88ksim	7588	240	57744	48000	0. 009
Perl	29472	528	248064	105600	0. 005
Raytrace	0. 005				
Raytrace	89490	354	91920	70800	0. 010
Xanim	15570	1562	353536	313400	0. 060

*Table 6: Overhead of dynamic code movement - Code Movement. Overhead is computed as (100 * overhead instructions / original instruction count).*

Benchmark	Additional I-Cache misses	Additional D-Cache Misses	Increased I-Cache Miss rate	Increased D-Cache Miss rate
Compress	4717	2877	0.0001	0.0001
Gcc	(4484481)	83765	(0.0032)	0.0001
m88ksim	(6907881)	4650	(0.0127)	0.0000
Perl	(34256776)	16899	(0.0159)	0.0000
Raytrace	(22911716)	6319	(0.0314)	0.0000
Xanim	(103351269)	24583	(0.0146)	0.0006

Table 7: Overhead of dynamic code movement - Cache penalties. This table shows the increase (decrease) in cache misses and cache miss rates that occurs with JITCL.

4.3 Overall Impact of JITCL for UNIX workloads

Figure 3 gives results for the overall impact of JITCL for the UNIX workloads. We estimate the benefit of JITCL in terms of cycles saved per instruction, using the cache miss penalties given in Section 3. Figure 3 shows that JITCL has a comparable benefit to profile-based procedure layout schemes such as P&H. JITCL is not beneficial for workloads such as compress where code layout is not a problem, but even in these cases, the overhead of JITCL is small enough to be insignificant.

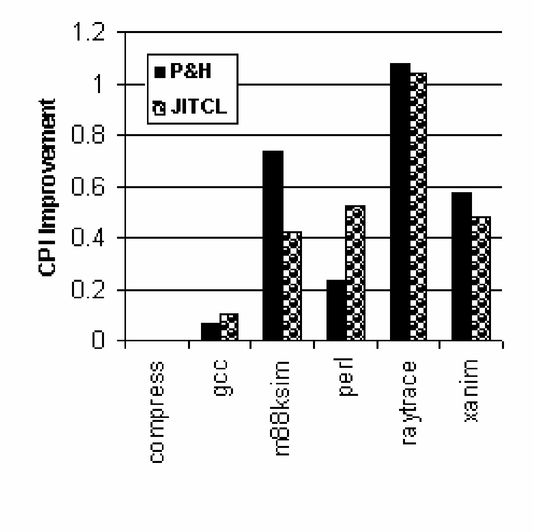

Figure 3. CPI Improvement for UNIX Workloads

4.4 Cache Effects of JITCL for Win32 Workloads

Figures 4 and 5 show instruction cache miss rates for the Win32 Workloads. Due to cache associativity and the large memory requirements of these programs, JITCL does not provide a consistent significant improvement over the default procedure ordering in the first-level cache. The behavior for P&H layout is consistently worse than for the other procedure orderings. The problem with P&H is that it neglects interaction between modules. P&H improves locality within a module by spreading the active procedures in a module evenly over the cache. However, the improved layout within modules also increases competition between modules. The overall result is that P&H procedure layout is not beneficial for the Win32 benchmarks.

For the larger second-level cache, JITCL provides somewhat better cache miss rates than the default cache layout, whereas performance for P&H is consistently worse. Although the effectiveness of JITCL for improving the cache miss behavior of these workloads is limited, the next *section shows that it can provide a substantial benefit in reducing program working set size.*

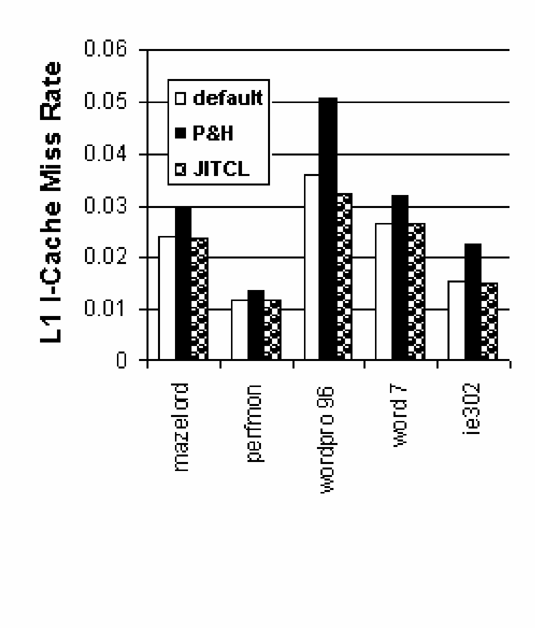

Figure 4. First-level cache miss rates for Win32 Workloads

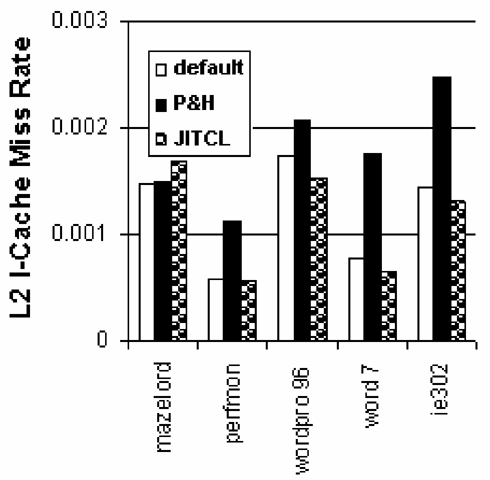

Figure 5. Second-level cache miss rates for Win32 Workloads

4.5 Working Set Size of Win32 Benchmarks

Figure 6 shows memory space occupied by executable program text for the Win32 programs, with and without JITCL procedure layout. JITCL only requires memory for the procedures that are actually used during a run of the application. As a result, JITCL commonly reduces executable memory requirement by about 50%. This can be a substantial benefit for decreasing the memory footprint of an application without sacrificing functionality.

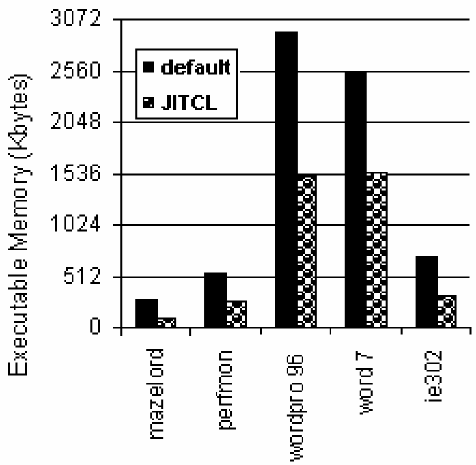

Figure 6: Executable Memory Requirements for Win32 Workloads, in 4k nality.

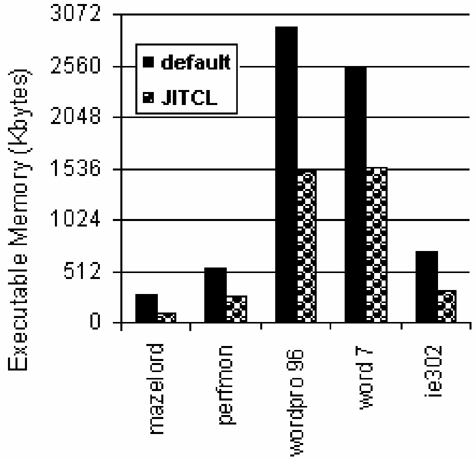

Figure 6: Executable Memory Requirements for Win32 Workloads, in 4k byte pages

Pettis and Hansen layout also has benefits in terms of reducing program memory requirements. However, a comparison of JITCL and P&H for reducing memory requirements is difficult. Given a fixed training/testing input, JITCL and P&H layouts will require a comparable number of memory pages, as both heuristics cluster the procedures used by the program in memory. For multiple inputs, however, the Pettis and Hansen layout will typically give worse performance, as the procedures used during training runs will not exactly match the procedures invoked during an actual use of the program. As a result, JITCL will tend to be more effective a reducing program working set size and improving overall performance.

5. Conclusions

We have described JITCL, a code layout optimization improving instruction cache behavior through dynamic code layout. The optimization is based on a simple heuristic that places procedures in the order in which they are called. This heuristic permits an on-line implementation, avoiding the need for training and profile information. For UNIX workloads, JITCL achieves a comparable benefit to popular profile-based procedure layout schemes without requiring profile information. Although the instruction cache benefits for our Win32 experiments are not significant, the reductions in memory requirements are typically about 50%, making JITCL interesting as a technique for reducing program working-set size. Although JITCL introduces some overhead, our experience indicates that the overhead is negligible compared to the *benefit of improved procedure ordering.*

Acknowledgements

This work was supported by a grant from the National Science Foundation (CCR-9501365). Additional support for this work was provided by Microsoft Corporation and Intel Corporation.

References

[CL97] J. Bradley Chen and Bradley D.D. Leupen. "Improving Instruction Locality with Just-In-Time Code Layout," Technical Report, Division of Engineering and Applied Sciences, Harvard University, March 1997.

[RVL97] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Brian Bershad, Hank Levy, and Brad Chen. "Etch, an Instrumentation and Optimization tool for Win32 Programs." Proceedings of the 1997 USENIX Windows NT Workshop, USENIX Association, Berkeley CA. See also http://www.cs.washington.edu/homes/bershad/Etch/ .

[FF92] J. A. Fisher and S. M. Freudenberger. "Predicting Conditional Branch Directions from Previous Runs of a Program," Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems , ACM, pp. 85-97, October 1992.

[GJS96] J. Gosling, B. Joy, and G. Steele, The Java Language Specification, Addison Wesley, Reading, MA, 1997.

[PH90] K. Pettis and R. Hansen. "Profile Guided Code Positioning," Proceedings of SIGPLAN '90 Conference on Programming Language Design and Implementation, ACM, pp. 16-27, June 1990.

[SE94] A. Srivastava and A. Eustace. "ATOM: A System for Building Customized Program Analysis Tools," Proceedings of the SIGPLAN 1994 Conference on Programming Language Design and Implementation, pages 196-205 , June 1994. See also DEC WRL Research Report 94/2.

This paper was originally published in the Proceedings of the USENIX Windows NT Workshop, Seattle, Washington August 11-13, 1997

Last changed: Aug 1, 1997 prowillen