To be presented at the 22nd International Symposium on Computer Architecture, June 18 - 24, 1995, Santa Margherita Ligure, Italy.

Instruction Fetching: Coping with Code Bloat

Richard Uhlig
Gesellshaft für Mathematik und Datenverarbeitung (GMD)
Schloβ Birlinghoven, 53757 Sankt Augustin, Germany
uhlig@gmd.de

David Nagle
Department of ECE, Carnegie Mellon University
Pittsburgh, PA 15213
dnagle@ece.cmu.edu

Trevor Mudge and Stuart Sechrest
EECS Department, University of Michigan
1301 Beal Ave., Ann Arbor, Michigan 48109-2122
{tnm,sechrest}@eecs.umich.edu

Joel Emer
Digital Equipment Corporation
77 Reed Road HLO2-3/J3, Hudson, MA 01749
emer@vssad.enet.dec.com

Abstract

Previous research has shown that the SPEC benchmarks achieve low miss ratios in relatively small instruction caches. This paper presents evidence that current software-development practices produce applications that exhibit substantially higher instruction-cache miss ratios than do the SPEC benchmarks. To represent these trends, we have assembled a collection of applications, called the Instruction Benchmark Suite (IBS), that provides a better test of instruction-cache performance. We discuss the rationale behind the design of IBS and characterize its behavior relative to the SPEC benchmark suite. Our analysis is based on trace-driven and trap-driven simulations and takes into full account both the application and operating-system components of the workloads.

This paper then reexamines a collection of previously-proposed hardware mechanisms for improving instruction-fetch performance in the context of the IBS workloads. We study the impact of cache organization, transfer bandwidth, prefetching, and pipelined memory systems on machines that rely on the use of relatively small primary instruction caches to facilitate increased clock rates. We find that, although of little use for SPEC, the right combination of these techniques substantially benefits IBS. Even so, under IBS, a stubborn lower bound on the instruction-fetch CPI remains as an obstacle to improving overall processor performance.

Key words*: code bloat, address traces, caches, instruction fetching.*

1 Introduction

It has long been recognized that the best selection of memory-system parameters, such as cache size, associativity and line size, is highly dependent on the workload that a machine is expected to support [Smith85]. Because application and operating system code continually evolves to incorporate new functions, and because memory technologies are constantly changing in capability and cost, it follows that memory-system parameters must be continually re-evaluated to achieve the best possible performance. This paper studies trends in software development that cause programs to grow in size and examines the impact of these trends on one important aspect of memory-system design: the fetching of instructions.

As application and operating system software evolves to include more features and to become more portable, maintainable and reliable, it also tends to consume more memory resources. The "bloating" of code affects a memory-system hierarchy at all levels and in a variety of ways: the larger static sizes of program executables occupy more disk space; the larger working sets of bloated programs require more physical main memory; bloated programs use virtual memory in a more sparse and fragmented manner, making their page-table entries less likely to fit in TLBs; finally, the increased path lengths of bloated code can reduce its locality, making caches less effective in holding code close to the processor for rapid execution.

Improvements in memory technology have offset some of these trends. For example, main-memory DRAMs have quadrupled in size roughly every 2 to 3 years and their price has dropped steadily from about $800 per megabyte in 1986 to a current price of about $40 per megabyte [Touma92]. Magnetic disk drives have exhibited similar improvements in capacity and reduction in cost [Touma92]. However, technology trends have resulted in more complex trade-offs in the case of TLBs and caches. Although continued advancements in integrated-circuit densities make it possible to allocate more die area to on-chip cache structures, reductions in cycle times constrain the maximum size and associativity of *primary* on-chip caches [Jouppi94]. These constraints follow from simple physical arguments that show that increasing cache size and associativity increases access times [Olukutun92, Wada92, Wilton94]. As a result, the primary caches in processors that have targeted fast cycle times (100+ MHz) usually have low associativity and are limited in size to 4-16KB [MReport94, MReport95]. The net effect of these trends is that primary caches have exhibited little growth during the past 10 years [Brunner91]. This results in code bloat having a larger relative impact on the instruction cache than on other parts of the system.

When CPU performance is reported in terms of SPECmarks [SPEC91], the effects of code bloat on system performance in an actual work environment are not revealed. Although the SPEC benchmarks are periodically upgraded (SPEC89, SPEC92 and the

This work was supported by Defense Advanced Research Projects Agency under DARPA/ARO Contract Number DAAL03-90-C-0028, an NSF Graduate Fellowship, and Grant Contract Number 2001 from Digital Equipment Corporation.

Benchmark	Execution Time (%) User	Execution Time (%) OS	Total Memory CPI	Components of Memory CPI: I-cache (CPI_{instr})	D-cache (CPI_{data})	TLB (CPI_{tlb})	CPU (CPI_{write})
SPECint89	97%	3%	0.285	0.067	0.100	0.044	0.074
SPECfp89	98%	2%	0.967	0.100	0.668	0.020	0.179
SPECint92	97%	3%	0.271	0.051	0.084	0.073	0.063
SPECfp92	98%	2%	0.749	0.053	0.436	0.134	0.126

Table 1: Memory System Performance of the SPEC Benchmarks

This table shows the memory-system performance of the SPEC benchmarks as measured by a hardware logic analyzer connected to the CPU pins of a DECstation 3100 running Ultrix. The DECstation 3100 uses a 16.6-MHz R2000 processor and implements split, direct-mapped, 64-KB, off-chip I- and D-caches with 4-byte lines. The miss penalty for both the I- and D-caches is 6 cycles. The R2000 TLB is fully-associative and holds 64 mappings of 4-KB pages.

Performance is reported in terms of cycles per instruction (CPI). Because this is a single-issue machine, the base CPI is 1.0, assuming no pipeline interlocks and a perfect memory system. The actual CPI, as measured by the logic analyzer, is higher primarily because of the memory-system stalls which are summarized under *Components of Memory CPI.*

recent SPEC95), they are done so under a set of constraints that make them less than ideal for exercising certain aspects of system performance. For example, they must be highly portable, they must be relatively easy to run, and the benchmarks themselves must constitute the majority of the code being run. The consequences are that the SPEC benchmarks make little use of OS services (see Table 1), do not include graphical user interfaces (which are non-standard across UNIX, VMS and Windows NT), and are relatively small because they do not link complicated libraries that aren't already included with the benchmarks. Consequently, instruction cache performance is one aspect of the system that is not well tested by the SPEC benchmarks (see also Table 1). In fact, the SPEC benchmarks have evolved to be even less demanding of instruction caches with their second release in 1992. A study of the effects of code bloat on instruction-cache performance must extend beyond SPEC to include a new set of workloads that better represents these trends.

This paper makes three main contributions. First, it describes and analyzes several common software-development practices that lead to growth in application and operating system code. This analysis includes the design of a new collection of workloads called the *Instruction Benchmark Suite* (IBS). Second, this paper re-evaluates several previously-proposed methods for improving instruction-fetching performance in the context of bloated code and new technology constraints. These methods include adding and tuning a *second level* of on-chip cache and then optimizing the interface between the primary and secondary cache by adjusting the transfer *bandwidth*, by *prefetching* instructions, by *bypassing* the cache on a line refill and by *pipelining* the memory system. Our simulations show that the IBS workloads are more sensitive to these optimizations than are the SPEC benchmarks, and that they exhibit larger absolute improvements in performance when these optimizations are applied. Third, our benchmark suite and its corresponding address traces, complete with operating system references, are available to the research community so that our findings can be confirmed and to enable further architectural studies.

In the next section, we examine related work on benchmark characterization and methods for improving instruction-fetching performance. In Section 3, we briefly describe our methodology and analysis tools. Section 4 studies software-development practices that cause programs to grow in size and relates these trends to our design of IBS, while Section 5 evaluates methods for recovering some of the I-cache performance lost to IBS.

2 Related Work

In recent years, much of the architecture research community has settled on using the SPEC benchmark suite as a measure of uniprocessor system performance[1] and considerable effort has been expended by commercial computer manufacturers to tune system performance on these workloads [Gee93]. Despite its popularity for evaluating a wide range of architectural structures, SPEC warns against the use of the SPEC89 or SPEC92 benchmarks for testing memory or I/O performance [SPEC93]. In particular, the SPEC benchmark suite is not a good test of instruction-cache performance, a point made most persuasively by Gee et al., who have shown through exhaustive simulation that most of the SPEC benchmarks fit easily into relatively small I-caches over a range of associativities and line sizes [Gee93].

One reason that the SPEC benchmarks exhibit such good I-cache performance is due to their infrequent invocation of operating system services. Memory-system studies that use workloads with a greater reliance on operating system services have found that much larger caches and TLBs are often required to attain satisfactory performance [Clark83, Emer84, Clark85, Clark88, Smith85, Alexander85, Alexander86, Agarwal88, Borg90, Mogul91, Torrellas92, Flanagan93, Chen93, Chen94, Huck93, Cvetanovic94, Maynard94, Nagle93, Nagle94].

Several hardware-based methods have been proposed to reduce the penalty of misses in small, direct-mapped primary I-caches. The most straightforward is to add a second level of cache, either on or off chip, to reduce time-consuming references to main memory [Short88, Baer87, Baer88, Przybylski89, Przybylski90, Happel92, Kessler91, Olukotun91, Jouppi94, Wang89]. Other methods focus on optimizing the interface from the primary I-cache to the next level in the memory hierarchy, whether it be a second-level cache, or main memory. These methods include the tuning of cache line sizes and *bandwidth* [Przybylski90], *prefetching* [Farrens89, Hill87, Smith78, Smith92, Pierce95], *pipelining* [Jouppi90, Olukotun92, Palcharla94] and *bypassing* [Hennessy90].

There are also software-based methods for improving I-cache performance. Compilers can reduce conflict misses by carefully placing procedures in memory with the assistance of execution-profile information and through call-graph analysis [Hwu89, McFarling89, Torrellas95]. When a cache is physically-indexed

1. During the past three ISCAs, over two thirds of the papers dealing with uniprocessor architecture issues used the SPEC benchmarks.

Workload	Description
`mpeg_play`	mpeg_play (version 2.0) from the Berkeley Plateau Research Group. Displays 85 frames from a compressed video file [Patel92].
`jpeg_play`	The xloadimage (version 3.0) program written by Jim Frost. Displays two JPEG images.
`gs`	Ghostscript (version 2.4.1) distributed by the Free Software Foundation. Renders and displays a single postscript page with text and graphics in an X window.
`verilog`	Verilog-XL (version 1.6b) simulating the logic design of an experimental microprocessor.
`gcc`	The GNU C compiler (version 2.6)
`sdet`	A multiprocess, system performance benchmark which includes programs that test CPU performance, OS performance and I/O performance. From the SPEC SDM benchmark suite.
`nroff`	Unix text formatting program shipped with Ultrix 3.1.
`groff`	GNU C++ implementation of the Unix nroff text formatting program. Version 1.09.
OS	**Description**
Ultrix	Version 3.1 from Digital Equipment Corporation.
Mach	CMU's version mk77 of the Mach 3.0 kernel and version uk38 of the 4.3 BSD UNIX server.

Table 2: The IBS Workloads

All benchmarks were compiled with the Ultrix MIPS C compiler version 2.1, using the -O2 optimization flag.

and larger than the page size, operating systems can implement page-allocation algorithms that more evenly distribute pages in the cache to help prevent conflict misses [Bray90, Kessler92, Bershad94].

Most previous studies of workloads with a significant operating system component have tended to consider simple memory systems. Most of the effort in these studies went into the collection of complete address traces that include multi-task and operating system references. Unfortunately, the resulting address traces are typically not publicly available and require considerable time and resources to recollect. It is therefore difficult to reproduce the findings of these studies or to investigate the performance of the workloads they consider on more sophisticated memory system designs. For this reason, most of the studies of more highly-optimized memory systems tend to use easily-traceable, single-task workloads (like those from SPEC) that do not stress instruction-fetching hardware in a significant way.

This work re-evaluates a collection of aggressive hardware-based instruction fetching optimizations on a more challenging workload than was used in earlier analysis. This workload is designed to represent current trends in software development. We do not consider the aforementioned software-based methods, nor do we consider compiler optimizations that can affect code bloat for better or sometimes for worse (e.g., trace scheduling, loop unrolling, procedure inlining).

3 Methodology

All experiments were run on MIPS-based DECstations under Ultrix 3.1 and Mach 3.0. Table 2 summarizes the benchmarks and operating systems in the IBS workload suite. The IBS workloads are mainly programs that we actually use in our day-to-day work

Benchmark	Execution Time (%) User	Execution Time (%) OS	Components of CPI: I-cache (CPI_{instr})	D-cache (CPI_{data})	Write (CPI_{write})
IBS (Mach 3.0)	62%	38%	0.36	0.28	0.16
IBS (Ultrix 3.1)	76%	24%	0.19	0.30	0.11
SPECint92	97%	3%	0.05	0.08	0.06
SPECfp92	98%	2%	0.05	0.44	0.13

Table 3: Memory Performance of the IBS Workloads

This table shows the memory-system performance of the IBS benchmarks as measured by a hardware logic analyzer connected to a DECstation 3100 (the measurements were made in the same manner as described in Table 1). For the purposes of comparison, the SPEC92 measurements from Table 1 are duplicated here.

and that we feel exhibit poor performance. Our Mosaic WWW browser frequently invokes `mpeg_play`, `jpeg_play` and `gs`, where they limit good interactive performance. The `verilog` workload is a logic simulation of an experimental GaAs processor being developed in our hardware design group. The `gcc` workload is similar to the SPEC workload of the same name, but uses our more recent version of the compiler. We selected one of the workloads from the SPEC SDM suite (`sdet`) to represent our frequent use of typical UNIX commands such as `mkdir`, `mv`, `rm`, `find`, `make`, `diff`, `nroff`, etc. The `groff` workload is the same as `nroff`, but rewritten in C++. Table 3 shows measurements made by a hardware monitor which confirm that the IBS workloads exhibit many more stall cycles due to instruction-cache misses than the SPEC92 benchmarks.

Our analysis of IBS uses two different and complementary methods: *trace-driven* and *trap-driven* simulation. For trace-driven simulation, we gathered address traces, complete with all user and operating system references, by using *Monster*, a hardware logic analyzer connected to the CPU pins of a DECstation 3100 [Nagle92]. Because the caches on this machine are implemented off chip, all memory references were captured using this technique. Long, continuous traces were obtained by stalling the DECstation while unloading the trace buffer in the logic analyzer whenever it became full. A total of 100 MB of references were collected from each workload. Although stalling the processor when the trace buffer becomes full leads to some trace distortion, we found the resulting simulation error to be small. As a check, simulation results using these traces were compared with measurements made by a non-invasive (i.e., non-stalling) hardware monitor and the two agreed within a 5% margin of error. To add an additional degree of confidence to our measurements and to take into account inherent variations in performance due to operating system effects, we use a trap-driven simulator called *Tapeworm II* [Uhlig94, Uhlig95]. Tapeworm simulates cache performance while running alongside the system in the OS kernel, enabling us to conduct multiple experimental trials for each workload and cache configuration.

We adopt a simple performance model based on cycles-per-instruction (CPI) that focuses on instruction-fetching performance [Emer84, Hennessey90, Smith92]:

$$CPI = CPI_{instr} + CPI_{other}$$

where CPI_{instr} is the performance lost to instruction-cache misses and CPI_{other} is determined by the instruction-issue rate and all other sources of processor stalls, such D-cache misses, TLB misses, CPU pipeline interlocks and issue constraints. The I-cache component, CPI_{instr} can be further factored into:

$$CPI_{instr} = MPI \cdot CPM$$

where *MPI* is the I-cache miss ratio (misses per instruction) and *CPM* is the I-cache miss penalty (cycles per miss).

For multi-level cache configurations, both the first-level (L1) cache and the second-level (L2) cache contribute to CPI_{instr}. We determined the L1 contribution by simulating an L1 cache backed by a perfect L2 cache (no L2 misses). L2 contribution is determined by simulating an L2 cache backed by main memory.

Some of our comparisons with the SPEC92 benchmarks are based on miss ratios reported by Gee et al. in [Gee93]. Because Gee et al. performed their study on the same machine type (MIPS-based DECstations) and with the same type of compiler used in this study, meaningful comparisons can be made. For the purposes of illustrating certain points, and to extend our analysis, we selected certain programs from SPEC92 to perform our own simulations and measurements. These programs, the integer benchmarks `eqntott`, `espresso` and `gcc`, span the range of SPEC benchmark sizes with respect to I-cache performance. Gee et al. characterize `eqntott` as small, `espresso` as medium and `gcc` as large in size.

4 Analysis of IBS

In this section, we analyze and compare the instruction-fetching requirements of both SPEC92 and IBS. Our analysis includes a discussion of some of the reasons behind software growth and relates these trends to our design of IBS.

4.1 The Instruction-fetching Demands of Bloated Code

To get a clear picture of the overall I-cache requirements of the SPEC92 and IBS suites, we measured the average performance of their workloads in caches ranging in size from 8-KB to 256-KB (see Figure 1). Following the Three-Cs model of cache performance [Hill87], this graph is a stacked-bar chart that breaks the cause of misses into three components: capacity, conflict and compulsory misses.[1] Capacity misses are removed by larger caches and conflict misses are removed by higher degrees of cache associativity. Figure 1 clearly illustrates that the IBS benchmarks benefit much more from larger and more associative I-caches than do the SPEC92 benchmarks. To achieve approximately the same level of performance as the SPEC92 benchmarks in a direct-mapped, 8-KB I-cache, the IBS workloads require a direct-mapped, 64-KB I-cache, or a highly-associative, 32-KB I-cache.

Table 4 gives another view of the I-cache performance of these workloads by summarizing the individual MPI values for each of the IBS workloads when running in an 8-KB I-cache. Note that IBS under Mach 3.0 exhibits an MPI that is 4 times as large as SPEC92. Also note that the same IBS workload suite running under different operating systems exhibits different average MPI values (The MPI under Mach 3.0 is about 35% higher than it is under Ultrix 3.1).

1. I/O and paging activity can cause a significant number of compulsory D-cache misses. However, compulsory misses account for a negligible fraction of all I-cache misses in both the SPEC92 and the IBS workloads because these workload exhibit little paging in their text segments after they become cached in the filesystem disk-block cache. As a result, compulsory misses are not visible on this plot.

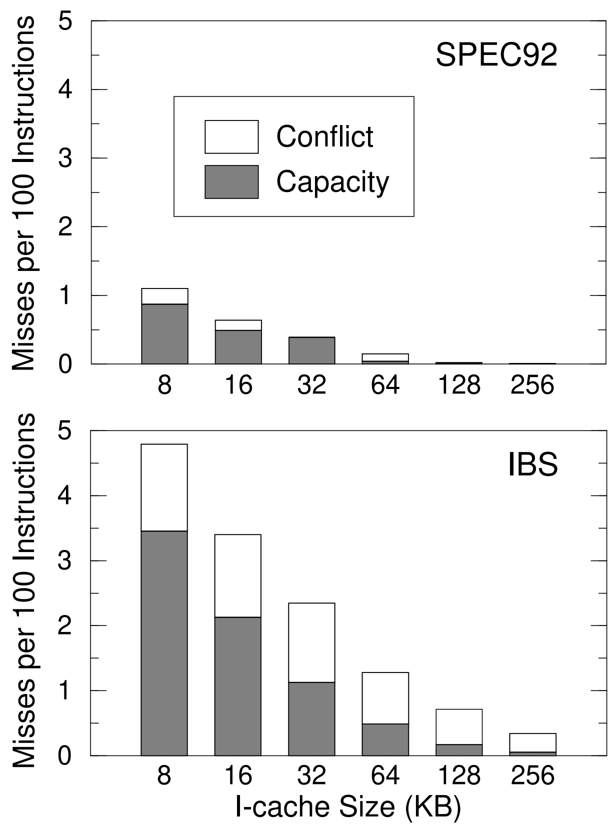

Figure 1: Capacity and Conflict Misses in SPEC92 and IBS

This figure shows I-cache misses per instruction (MPI) for the SPEC92 and IBS workloads. The stacked bars show the relative contribution of capacity and conflict misses to the overall MPI. Capacity misses were approximated by simulating an 8-way, set-associative cache to remove most conflict misses. Conflict misses were found by simulating a direct-mapped cache and counting the number of additional misses compared to the 8-way set-associative simulation. The I-cache line size for all simulations was 32 bytes.

In addition to MPI, Table 4 also gives the percentage of time each workload spends executing in the OS kernel and user-level OS servers. While the SPEC92 benchmarks tend to spend most of their time executing in a single task, the execution of the IBS workloads is spread across multiple address-space domains, including the kernel and the user-level BSD and X servers. Figure 2 illustrates some differences in the structure of the SPEC92 and IBS workloads to help explain the reasons behind their distributions in execution times, and the resulting differences in their I-cache performance. Each of the SPEC92 benchmarks generally consist of a single task that only uses the operating system to load its executable text and to provide some minimal file service for reading inputs. On the other hand, the IBS workloads are composed of many more components, reflecting the increasingly modular nature of modern applications and operating systems. For example, they each link multiple code libraries to gain access to a variety of OS services that are themselves implemented in modular, independent units.

4.2 Reasons for Code Bloat

The benchmarks in IBS were carefully selected to reflect several software-development practices that inevitably lead to growth

Workload			Misses per 100 Instructions (MPI)	Workload Components (% of Execution Time)			
Suite	OS	Application		User	Kernel	BSD	X
IBS	Mach 3.0	`mpeg_play`	4.28	40%	23%	30%	7%
		`jpeg_play`	2.39	67%	13%	17%	3%
		`gs`	5.15	47%	34%	10%	9%
		`verilog`	5.28	75%	14%	11%	0%
		`gcc`	4.69	75%	17%	8%	0%
		`sdet`	6.05	10%	70%	20%	0%
		`nroff`	3.99	80%	5%	15%	0%
		`groff`	6.51	82%	13%	5%	0%
IBS	Mach 3.0	Average	4.79	62%	22%	14%	2%
IBS	Ultrix 3.1	Average	3.52	76%	16%		8%
SPEC92	Ultrix 4.1	Average	1.10	98%	2%		0%

Table 4: Detailed I-cache Performance of the IBS Workloads

This table reports misses per instruction (MPI) for individual IBS workloads when running in an 8-KB, direct-mapped I-cache with a 32-byte line. Detailed MPI values are given for Mach 3.0 only. For the purposes of comparison, the average MPI for the IBS workloads running under Ultrix 3.1 and the SPEC92 benchmarks running under Ultrix 4.1 are also given. The SPEC92 results are based on miss ratios reported by Gee et al. in [Gee93]. Workload components include the user application task(s), the Mach 3.0 kernel, and the BSD and X display servers. The relative importance of each of these *Workload Components* is given as a fraction of total execution time.

in program sizes. These development practices are a consequence of increasing demands on software *functionality*, *portability* and *maintainability* by both application users and developers.

Functionality

To remain competitive, software developers are under constant pressure to add new features and functions to their programs. For example, many commercial applications now support the capability to output non-textual data (graphs, images, video, etc.) in a graphical user interface. Such features are usually implemented with the help of multiple layers of system software that comprise a window system. The dominant window system in UNIX-based workstations is X11 [Scheifler86], which includes an X display server, a window manager and a set of application-linked libraries that implement the core X calls and higher-level graphical objects such as the `tk` widget set [Ousterhout94]. The use of any X application implies that all of these layers of code will be activated, increasing instruction path lengths over workloads with simple textual user interfaces. The IBS workloads represent the overhead of graphics functionality by including the X applications `jpeg_play` and `mpeg_play`, which decode and display compressed still images and moving video, respectively. IBS also includes `gs`, a postscript interpreter that renders full-page layouts, consisting of text and graphics, in an X window.

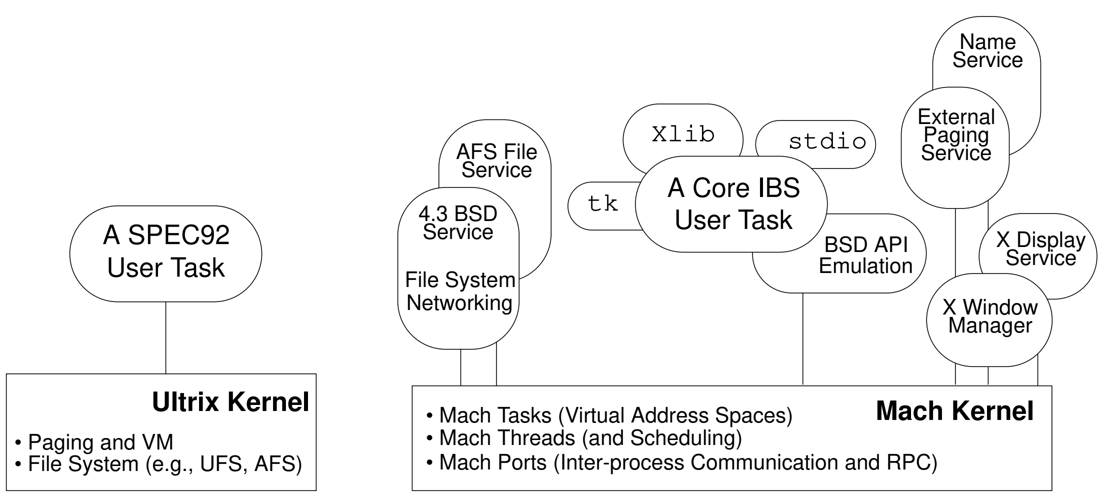

Figure 2: The Components of the SPEC92 and IBS Workloads

Most of the SPEC92 benchmarks consist of a single task that only rely on the operating system to load their executable text and for small file reads. The IBS workloads, however, consist of several modules that communicate through same-task or remote-task procedure calls.

Some applications bloat in size over time because new functions are added to their own core code. As an example of this, IBS includes a recent version of the `gcc` benchmark which exhibits an MPI that is about 15% higher than the older (and smaller) version of `gcc` used in SPEC. IBS also includes the logic simulator `verilog`, which has steadily grown in size with each new release, and which has one of the highest miss ratios among all the applications in IBS.

Portability

To reach the largest possible marketplace, software developers must contend with the problem of making their applications run under several different operating systems and instruction-set architectures. Two different software techniques that increase application portability are *API emulation* and *ABI emulation*.

Porting an application to a different operating system requires that it be rewritten to use the *application-procedure interfaces* or APIs of the new host OS. To simplify this process, some operating systems, including Windows NT [Custer93], Mach 3.0 [Accetta86], and others [Bomberger92, Cheriton84, Malan91, Rozier92, Wiecek92], have been designed to emulate multiple APIs. Overhead due to API emulation is represented in IBS through the use of a 4.3 BSD emulation library that is dynamically linked into the address space of each user application. To isolate this effect, Table 4 also gives the average MPI of IBS running under Ultrix 3.1, a system that does not include the overhead of API emulation. The difference in MPI between the two systems is also due, in part, to other structural differences between Ultrix and Mach (see the next section on maintainability).

By emulating one *application-binary interface* (ABI) in terms of another, some of the difficulties with porting an application to a new instruction-set architecture can be avoided. ABI emulation is sometimes used to ease the transition from an older processor architecture to a newer one. For example, DEC implements ABI emulation by statically translating VAX and MIPS binaries into Alpha binaries [Sites92]. Apple uses a similar strategy to dynamically translate 68040 binaries to the PowerPC architecture [Koch94]. Several other examples of ABI emulators are given in [Cmelik94]. ABI emulation causes code bloat because several host instructions are usually required to emulate a single source instruction. An emulation environment typically also includes a large amount of additional execution state, such as translated instruction blocks or jump tables that lead to frequent indirect jumps [Cmelik94]. We are currently looking for an ABI emulation workload to include in the IBS.

Maintainability

As it grows in size and complexity, application and system software becomes increasingly difficult to maintain. To help manage this complexity, software developers rely on techniques such as object-oriented programming and the restructuring of code into independent and interchangeable modules. For example, the Windows NT Executive bases all of its system abstractions, such as processes, threads and files on an object-oriented model [Custer93]. Windows NT also separates its different API servers (Win32, OS/2, POSIX, etc.) into independent modules or subsystems that are loaded into the system only as needed [Custer93].

The benefits of object-oriented and modular code are well-recognized [Budd91], but because they incur a variety of overheads, these techniques come with a cost. The IBS benchmark suite represents these costs in two ways. First, we run the IBS benchmarks under Mach 3.0, a micro-kernel operating system that uses modularity concepts similar to Windows NT by implementing portions of its code in separate user-level servers. As noted previously, the average MPI of the IBS benchmarks running under Mach is about 35% higher than when they run under the less modular, monolithic-kernel Ultrix. Second, IBS includes the benchmark `groff` which is the `nroff` text-formatting program rewritten in an object-oriented programming language (C++). Notice from Table 4 that the MPI of `groff` is about 60% higher than that of `nroff` when run on the same input. Although IBS currently includes only one C++ program, we believe that `groff` is representative of the poor I-cache performance exhibited by C++ programs in general. This assertion is supported by the recent work of Calder et al. who have performed a more detailed study of 10 C and 10 C++ programs in [Calder94]. Calder et al. report that to achieve equivalent average miss ratios, the C++ programs considered in their study require I-caches that are about four times as large as those required by their C programs.

4.3 Analysis Summary and Comments

The IBS workloads were selected to represent basic pressures on software development that invariably lead to larger programs. As such, they must necessarily include forms of code that make them less portable and harder to use as a benchmark in comparison with SPEC. Nevertheless, most of the workloads in IBS (with the exception of `verilog`) are widely available and can be run on most UNIX-based systems.

Although it could be argued that the programs in IBS could be rewritten to remove their various inefficiencies, they would also lose many of their desirable properties with respect to functionality, portability and maintainability. Therefore, we take these trends as given and now focus on ways to design instruction-fetching hardware to help recover some of the performance lost to bloated code.

5 Instruction Fetch Support for IBS

The IBS workloads require significantly larger I-caches to achieve the same miss rates as the SPEC benchmarks, but cycle-time constraints prevent level-1 (L1) caches from providing the size and/or associativity necessary to deliver good performance [Jouppi94]. However, integration levels have reached a point where small L1 caches can be supported by a variety of on-chip structures that reduce the L1 miss penalty. The remainder of this

Parameters	Configuration	
	Economy	High Performance
Next Level in Hierarchy	Main Memory	Ideal Off-chip Cache
Latency to First Word (Cycles)	30	12
Bandwidth (Bytes/Cycle)	4	8
CPI_{instr} (SPEC)	0.54	0.18
CPI_{instr} (IBS)	1.77	0.72

Table 5: CPI_{instr} for Base System Configurations

Both configurations contain an 8-KB, direct-mapped, on-chip L1 I-cache. In the economy configuration, the L1 I-cache is backed by main memory, while the high-performance configuration is backed by a large, off-chip cache. These latencies and bandwidths were selected by surveying a number of processors in [MReport94]. Latency is the number of cycles until the first word is returned to the cache. For example, a system with a 12-cycle latency and a bandwidth of 8 bytes/cycle requires 12 cycles to return the first 8 bytes and delivers 8 additional bytes in each subsequent cycle. Filling a 32-byte line would require 12+1+1+1 = 15 cycles. For our base configurations, we consider an ideal off-chip cache with zero contribution to CPI_{instr}. Our simulations show that for IBS, a 512-KB, direct-mapped I-cache is close to ideal, contributing only 0.03 to the total CPI_{instr}.

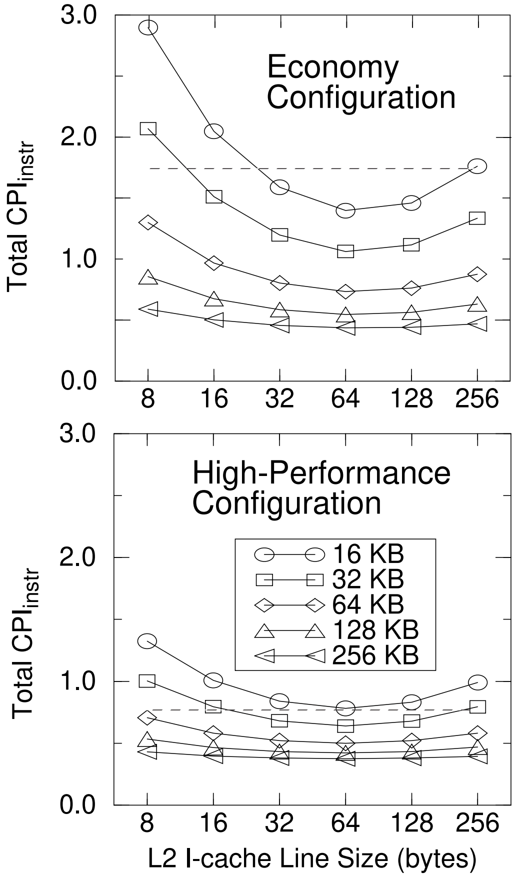

Figure 3: Total CPI_{instr} vs. L2 Line Size

These plots show the CPI_{instr} when an on-chip, direct-mapped L2 cache is added to both baseline configurations. The L2 cache reduces an L1 miss to a 6-cycle latency with a bandwidth of 16 bytes per cycle. This reduces the CPI_{instr} of an 8-KB, direct-mapped L1 I-cache to 0.34. Total CPI_{instr} is computed by adding this value to the stalls caused by L2 misses.

The dotted lines represent the CPI_{instr} for the baseline configurations.

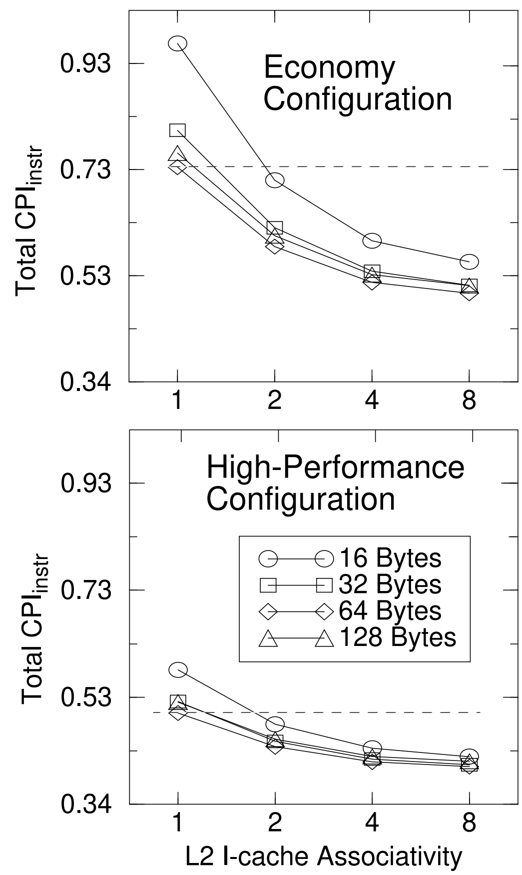

Figure 4: CPI_{instr} vs. L2 Associativity

These plots show the performance benefits of associativity with a 64-KB L2 cache. Notice that the performance of the economy configuration with an 8-way, set-associative cache is nearly equivalent to that of a direct-mapped cache backed by a high-performance memory system.

The dotted line represents the best CPI_{instr} for the direct-mapped, 64-byte line, 64-KB configurations in Figure 3. Notice that the y-axis starts at 0.34, the value of the base L1 CPI_{instr} value.

paper examines the effectiveness of some of these structures when supporting IBS.

Our analysis begins with two baseline configurations outlined in Table 5. The *economy* configuration represents a low-end memory system, while the *high-performance* configuration represents a more-costly, but better-performing memory system that implements an off-chip cache between the on-chip caches and main memory. We extend both configurations by adding an on-chip *second-level* (L2) *cache* and then explore various L2 design tradeoffs. After arriving at an optimized L2 design, we consider how *bandwidth*, *prefetching*, *bypassing* and *pipelining* the L1-L2 interface can further improve performance.

Throughout this section, we draw on the work of numerous researchers who have explored various instruction-fetching techniques, including multi-level caching, prefetching and pipelined-memory systems [Farrens89, Hill87, Kessler91, Jouppi90, Jouppi94, Olukotun92, Przybylski,89, Smith78, Smith82]. This work uses IBS to compare and evaluate these various architectural mechanisms under a more challenging workload. Throughout this analysis, we only consider instruction references. This allows us to factor away data-reference effects that might cloud our specific study of instruction fetching behavior. However, because an L2 cache is likely to be shared by both instructions and data, our results represent a lower bound relative to an actual system.

5.1 Configuring Multi-level Caches for IBS

Our first optimization adds a non-pipelined on-chip L2 cache to both baseline configurations. Figure 3 plots the resulting combined L1 and L2 contributions to CPI_{instr} across a range of L2 cache and line sizes. For the economy configuration, even the smallest L2 cache improves performance over the baseline, provided that the line size is tuned. In contrast, the high-performance system requires at least a 32-KB or 64-KB on-chip L2 cache to improve over its baseline. Comparing the two systems, we see that at 64-KB, the economy configuration's performance matches the high-performance baseline configuration. This suggests that a processor with a 64-KB on-chip L2 I-cache and an economy memory system could provide better I-fetch performance than a processor with a high-performance memory system where the L2 cache is implemented off-chip.

Because an L2 cache is not in the critical path, its associativity is not restricted in the same way as our baseline L1 cache.[1]

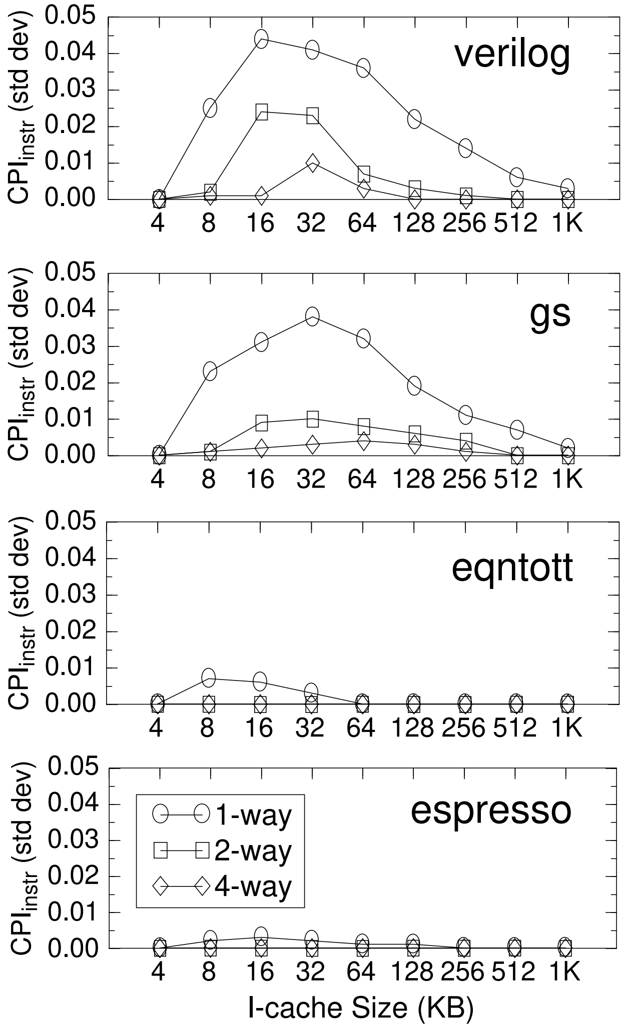

Figure 5: Variability in CPI_{instr} versus I-cache Size and Associativity

These plots show variability in performance of physically-indexed I-caches. Performance varies because the allocation of virtual pages to physical cache page frames is different from run to run of a given workload. Each datapoint above represents 5 experimental trials conducted with the Tapeworm simulator running in an actual system. Variability is reported on the y-axis in terms of one standard deviation of CPI_{instr}.

Figure 4 shows the benefits of L2 cache associativity. Notice that both configurations exhibit the greatest reduction in CPI_{instr} (approximately 25%) between the direct-mapped and 2-way set-associative caches; further increases in associativity (up to 8-way set-associative) only reduce CPI_{instr} another 20%.

Increased associativity improves miss rates by reducing conflict misses. As a result, associativity also reduces variability in performance caused by random OS page-mapping effects in a physically-indexed cache [Kessler91, Sites88]. Variability occurs because different page mappings cause different patterns of conflict misses from run to run of a workload. Figure 5 shows that the amount of variability is a function of the workload, cache size and associativity. Workloads such as `eqntott` and `espresso` (from the SPEC benchmark suite) tend to exhibit little performance variation, but certain workloads from IBS (such as `verilog` and `gs`) are highly variable with certain cache sizes. The plots also show that small amounts of associativity reduce variability by avoiding conflict misses before they happen. This suggests that on-chip, associative L2 caches offer an attractive alternative to the recently-proposed cache miss lookaside (CML) buffers [Bershad94], which detect and remove conflict misses only after they begin to affect performance.

A final advantage of associativity is that it allows designers to more easily add cache memory in increments smaller than a power of two. Recent examples of this include the SuperSPARC, with its 5-way, 20-KB L1 I-cache and the DEC 21164, with its 3-way 96-KB L2 cache [MReport92, MReport94]. This is especially important for on-chip caches because chip size and layout constraints might provide enough area to increase a cache's associativity by 1, but not enough area to double the size of the cache. The ability to change cache sizes in smaller increments also helps to more optimally allocate chip die-area among various on-chip memory-system structures (I-cache, D-cache, TLB) [Nagle94].

5.2 Tuning the L1-L2 Interface

For both configurations, a 64-KB 8-way, set-associative L2 cache contributes less than one third to the total CPI_{instr}, making the 8-KB L1 I-cache the performance bottleneck (see Figure 4). Although the basic structure (size and associativity) of the L1 I-cache is constrained, a number of optimizations to the interface between the L1 and L2 caches is still possible. We now focus on such techniques.

Bandwidth

Figure 6 shows that increasing the bandwidth to the L1 cache significantly improves performance by reducing the L1 cache's fill latency.[1] This relationship between bandwidth and latency suggests that low-bandwidth systems can achieve similar performance improvements by implementing a dual-ported cache. The dual-ported cache allows the processor to continue execution as soon as the missing instruction is returned from memory, hiding fill costs and reducing the effective latency.

Figure 6 also shows that a side-effect of increased bandwidth is an increase in the optimal L1 line size (denoted by the black symbols). This benefits cache design in two ways. First, increasing the line size decreases the size of the cache tags. Second, the reduction in area reduces the cache access time. The Mulder area model predicts a 10% reduction in area when moving from a 16-byte to a 64-byte line (8-KB, direct-mapped cache) [Mulder91], while the Wilton and Jouppi timing model shows a 6% decrease in access time [Wilton94].

The incremental improvements due to increasing bandwidth begin to diminish for rates greater than 16 bytes/cycle. Moreover, building large cache busses (> 128 bits) can consume a significant amount of chip area and possibly impact the overall cache size. This suggests that once the L1-L2 interface reaches a bandwidth of 16 or 32 bytes/cycle, other techniques might be better suited to

1. The additional delay due to the associative lookup will increase the access time to the L2 cache, possibly increasing the L1-L2 latency by 1 full cycle. This would increase the L1 contribution to CPI_{instr} from 0.34 to 0.38. It is also possible that the increase would be small enough so as not to impact the latency. Przybylski [Przybylski88] and Wilton [Wilton94] present detailed models that accurately account for these effects.

1. Fill latency is the number of cycles required to fill a line once the system begins writing data into the cache. For example, a system that can deliver 4 bytes/cycle would have fill latency of 4 cycles when filling a 16 byte line.

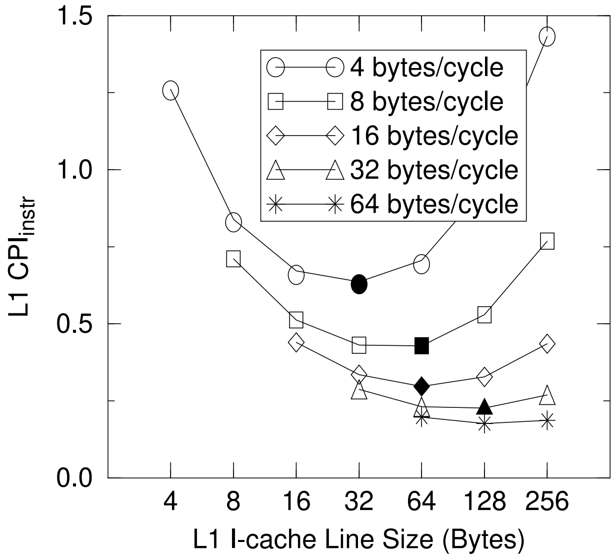

Figure 6: Bandwidth and L1 CPI_{instr} vs. Line Size

This figure shows the L1 contribution to CPI_{instr} for a direct-mapped, 8-KB I-cache backed by an L2 cache with a 6-cycle latency. For these data, the execution model assumes the processor must wait for the entire cache line to refill before it resumes execution.

improving the L1 cache performance. To investigate this, we fixed the L1-L2 interface at 16 bytes/cycle and used this configuration to examine the effects of prefetching, bypassing and pipelining.

Prefetching

One simple prefetch strategy is sequential prefetch-on-miss, where a cache miss is serviced by fetching both the missing line and the next N sequential lines into the cache. Table 6 shows that for small line sizes, prefetching can significantly improve performance. The table also shows a result previously noted by Smith [Smith82]: prefetching over multiple small lines yields better performance than implementing a cache with longer lines. For example, the cache with the 64-byte line has a CPI_{instr} of 0.297, while the cache with the 16-byte line and 3 prefetched lines has a lower CPI_{instr} of 0.260. Both configurations return 64 bytes of instructions, but the system with the longer line size forces it to fetch more potentially useless instructions and to cause more conflict misses. This is particularly true for a miss on the second half of a long cache line because the system must fetch the first half of the line. Our simulations show that when the miss occurs near the end of a line, instructions in the first part of the line are often evicted from the cache before they are referenced. The finer granularity of a 16-byte line overcomes this problem by beginning the fetch closer to the missing word, while allowing the system to prefetch instructions that have a greater potential for being referenced.[1]

Number of Lines Prefetched	Line Size (Bytes) 16	32	64
0	0.439	0.335	0.297
1	0.305	0.271	—
2	0.270	—	—
3	0.260	—	—

Table 6: Prefetching

This table shows the L1 CPI_{instr} (8-KB, direct-mapped) for various line sizes and prefetch lengths. The L1-L2 bandwidth is 16 bytes/cycle and the execution model assumes that the processor must stall until both the miss and the prefetches are returned to the cache. Prefeches are not cancelled.

The cells with an "—", denote data points that are either not reasonable, or that show an increase in CPI_{instr}.

Number of Lines Prefetched	Line Size (Bytes) No Bypass Buffers			Line Size (Bytes) With Bypass Buffers		
	16	32	64	16	32	64
0	0.439	0.335	0.297	—	0.296	0.226
1	0.305	0.271	—	—	0.218	0.224
2	0.270	—	—	0.205	—	—
3	0.260	—	—	0.181	—	—

Table 7: Prefetching + Bypassing

This table compares the performance of configurations with and without bypass buffers. Bypass buffers reduce CPI_{instr} by allowing the processor to continue execution as soon as the missing word returns.

For each system, there are as many bypass buffers as lines returned from the memory system (fetched + prefetched lines). The cells with an "—", denote data points that are either not reasonable or that show an increase in CPI_{instr}.

Bypassing

Sequential prefetch-on-miss can be enhanced by placing the missing line into both the cache and into special bypass buffers. These dual-ported buffers allow the processor to continue execution as soon as the missing word has returned from the L2 cache. Under this scheme, as the cache refills, the processor may only fetch instructions from the bypass buffers. Table 7 shows CPI_{instr} with and without bypassing logic.

To avoid cache pollution due to prefetching, we modified the prefetching+bypassing configuration to only cache prefetched lines if they were used by the processor. For configurations with a small number of prefetches and a small to medium line size, this modification actually reduced performance (not pictured).

Pipelining

The final enhancement that we investigate is pipelining the L1-L2 interface. This allows the L2 cache to accept and fill a request on every cycle with some latency between requests and refills. During cycles where the processor hits in the cache, the memory pipeline is kept busy with sequential prefetch requests.[2] These prefetches are not placed directly into the cache; instead, they are stored in a special memory, called a *stream buffer* [Jouppi90].

We model the stream buffer as a fully-associative, dual-ported memory that can store N prefetched lines (see Table 8) and that can be accessed in parallel with the cache. On a miss in both the I-cache and the stream buffer, a request is sent to memory for the missing line. In the N cycles following the miss request,

1. Our simulations also show that a 64-byte line with 16-byte sub-block allocation can perform almost as well as a 16-byte line with 3 line prefetch. On a cache miss, the system only refills the missing sub-block and all subsequent sub-blocks in the line. While the sub-block configuration had more cache pollution, the decrease in refill cost provided the performance gains.
2. Pipelining the memory system also allows data references to be mixed with instruction prefetch requests.

Number of Lines in Stream Buffer	16 Bytes/Cycle CPI_{instr}	32 Bytes/Cycle CPI_{instr}
0	0.439	0.287
1	0.267	0.186
3	0.184	0.137
6	0.147	0.118
12	0.122	0.103
18	0.114	0.099

Table 8: Pipelined System with a Stream Buffer

The L1 cache line size is set by the bandwidth between the L1 and L2 caches (16 or 32 bytes/cycle). This allows the memory system to accept a request on every cycle.

This execution model assumes that instructions can be moved from the stream buffer to the I-cache without incurring a penalty. Some implementations may incur a 1 cycle penalty during the move if an instruction fetch cannot be serviced by the stream buffers.

N additional sequential lines are also requested from memory. Upon its return from memory, the initial missing line is placed into the I-cache and forwarded to the processor. The prefetched lines are stored in the stream buffer and only move to the I-cache when they are used by the processor. If a miss in both the I-cache and the stream buffer occurs before the control logic can issue all N prefetch requests, prefetching is cancelled and a new miss request is issued. After the miss request is issued, prefetching begins again, starting with the line following the new missing address.

Simulation results show that stream buffers can effectively improve I-fetch performance until the buffer size reaches about 6 lines — reducing CPI_{instr} 66% for the 16 bytes/cycle configuration and 59% for the 32 bytes/cycle configuration. After 6 lines, the improvements become marginal. For a stream buffer with a small number of lines, its effective size can be increased by modifying the prefetch algorithm to prefetch additional instructions when a stream buffer line is moved to the cache.

The data in Table 7 and Table 8 also show that, for a limited number of prefetched lines, prefetch+bypass can perform as well as a pipelined memory system with stream buffers. We believe that this is due to the different policies for caching prefetched instructions: the prefetch+bypass system caches all prefetched lines while the stream buffer system caches only those lines that are used by the processor. Code fragments such as short subroutine calls and short conditional forward branches in loops benefit from the caching-all-prefetched-instructions policy. Further, Pierce has shown that even if the prefetching is on the not-taken path of a branch, these wrong-path prefetched instructions are frequently used soon enough after the prefetch that they benefit from being cached [Pierce95].

This comparison also shows how subtle differences in cache and prefetch policies can influence performance, underscoring the point that there are numerous tradeoffs with respect to bandwidth, latency, line size, prefetching, bypassing and pipelining in the memory hierarchy.

Summary of Optimizations

Figure 7 summarizes the optimizations. For both configurations, adding an associative L2 cache provides the largest performance gains. The improvement is quite dramatic in the case of the economy system. The largest performance improvement in the L1-L2 interface is achieved with the addition of pipelining. In the high-performance system, the L1 CPI_{instr} (0.11 for the 16-byte/cycle configuration) is the dominant factor in total CPI_{instr} (0.18). While this an acceptable level of I-cache performance for a single-issue machine, dual- or quad-issue machines with a minimum CPI of 0.50 and 0.25, respectively, will spend a considerable amount of time stalling on I-cache misses.

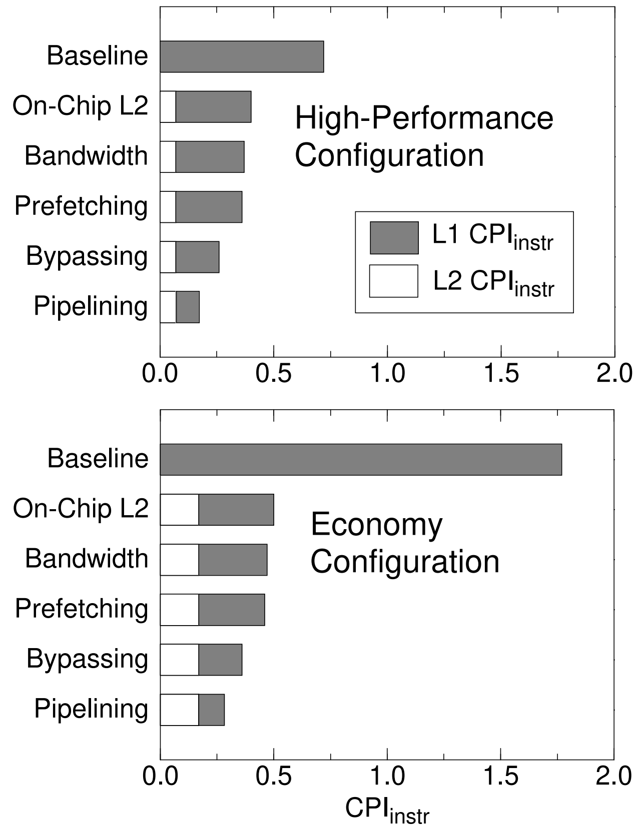

Figure 7: Summary of L1 and L2 Cache Optimizations

This figure shows the cumulative effect of the various optimizations. In both configurations, adding an 8-way, set-associative on-chip L2 cache significantly reduced the CPI_{instr}. For the Economy Configuration, prefetching, bypassing and a pipelined memory system reduced the L1 CPI_{instr} below the L2's CPI_{instr}. However, for the High-Performance Configuration, the L1 cache remains the dominate factor in performance.

Our conclusions would be very different if we had used the SPEC benchmark suite. For example, the optimal on-chip L2 line size for SPEC is (at least) 256 bytes, and associativity decreases CPI_{instr} a mere 0.026. Under SPEC, the optimal L2 cache configuration would have a total CPI_{instr} of only 0.083, before any optimizations to the L1-L2 interface. Some L1 enhancements would also yield significantly different results. For example, the optimal 8-KB L1 line size for a 16-byte/cycle configuration is 128 bytes, which is double the optimal line size for IBS. However, with a CPI_{instr} of only 0.083, there is little motivation to consider the other L1-L2 interface optimizations.

6 Conclusions and Future Work

Relying on the SPEC benchmarks to predict the instruction performance of a proposed memory system design would be unwise, since they are simply unreflective of the complex applications that will run on new machines. We have suggested an alternative set of benchmarks and have described the ways in which they illustrate trends in software leading to relatively poor instruction locality. Using these benchmarks, we have shown how one might design and refine a two-level on-chip cache. This design is quite different than that one might choose based on the SPEC92 benchmarks alone. Simulation results show that this design contributes at least 0.18 cycles to the CPI. This is a considerable reduction from an initial baseline design, but shows that instruction-fetch overhead will be an important component of the execution time of future multi-issue processors that rely on small primary caches to facilitate high clock rates.

This study did not consider more aggressive (non-sequential) prefetching schemes, or the interactions between branch-prediction and instruction-fetching hardware. By making the IBS traces available, we hope to encourage the exploration of these more sophisticated hardware mechanisms on demanding workloads. The IBS traces include both instruction and data memory references, and cover the full activity of all user and kernel processes. To obtain the traces, consult our home page at http://www.eecs.umich.edu/~bassoon.

7 References

[Accetta86] Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A. and Young, M. *Mach: A new kernel foundation for UNIX development*, In the Summer 1986 USENIX Conference.

[Alexander85] Alexander, C. A., Keshlear, W. M. and Briggs, F. *Translation buffer performance in a UNIX environment*. Computer Architecture News **13** (5): 2-14, 1985.

[Alexander86] Alexander, C., Keshlear, W., Cooper, F. and Briggs, F. *Cache memory performance in a UNIX environment*. Computer Architecture News **14**: 14-70, 1986.

[Agarwal88] Agarwal, A., Hennessy, J. and Horowitz, M. *Cache performance of operating system and multiprogramming workloads*. ACM Transactions on Computer Systems **6** (Number 4): 393-431, 1988.

[Baer87] Baer, J.-L. and Wang, W.-H. *Architectural choices for multi-level cache hierarchies*. In the 16th International Conference on Parallel Processing: 258-261, 1987.

[Baer88] Baer, J.-L. and Wang, W.-H. *On the inclusion properties for multi-level cache hierarchies*. In the 15th ISCA, Honolulu, Hawaii, 73-80, 1988.

[Bershad94] Bershad, B., Lee, D., Romer, T. and Chen, B. *Avoiding conflict misses dynamically in large direct-mapped caches*, In the 6th ASPLOS, San Jose, CA, 158-170, 1994.

[Bomberger92] Bomberger, A., Hardy, N., Frantz, A. P., Landau, C. R., Frantz, W. S., Shapiro, J. S. and Hardy, A. C. *The KeyKOS Nanokernel Architecture*, In the USENIX Micro-Kernels and Other Kernel Architectures Workshop, Seattle, WA, 95-112, 1992.

[Borg90] Borg, A., Kessler, R. and Wall, D. *Generation and analysis of very long address traces*, In the 17th ISCA, Seattle, WA, 1990.

[Bray90] Bray, B., Lynch, W. and Flynn, M. J. *Page allocation to reduce access time of physical caches*. Stanford University, Computer Systems Laboratory. CSL-TR-90-454. 1990.

[Brunner91] Brunner, R. A. *VAX Architecture Reference Manual*. Digital Press, 1991.

[Budd91] Budd, T. *An Introduction to Object-Oriented Programming*. Addison-Wesley Publishing IBSN 0-201-54709-0, 1991.

[Calder94] Calder, B., Grunwald, D. and Zorn, B. *Quantifying behavioral differences between C and C++ programs*. The Department of Computer Science, University of Colorado. CU-CS-698-94. 1994.

[Chen93] Chen, B. and Bershad, B. *The impact of operating system structure on memory system performance*, In the 14th Symposium on Operating System Principles, 1993.

[Chen94] Chen, B. *Memory behavior of an X11 window system*, In the USENIX Winter 1994 Technical Conference, 1994.

[Cheriton84] Cheriton, D. R. *The V kernel: A software base for distributed systems*. IEEE Software **1** (2): 19-42, 1984.

[Clark83] Clark, D. *Cache performance in the VAX-11/780*. ACM Transactions on Computer Systems **1**: 24-37, 1983.

[Clark85] Clark, D. W. and Emer, J. S. *Performance of the VAX-11/780 translation buffer: Simulation and measurement*. ACM Transactions on Computer Systems **3** (1): 31-62, 1985.

[Clark88] Clark, D. W., Bannon, P. J. and Keller, J. B. *Measuring VAX 8800 Performance with a Histogram Hardware Monitor*, In the 15th ISCA, Honolulu, Hawaii, 176-185, 1988.

[Cmelik94] Cmelik, B. and Keppel, D. *Shade: A fast instruction-set simulator for execution profiling*, In SIGMETRICS, Nashville, TN, ACM, 128-137, 1994.

[Custer93] Custer, H. *Inside Windows NT*. Redmond, WA, Microsoft Press, 1993.

[Cvetanovic94] Cvetanovic, Z. and Bhandarkar, D. *Characterization of Alpha AXP performance using TP and SPEC Workloads*, In the 21st ISCA, Chicago, Ill., 1994.

[Emer84] Emer, J. and Clark, D. *A characterization of processor performance in the VAX-11/780*, In the 11th ISCA, Ann Arbor, MI, 301-309, 1984.

[Farrens89] Farrens, M. and Pleszkun, A. *Improving performance of small on-chip instruction caches*, In the 16th ISCA, 234-241, 1989.

[Flanagan93] Flanagan, J. K., Nelson, B. E. and Archibald, J. K. *The inaccuracy of trace-driven simulation using incomplete trace data*. Brigham Young University. 1993.

[Gee93] Gee, J., Hill, M., Pnevmatikatos, D. and Smith, A. J. *Cache Performance of the SPEC92 Benchmark Suite*. IEEE Micro (August): 17-27, 1993.

[Happel92] Happel, L. P. and Jayasumana, A. P. *Performance of a RISC machine with two-level caches*. IEE Proceedings-E **139** (3): 221-229, 1992.

[Hennessy90] Hennessy, J. L. and Patterson, D. A. *Computer Architecture A Quantitative Approach*. San Mateo, Morgan Kaufmann, 1990.

[Hill87] Hill, M. *Aspects of cache memory and instruction buffer performance*. The University of California at Berkeley. 1987.

[Huck93] Huck, J. and Hays, J. *Architectural support for translation table management in large address space machines*, In the 20th ISCA, San Diego, CA, 39-50, 1993.

[Hwu89] Hwu, W.-m. and Chang, P. *Achieving high instruction cache performance with an optimizing compiler*, In the 16th ISCA, Jerusalem, Isreal, 242-251, 1989.

[Jouppi90] Jouppi, N. *Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers*, In the 17th ISCA, Seattle, WA, 364-373, 1990.

[Jouppi94] Jouppi, N. and Wilton, S. *Tradeoffs in two-level on-chip caching*, In the 21st ISCA, Chicago, IL, 34-45, 1994.

[Koch94] Koch, P. *Emulating the 68040 in the PowerPC Macintosh*, In Microprocessor Forum, San Francisco, CA, 1994.

[Kessler91] Kessler, R. *Analysis of multi-megabyte secondary CPU cache memories*. University of Wisconsin-Madison. 1991.

[Kessler92] Kessler, R. and Hill, M. *Page placement algorithms for large real-indexed caches*. ACM Transactions on Computer Systems **10** (4): 338-359, 1992.

[Malan91] Malan, G., Rashid, R., Golub, D. and Baron, R. *DOS as a Mach 3.0 application*, In the USENIX Mach Symposium, 27-40, 1991.

[Maynard94] Maynard, A. M., Donnelly, C. and Olszewski, B. *Contrasting characteristics and cache performance of technical and multi-user commercial workloads*, In the 6th ASPLOS, San Jose, CA, 145-156, 1994.

[McFarling89] McFarling, S. *Program optimization for instruction caches*, In the 3rd ASPLOS, Boston, MA, 183-191, 1989.

[Mogul91] Mogul, J. C. and Borg, A. *The effect of context switches on cache performance*, In the 4th ASPLOS, Santa Clara, CA, 75-84, 1991.

[MReport92-95] Microprocessor Report. Sebastopol, CA, MicroDesign Resources, 1992, 1993, 1994 and 1995.

[Mulder91] Mulder, J., Quach, N. and Flynn, M. *An area model for on-chip memories and its application*. IEEE Journal of Solid-State Circuits **26** (2): 98-106, 1991.

[Nagle92] Nagle, D., Uhlig, R., Mudge, T., *Monster: a tool for analyzing the interaction between operating systems and architectures*. CSE-TR147-92. University of Michigan, 1992.

[Nagle93] Nagle, D., Uhlig, R., Stanley, T., Sechrest, S., Mudge, T. and Brown, R. *Design tradeoffs for software-managed TLBs*. In the 20th ISCA, San Diego, CA, 27-38, May 1993.

[Nagle94] Nagle, D., Uhlig, R., Mudge, T. and Sechrest, S. *Optimal allocation of on-chip memory for multiple-API operating systems*, In the 21st ISCA, Chicago, IL, May 1994.

[Olukotun91] Olukotun, O. A., Mudge, T. N. and Brown, R. B. *Implementing a cache for a high-performance GaAs microprocessor*, In the 18th ISCA, Toronto, Canada, 138-147, 1991.

[Olukotun92] Olukotun, K., Mudge, T. and Brown, R. *Performance optimization of pipelined primary caches*, In The 19th ISCA, Gold Coast, Australia, 181-190, 1992.

[Ousterhout94] Ousterhout, J. K. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company, 1994.

[Palcharla94] Palcharla, S. and Kessler, R. E. *Evaluating stream buffers as a secondary cache replacement*, In the 21st ISCA, Chicago, IL, 24-33, 1994.

[Patel92] Patel, K., Smith, B. C. and Rowe, L. A. *Performance of a Software MPEG Video Decoder*. University of California, Berkeley. 1992.

[Pierce95] Pierce, J., *Cache Behavior in the Presence of Speculative Execution-The Benefits of Misprediction*, Ph.D. Thesis, The University of Michigan, 1995.

[Przybylski89] Przybylski, S., Horowitz, M. and Hennessy, J. *Characteristics of performance-optimal multi-level cache hierarchies*, In the 16th ISCA, Jerusalem, Israel, 114-121, 1989.

[Przybylski90] Przybylski, S. *The performance impact of block sizes and fetching strategies*, In the 16th ISCA, Seattle, WA, 160-169, 1990.

[Rozier92] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrman, F., Kaise, C., Langlois, S., Leonard, P. and Neuhauser, W. *Overview of the Chorus distributed operating system*, In the Micro-kernels and Other Kernel Architectures Workshop, Seattle, WA, USENIX, 39-69, 1992.

[Scheifler86] Scheifler, R. and Gettys, J. *The X window system*. ACM Transactions on Graphics 5 (2): 79-109, 1986.

[Short88] Short, R. and Levy, H. *A simulation study of two-level caches*, In the 15th ISCA, Honolulu, Hawaii, 81-88, 1988.

[Sites88] Sites, R. L. and Agarwal, A. *Multiprocessor cache analysis with ATUM*, In the 15th ISCA, Honolulu, Hawaii, 186-195, 1988.

[Sites92] Sites, R., Chernoff, A., Kirk, M., Marks, M. and Robinson, S. *Binary translation*. Digital Technical Journal **4** (4): 137-152, 1992.

[Smith78] Smith, A. J. *Sequential program prefetching in memory hierarchies*. IEEE Computer **11** (12): 7-21, 1978.

[Smith82] Smith, A. J. *Cache Memories*. Computing Surveys **14** (3): 473-530, 1982.

[Smith85] Smith, A. J. *Cache evaluation and the impact on workload choice*, In the 12th ISCA, Boston, MA, 64-73, 1985.

[Smith92] Smith, J. E. and Hsu, W.-C. *Prefetching in supercomputer instruction caches*, In Supercomputing '92, 588-597, 1992.

[SPEC91] SPEC. *The SPEC Benchmark Suite*. SPEC Newsletter. **3:** 3-4, 1991.

[SPEC93] SPEC. *SPEC: A five year retrospective*. *The SPEC Newsletter* **5** (4): 1-4, 1993.

[Taylor90] Taylor, G., Davies, P. and Farmwald, M. *The TLB slice - A low-cost high-speed address translation mechanism*, In the 17th ISCA, Seattle, WA, 355-363, 1990.

[Torrellas92] Torrellas, J., Gupta, A. and Hennessy, J. *Characterizing the caching and synchronization performance of multiprocessor operating system*, In the 5th ASPLOS, Boston, MA, 162-174, 1992.

[Torrellas95] Torrellas, J., Xia, C. and Daigle, R. *Optimizing instruction cache performance for operating system intensive workloads*, In the 21st International Symposium on High-Performance Computer Architecture (HPCA), Raleigh, North Carolina, to appear, 1995.

[Touma92] Touma, W. R. *The Dynamics of the Computer Industry*. University of Texas at Austin. 1993.

[Uhlig94] Uhlig, R., Nagle, D., Sechrest, S. and Mudge, T. *Trap-driven simulation with Tapeworm II*. In the 6th ASPLOS, San Jose, CA, 132-144, 1994.

[Uhlig95] Uhlig, R. *Trap-driven Memory Simulation*, Ph.D. Thesis, The University of Michigan, 1995.

[Wada92] Wada, T., Rajan, S. and Przybylski, S. *An analytical access time model for on-chip cache memories*. IEEE Journal of Solid-State Circuits **27** (8): 1147-1156, 1992.

[Wang89] Wang, W.-H., Baer, J.-L. and Levy, H. *Organization and performance of a two-level virtual-real cache hierarchy*, In the 16th ISCA, Jerusalem, Israel, 140-148, 1989.

[Wiecek92] Wiecek, C. A., Kaler, C. G., Fiorelli, S., Davenport, W. C. and Chen, R. C. *A Model and Prototype of VMS Using the Mach 3.0 Kernel*, In the USENIX Micro-kernels and Other Kernel Architectures Workshop, Seattle, WA, 187-203, 1992.

[Wilton94] Wilton, S. and Jouppi, N. *An enhanced access and cycle time model for on-chip caches*. DEC Western Research Lab. Technical Report 93/5. 1994.