Instruction Fetching: Coping with Code Bloat

Richard Uhlig  
German National Research Center for Computer Science  
Schloß Birlinghoven, 53757 Sankt Augustin, Germany  
uhlig@gmd.de

David Nagle  
ECE Department, Carnegie Mellon University  
4902 Forbes Ave, Pittsburgh, PA 15213  
bassoon@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 pipeline memory systems on machines that rely on the use of relatively small primary caches to facilitate increased clock rates. We find that, although of little use for SPEC, the right combination of these techniques provides significant benefit for 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 over time to incorporate new functions, and because memory technologies are constantly changing in capability and cost, it follows that memory-system parameters must be periodically 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 dynamic size 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 can 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]. This is true because for a given integrated-circuit technology, 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) in recent years tend to have low associativity and are limited to 4KB to 16KB [MReport93, MReport94]. The net effect of these trends is that primary caches have not grown in size during the past 10 years [Brunner91]. These hardware and software trends are particularly demanding of instruction caches because code bloat can increase the active instruction working-set sizes that CPU caches must retain close to the processor.

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 for two reasons. First, bloat in operating system code does not affect the SPEC suite because less than 2% of the entire execution time of the SPEC benchmarks is spent in system mode [Gee93]. Second, unlike real-world applications, the SPEC benchmarks are not regularly augmented with new features or restructured to enhance their portability and maintainability. For example, a graphical user
interface has not been added to programs in the SPEC suite as it has to most commercial applications. In fact, the SPEC benchmarks have evolved to be even less demanding of instruction caches with their second release in 1992 (see Section 1). 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 represent these effects.

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 codes. Second, it re-evaluates the effectiveness of several previously-proposed methods for improving instruction-fetching performance in the context of bloated code and new technology constraints. 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.

Our study of software-development trends includes the design of a new collection of workloads which we call the Instruction Benchmark Suite (IBS). In an analysis that takes into account the full activity of both the user-level and kernel-level components of these workloads, we characterize and compare this workload suite against programs in SPEC92. This analysis confirms that code-bloat trends lead to increased I-cache capacity and associativity requirements.

Then, starting with the assumption that future high-speed processors will have to limit their primary I-caches to small, direct-mapped memories [Jouppi94], we evaluate various methods for reducing primary I-cache miss penalties. These methods include adding and tuning a second level of on-chip cache, and then implementing a variety of optimizations to the interface between the primary and secondary cache, such as increasing the transfer bandwidth, prefetching instructions, bypassing the cache on a line refill and pipelining portions of the memory system. Our analysis shows that the IBS workloads are more sensitive to these optimizations than the SPEC benchmarks are, and exhibit larger absolute improvements in performance when these optimizations are applied.

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 performance1 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, Clark85a, Clark85b, Smith85, Alexander85, Alexander86, Agarwal88, Borg90, Mogul91, Torrellas92, Flanagan93, Chen93c, Chen94a, Huck93, Cventanovic94, 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 the cache line sizes [Przybylski90, prefetching [Farrens89, Hill87, Smith78, Smith92], pipelining

1. During the past three ISCAS, over two thirds of the papers dealing with uniprocessor architecture issues used the SPEC benchmarks [ISCA92, ISCA93, ISCA94].
This work distinguishes itself from previous studies by re-evaluating a collection of aggressive hardware-based\(^1\) instruction fetching optimizations on a more challenging workload that is designed to represent current trends in software development. The address traces that we have collected from the IBS workloads are available to the general architectural community so that our findings can be confirmed, and to enable further architectural studies by other researchers.\(^2\)

### 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 and that we feel exhibit poor performance. Our Mosaic WWW browser frequently invokes mpeg_play, jpeg_play and gs which seem to be the limiting factor to good interactive performance, just behind lack of network bandwidth. 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. The spim workload is a MIPS-binary code emulator that is set to emulate the SPEC espresso program. 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 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 stalls the DECstation while unloading the trace buffer in the logic analyzer.

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>User OS</th>
<th>I-cache (CPI(_{\text{Instr}}))</th>
<th>D-cache (CPI(_{\text{Data}}))</th>
<th>Write (CPI(_{\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>
</tr>
<tr>
<td>IBS (Ultrix 3.1)</td>
<td>76%</td>
<td>24%</td>
<td>0.19</td>
<td>0.30</td>
</tr>
<tr>
<td>SPECint92</td>
<td>97%</td>
<td>3%</td>
<td>0.05</td>
<td>0.08</td>
</tr>
<tr>
<td>SPECfp92</td>
<td>98%</td>
<td>2%</td>
<td>0.05</td>
<td>0.44</td>
</tr>
</tbody>
</table>

---

1. We do not examine the aforementioned software-based methods in this paper.
2. Although this work only examines instruction references, the IBS traces include all instruction and data memory references, as well as instruction traces.
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].

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 performance lost to I-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).

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, xlisp and gcc, span the range of SPEC benchmark sizes with respect to I-cache performance. Gee et al. characterize eqntott as small, espresso and xlisp 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. 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 25% higher than it is under Ultrix 3.1).

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.

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.
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.

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. We also include the logic simulator `verilog`, which has steadily grown in size with each new release, and 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. The IBS workloads represent two different software techniques that increase application portability: 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, such as Windows NT [Custer93], Mach 3.0 [Accetta86], and others [Bomberger92, Cheriton84, Malan91, Rozier92, Wicek92] 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 overheads 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 over 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 represent ABI emulation in our benchmark suite with `spim`, which emulates the MIPS instruction set on a variety of other architectures including the SPARC, HP-PA, x86 and the MIPS itself [Larus91]. The IBS `spim` workload emulates a MIPS binary of the `espresso` SPEC benchmark and exhibits an MPI that is approximately 3 times the normal MPI of `espresso` when not being emulated.

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 25% 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

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Economy</th>
<th>High Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next Level in Hierarchy</td>
<td>Main Memory</td>
<td>Ideal Off-chip Cache</td>
</tr>
<tr>
<td>Latency (Cycles)</td>
<td>30</td>
<td>12</td>
</tr>
<tr>
<td>Bandwidth (Bytes/Cycle)</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>CPI\textsubscript{instr} (SPEC)</td>
<td>0.54</td>
<td>0.18</td>
</tr>
<tr>
<td>CPI\textsubscript{instr} (IBS)</td>
<td>1.77</td>
<td>0.72</td>
</tr>
</tbody>
</table>

**Table 5: CPI\textsubscript{Instr} for Base System Configurations**

Both configurations contain an 8-KB, direct-mapped, on-chip L1 cache. In the economy configuration, the L1 cache is backed by main memory (30 cycle latency, 4 bytes per cycle) while the high-performance configuration is backed by a large, off-chip cache (12 cycle latency, 8 bytes per cycle). For our base configurations, we consider an ideal off-chip cache with zero contribution to CPI\textsubscript{Instr}. Our simulations show that for IBS, a 512-KB, direct-mapped L1 cache would be close to ideal, contributing only 0.03 to the total CPI\textsubscript{Instr}. These latencies and bandwidths were selected by surveying a number of processors in [MReport94].
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

The IBS workloads were selected to represent basic pressures on software development that invariably lead to larger programs. 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 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 imple-
ments 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 exploring 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, Smith92]. 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 best case 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\textsubscript{instr} across a range of L2 cache and line sizes. For even the smallest L2 cache, the performance of the economy configuration improves over the baseline configuration, provided that the line size is tuned. However, the high-performance system requires at least a 32-KB or 64-KB on-chip cache to improve over its baseline. At 64-KB, the economy configuration’s performance actually matches the high-performance baseline configuration (Table 5). 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 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. Figure 4 shows the benefits of L2 cache associativity. Notice that both configurations exhibit the greatest reduction in CPI\textsubscript{instr} (approximately 40%) between the direct-mapped and 2-way set-associative caches; further increases in associativity (up to 8-way set-associative) only reduce CPI\textsubscript{instr} another 20%.

Increased associativity improves miss rates by reducing conflict misses. As a result, associativity also reduces variability in performance caused by OS page-mapping effects in a physically-indexed cache [Kessler91, Sites88]. 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 spim, 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].

These plots show variability in performance over multiple runs of the same workload in a physically-indexed I-cache. Performance varies because the allocation of virtual pages to physical cache page frames is different from run to run. Variability is reported on the y-axis in terms of one standard deviation of CPI\textsubscript{instr}.

---

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\textsubscript{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 more accurately account for these effects.
A final advantage of associativity is that it allows designers to 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 not enough area to double the size of the cache. The ability to 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 L1 I-cache the performance bottleneck. Although the basic structure of (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 plots the impact of increasing L1-L2 transfer bandwidth on L1 cache performance. The plot 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 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. 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

<table>
<thead>
<tr>
<th>Number of Lines Prefetched</th>
<th>Line Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.439 0.335 0.297</td>
</tr>
<tr>
<td>1</td>
<td>0.305 0.271 —</td>
</tr>
<tr>
<td>2</td>
<td>0.270 — —</td>
</tr>
<tr>
<td>3</td>
<td>0.260 — —</td>
</tr>
</tbody>
</table>

Table 6: Prefetching

This table shows the L1 CPI_instr 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.

The cells with an “—” denote data points that are either not reasonable, or that show an increase in CPI_instr.

1. Our simulations also show that a 64-byte line with 16-byte sub-block placement 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.
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_{\text{instr}} with and without bypassing logic.

A different policy is to only cache prefetched lines if they are used by the processor. This eliminates any cache pollution due to prefetching. However, our simulations show that this policy does not improve performance over prefetching into the cache. Unless prefetched instructions are used almost immediately, they are likely to be replaced due to the limited number of bypass buffers. Placing the prefetched data into both the cache and bypass buffers increases the chance that a prefetched instruction will be available if it is accessed later in a program run. This is particularly useful for short subroutine calls and forward branches in loops.

Pipelining

The final enhancement we investigate is pipelining the L1-L2 interface, which 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 pipeline is kept busy with sequential prefetch requests. Prefetches are stored in a special buffer, called a Stream Buffer [Jouppi90]. The stream buffer is a fully-associative memory with 1 or more lines and is very similar to a bypass buffer.

The results (see Table 8) show that stream buffers effectively improve I-fetch performance until they reach sizes of about 6 lines, after which the improvements are marginal. However, stream-buffer performance might be further improved by implementing multiple stream buffers and switching between the stream buffers on subroutine jumps. This would be particularly useful for short leaf-node function calls. Another optimization would be to add a target prefetch table [Smith78]. This table would store the addresses of non-sequential pairs of lines. As every fetch or prefetch is issued into the pipelined memory system, the table is checked to see if there exists an entry. If so, the next prefetch would use the address stored in the table and not a sequential address. We are currently evaluating both techniques.

### Table 7: Bypassing

<table>
<thead>
<tr>
<th>Number of Lines Prefetched</th>
<th>Line Size (Bytes) No Bypass Buffers</th>
<th>Line Size (Bytes) With Bypass Buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>0</td>
<td>0.439</td>
<td>0.335</td>
</tr>
<tr>
<td>1</td>
<td>0.305</td>
<td>0.271</td>
</tr>
<tr>
<td>2</td>
<td>0.270</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>0.260</td>
<td>—</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Number of Lines in Stream Buffer</th>
<th>16 Bytes/Cycle CPI_{\text{instr}}</th>
<th>32 Bytes/Cycle CPI_{\text{instr}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.439</td>
<td>0.287</td>
</tr>
<tr>
<td>1</td>
<td>0.267</td>
<td>0.186</td>
</tr>
<tr>
<td>3</td>
<td>0.184</td>
<td>0.137</td>
</tr>
<tr>
<td>6</td>
<td>0.147</td>
<td>0.118</td>
</tr>
<tr>
<td>12</td>
<td>0.122</td>
<td>0.103</td>
</tr>
<tr>
<td>18</td>
<td>0.114</td>
<td>0.099</td>
</tr>
</tbody>
</table>

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

---

1. Pipelining the memory system also allows data references to be mixed with prefetch requests.
reduction from an initial baseline design, but shows that instruction-fetch overhead will be a dominant component of the execution time of multi-issue processors with very short cycle times, operating out of small primary caches.

The full traces used in this study are available by sending E-mail to the authors. These traces include both instruction and data memory references and cover the full activity of all user and kernel processes. By making these full traces available, we hope to encourage wider explorations of memory systems and the comparison of results between investigators.

7 References


