Instruction Fetching: Coping with Code Bloat

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

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

David Nagle
Department of ECE, Carnegie Mellon University
Pittsburgh, PA 15213
dnagle@ece.cmu.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 [Olkutum92, 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 (SPEC90, SPEC92 and the...
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 includes the design of a new collection of workloads that better represents these trends. 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.

In recent years, much of the architecture research community has settled on using the SPEC benchmark suite as a measure of uniprocessor system performance 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, Bort90, 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, Happe92, 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 [Farren93, Hill87, Smith87, Smith92, Pierce95], pipelineing [Jouppi90, Olukotun92, Palcharla94] and bypassing [Hennessy90].

There are also software-based methods for improving I-cache performance. Compilers can reduce instruction conflicts 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

### Table 1: Memory System Performance of the SPEC Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Execution Time (%)</th>
<th>Total Memory CPI</th>
<th>Components of Memory CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>User</td>
<td>OS</td>
<td>l-cache (CPI\textsubscript{instr})</td>
</tr>
<tr>
<td>SPECint89</td>
<td>97%</td>
<td>3%</td>
<td>0.285</td>
</tr>
<tr>
<td>SPECfp89</td>
<td>98%</td>
<td>2%</td>
<td>0.967</td>
</tr>
<tr>
<td>SPECint92</td>
<td>97%</td>
<td>3%</td>
<td>0.271</td>
</tr>
<tr>
<td>SPECfp92</td>
<td>98%</td>
<td>2%</td>
<td>0.749</td>
</tr>
</tbody>
</table>

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.

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

Two related work

1. During the past threeISCAs, over two thirds of the papers dealing with uniprocessor architecture issues used the SPEC benchmarks.
are mainly programs that we actually use in our day-to-day work operating systems in the IBS workload suite. The IBS workloads Ultrix 3.1 and Mach 3.0. Table 2 summarizes the benchmarks and workloads (like those from SPEC) that do not stress instruction-mized memory systems tend to use easily-traceable, single-task workloads they consider on more sophisticated memory system references. Unfortunately, the resulting address traces 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]:

$$ CPI = CPI_{\text{inst}} + CPI_{\text{other}} $$

where $CPI_{\text{inst}}$ is the performance lost to instruction-cache misses and $CPI_{\text{other}}$ is determined by the instruction-issue rate and all

### Table 1

<table>
<thead>
<tr>
<th>Workload</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpeg_play</td>
<td>mpeg_play (version 2.0) from the Berkeley Plateau Research Group. Displays 85 frames from a compressed video file [Patel92].</td>
</tr>
<tr>
<td>jpeg_play</td>
<td>The xloadimage (version 3.0) program written by Jim Frost. Displays two JPEG images.</td>
</tr>
<tr>
<td>gs</td>
<td>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.</td>
</tr>
<tr>
<td>verilog</td>
<td>Verilog-XL (version 1.8b) simulating the logic design of an experimental microprocessor.</td>
</tr>
<tr>
<td>gcc</td>
<td>The GNU C compiler (version 2.6)</td>
</tr>
<tr>
<td>sdet</td>
<td>A multiprocess, system performance benchmark which includes programs that test CPU performance, OS performance and I/O performance. From the SPEC SDM benchmark suite.</td>
</tr>
<tr>
<td>nroff</td>
<td>Unix text formatting program shipped with Ultrix 3.1.</td>
</tr>
<tr>
<td>groff</td>
<td>GNU C++ implementation of the Unix nroff text formatting program. Version 1.09.</td>
</tr>
</tbody>
</table>

### Table 2: The IBS Workloads

<table>
<thead>
<tr>
<th>Workload</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User OS</td>
<td>Description</td>
</tr>
<tr>
<td>Ultrix</td>
<td>Version 3.1 from Digital Equipment Corporation.</td>
</tr>
<tr>
<td>Mach</td>
<td>CMU’s version mk77 of the Mach 3.0 kernel and version uk38 of the 4.3 BSD UNIX server.</td>
</tr>
</tbody>
</table>

### Table 3: Memory Performance of the IBS Workloads

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>User</th>
<th>OS</th>
<th>L-cache (CPI\text{\text{inst}})</th>
<th>D-cache (CPI\text{\text{data}})</th>
<th>Write (CPI\text{\text{write}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBS (Mach 3.0)</td>
<td>62%</td>
<td>38%</td>
<td>0.36</td>
<td>0.28</td>
<td>0.16</td>
</tr>
<tr>
<td>IBS (Ultrix 3.1)</td>
<td>76%</td>
<td>24%</td>
<td>0.19</td>
<td>0.30</td>
<td>0.11</td>
</tr>
<tr>
<td>SPECint92</td>
<td>97%</td>
<td>3%</td>
<td>0.05</td>
<td>0.08</td>
<td>0.06</td>
</tr>
<tr>
<td>SPECfp92</td>
<td>98%</td>
<td>2%</td>
<td>0.05</td>
<td>0.44</td>
<td>0.13</td>
</tr>
</tbody>
</table>
other sources of processor stalls, such D-cache misses, TLB misses, CPU pipeline interlocks and issue constraints. The I-cache component, CPI_{Inst} can be further factored into:

\[ CPI_{Inst} = MPI \cdot CPM \]

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_{Inst}. 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 egntott, espresso and gcc, span the range of SPEC benchmark sizes with respect to I-cache performance. Gee et al. characterize egntott 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 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 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.

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.
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.

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

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.

---

**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. Windows NT also separates its different API servers as given and now focus on ways to design instruction-fetching hardware to help recover some of the performance lost to bloated code.

### 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
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, Przybylski89, 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-piped on-chip L2 cache to both baseline configurations. Figure 3 plots the resulting combined L1 and L2 contributions to CPI<sub>inst</sub> 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.
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_{inst} 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. 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. 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.
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 a 6-byte latency. For these data, the execution model assumes the processor must wait for the entire cache line to refill before it resumes execution.

![Figure 6: Bandwidth and L1 CPI_{instr} vs. Line Size](image)

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.

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. These prefetches are not placed directly into the cache; instead, they are stored in a special memory, called a stream buffer [Jouppi90].

**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. 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,

![Diagram](image)

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

<table>
<thead>
<tr>
<th>Number of Lines Prefetched</th>
<th>Line Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
</tr>
<tr>
<td>0</td>
<td>0.439</td>
</tr>
<tr>
<td>1</td>
<td>0.305</td>
</tr>
<tr>
<td>2</td>
<td>0.270</td>
</tr>
<tr>
<td>3</td>
<td>0.260</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Number of Lines Prefetched</th>
<th>Line Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Bypass Buffers</td>
</tr>
<tr>
<td></td>
<td>With Bypass Buffers</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>0.439</td>
</tr>
<tr>
<td>1</td>
<td>0.305</td>
</tr>
<tr>
<td>2</td>
<td>0.270</td>
</tr>
<tr>
<td>3</td>
<td>0.260</td>
</tr>
</tbody>
</table>

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.
high-performance system, the L1 CPI is achieved with the addition of pipelining. In the economy system. The largest performance improvement in the case of the optimizations, adding an associative L2 cache provides the largest performance gains. The improvement is quite dramatic in the case of the Memory hierarchy.

point that there are numerous tradeoffs with respect to bandwidth, and prefetch policies can influence performance, underscoring the importance of caching all prefetched instructions. 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 L1-L2 interface. The largest performance improvement in the L1-L2 interface is achieved with the addition of pipelining. In the high-performance system, the L1 CPI (0.11 for the 16-byte/cycle configuration) is the dominant factor in total CPI (0.18). While this is an acceptable level of I-cache performance for a single-issue machine, dual- or quad-issue machines with a maximum CPI of 0.50 and 0.25, respectively, will spend a considerable amount of time stalling on I-cache misses.

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 by a mere 0.026. Under SPEC, the optimal L2 cache configuration would have a total CPI 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 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


