Runahead Execution vs. Conventional Data Prefetching in the IBM POWER6 Microprocessor

Harold W. Cain   Priya Nagpurkar

IBM T.J. Watson Research Center
Yorktown Heights, NY
{tcain, pnagpurkar}@us.ibm.com

Abstract

After many years of prefetching research, most commercially available systems support only two types of prefetching: software-directed prefetching and hardware-based prefetchers using simple sequential or stride-based prefetching algorithms. More sophisticated prefetching proposals, despite promises of improved performance, have not been adopted by industry. In this paper, we explore the efficacy of both hardware and software prefetching in the context of an IBM POWER6 commercial server. Using a variety of applications that have been compiled with an aggressively optimizing compiler to use software prefetching when appropriate, we perform the first study of a new runahead prefetching feature adopted by the POWER6 design, evaluating it in isolation and in conjunction with a conventional hardware-based sequential stream prefetcher and compiler-inserted software prefetching.

We find that the POWER6 implementation of runahead prefetching is quite effective on many of the memory intensive applications studied; in isolation it improves performance as much as 36% and on average 10%. However, it outperforms the hardware-based stream prefetcher on only two of the benchmarks studied, and in those by a small margin. When used in conjunction with the conventional prefetching mechanisms, the runahead feature adds an additional 6% on average, and 39% in the best case (GemsFDTD).

1. Introduction

By necessity, most new prefetching research occurs in the context of experimental frameworks consisting of somewhat immature simulators or compilation infrastructure, with comparisons to baseline machine models whose memory hierarchy may not include the aggressive tuning that occurs for commercially available systems. While such environments are perfectly reasonable to demonstrate the potential of a new prefetching algorithm, it is also important to periodically revisit and characterize the current state-of-the art on solid ground: a commercially available system that has been heavily optimized to employ prefetching as much as will be economically beneficial.

In this paper we do not propose any new prefetching algorithm, but instead measure natively the benefits due to prefetching observed in a highly optimized POWER6-based commercial server, providing a reference point for future researchers and practitioners of the current prefetching state-of-the-art, as well as to gauge the opportunity for further prefetching research.

The POWER6 system used for our experiments supports software-initiated load and store prefetching, a hardware-based sequential stream prefetcher, and the first commercially available form of run-ahead prefetching, called *load lookahead prefetching* in the POWER6 context. Using performance counters, and a set of undocumented hardware switches to independently control the various prefetching modes, we study the performance of various applications using each type of prefetching mechanism, independently as well as in conjunction with one another.

We find that when evaluating each in isolation, the conventional hardware prefetching technique is the most effective of the three, yielding an average speedup of 74%, and delivering performance improvements for most of the benchmarks we analyzed. The load lookahead prefetching feature yields relatively lower performance improvements, with an average 10% improvement across all benchmarks, but is at times effective even when hardware prefetching is not, specifically for our Java benchmarks. Software prefetching, on the other hand, is beneficial for only a few of the benchmarks we analyzed, however the XL compiler used often suppresses software prefetching in reliance of the hardware stream prefetcher. In spite of an impressive 90% average improvement in performance from applying all three techniques together, there is room for further improvement with on average 22% of the time still being spent on data miss stalls.

In summary, this paper makes the following contributions:

- We present the first study of run-ahead prefetching in a commercially available design, finding that it provides significant benefits in isolation or in conjunction with other types of prefetching, although its benefits in isolation are substantially less than those of a conventional sequential stream prefetcher.

- We measure the performance improvements for each type of prefetching across applications commonly used in other prefetching studies, providing a reference point for expected performance improvements and opportunity.
- We demonstrate that data cache miss stalls remain a significant performance problem, even in the context of a high-end server including relatively large caches (a 4MB private L2, and 32MB L3 shared by 2 cores) and a variety of aggressive prefetching mechanisms.

This paper is organized as follows: we begin in Section 2 with an overview of hardware prefetching support in the POWER6 design, as well as software prefetching mechanisms employed by the XL compiler and J9 Java Virtual Machine used in our experiments. In Section 3 we describe the experimental setup and methodology used for our evaluation, which is described in Section 4. Related work is summarized in Section 5, followed by conclusions.

2. Prefetching in the POWER6 Microprocessor

POWER6 includes support for a variety of prefetching mechanisms: several forms of software-prefetch instructions, a hardware sequential stream prefetcher, and a runahead "load-lookahead" prefetching mechanism. We begin this overview with the most novel of the bunch, load-lookahead prefetching.

2.1. Load-Lookahead Prefetching

Because the POWER6 design employs a high-frequency in-order microarchitecture, there was a desire to "buy-back" some of the benefits of an out-of-order design using a runahead prefetching mechanism [12], called *load-lookahead prefetching* (LLA) in the POWER6 design. This LLA prefetching mechanism works as follows:

When a L1 data cache or DERAT[1] miss occurs due to a load, the pipeline switches to LLA mode. In LLA mode, the pipeline continues fetching and executing instructions, in the hope that these instructions will initiate useful prefetches while the initial miss is outstanding. In this mode instructions are dispatched from the 64-entry instruction fetch buffer that is used to queue instructions fetched by the front-end of the pipeline before they can be dispatched by the in-order core. When executing in LLA mode, entries in this buffer are not deallocated. They are retained, so that once LLA mode execution completes, at the time that the inital miss returns, they will be re-dispatched and executed conventionally.

The primary implementation challenge in a design supporting runahead execution is the management of speculative state. Because execution resumes from the instruction that initiated lookahead mode, the architectural state of the machine must precisely reflect the machine state that existed before that instruction was executed. Supporting rollback for each type of update to architectural state (e.g. stores to memory, or writes to various register types) is costly, and it is therefore desirable to approximately maintain machine state during runahead mode. The degree of approximation is a trade-off between the accuracy of the run-ahead thread, and the amount of resources used to maintain speculative state. Should instructions in run-ahead mode receive incorrect input values due to this approximation, the performance gains from runahead will be impacted, but not program correctness.

The POWER6 design handles various components of architectural state differently. Because address calculations (the source of addresses for prefetching), are not frequently dependent on floating point registers, floating point instructions are treated as no-ops in LLA mode. Due to the difficulty checkpointing memory, store instructions are also treated as no-ops. When the core operates in single-thread mode, general-purpose register updates are performed speculatively, leveraging the spare SMT thread's context. When operating in SMT mode, however, general purpose register writes are dropped during LLA execution. Dependences may be maintained through forwarding paths, but a lack of speculative register writes result in reduced ability for LLA mode to productively run ahead, since the register contents will be inaccurate, leading to incorrect control flow and address calculations. For the purposes of this paper, we focus on performance improvements in single-threaded mode only, however do note that the latency tolerance benefits of multithreading outweigh any degradation in LLA mode benefits for multithreaded applications.

As has been pointed out by others, the performance of runahead execution is very dependent on branch prediction accuracy [12], [8], [21], [1], since prefetches are less likely to be useful if they are from the wrong path. The POWER6 processor also includes a mode in which branch prediction history is warmed during runahead; this mode is disabled by default in POWER6 systems, but we also test its benefits in Section 4.

2.2. Hardware Stream Prefetcher

Since POWER4, POWER systems have also included a hardware-based data prefetching engine targeting stride-one workloads. By monitoring the addresses of L1 data cache misses, it recognizes sequential misses, allocating a miss stream when encountering misses to adjacent cache lines, in either a descending or ascending reference pattern. Once a sequential miss stream is detected, the prefetcher will prefetch up to two lines into the L1 data cache, and up to a programmable maximum of 24 lines into the L2 cache, staging data into each cache level as prefetched blocks are touched. Up to sixteen simultaneous load and store streams are supported, with store streams exclusively prefetching blocks

1. Data Effective to Real Address Translation Cache, the equivalent of a TLB in POWER processors

into the L2 cache only. Since streams are maintained using effective addresses[2], each prefetch stream is terminated when a page boundary is reached.

2.3. Software Prefetching

In POWER-based systems, the data-cache-block touch *dcbt* and data-cache-block touch for store *dcbtst* instructions can be used to prefetch a single data cache block into the L1 data cache, in either a shared or exclusive coherence state, respectively. In addition, POWER6 provides a means for software to control the hardware-based data prefetching engine via an enhanced *dcbt* variant, called *edcbt*, that allows initiation of prefetch streams and control over parameters like the prefetch stream to use, the direction, and the length of the prefetch. The benefit from software prefetching depends on how effectively compilers employ static analysis techniques and other heuristics to insert these instructions.

The JIT compiler in the Java Virtual Machine that we use to run our Java benchmarks uses profile information and heuristics to selectively insert the *dcbt* instruction to perform targeted prefetching for loops. *dcbt* is also used during the allocation of thread local heaps. In the version of the Java Virtual Machine we used, *edcbt* is not used.

The IBM XL optimizing compiler used in this study includes separate front-ends for Fortan and C/++, but shares a common back-end such that optimizations described here apply to both the Fortran and C/C++ benchmarks. The XL compiler includes the following optimizations for data cache performance:

- **dcbt insertion**: In loops where the compiler can identify access patterns, and whose iteration count is sufficiently large, the compiler will automatically insert dcbt or dcbtst instructions for data to be referenced in subsequent loop iterations. When compiling for the POWER6 processor, however, the compiler applies this optimization selectively. When a loop contains a sequential access patterns that is amenable to hardware prefetching, insertion of software prefetch instructions is suppressed; consequently, this optimization is only performed when the access pattern includes a stride larger than the 128B cache block size, which are not detectable by the hardware prefetcher.
- **Stream throttling**: Insertion of ecdbt instructions to terminate prefetch streams in progress by the hardware prefetcher. While POWER6 supports both software initiated creation and termination of streams, the compiler only takes advantage of stream termination, performed when a loop which has been identified as stream-friendly terminates.
- **Stream splitting**: A loop splitting optimization is employed when the number of memory streams in a loop is expected to exceed the 16-entry limit of the hardware prefetcher.
- **dcbz prefetching**: For store streams where an entire cache block is being written, the referenced cache block is pre-allocated in-cache using the POWER dcbz (data cache block zero) operation, which installs a zero-filled cache block without requiring data to be fetched from memory.

2. In the POWER architecture's two-level address translation, effective addresses are the first of two levels of virtual address.

Of the optimizations described above, dcbz prefetching cannot be controlled in our experiments since the dcbz instruction cannot be disabled, so this optimization is always on during our experiments. Stream splitting is always enabled, but is not meaningful when the hardware prefetcher is disabled. The dcbt, dcbtst, and edcbt instructions are always present in the instruction stream for our experiments, but are treated as no-ops depending on certain hardware switches. There are no compiler optimizations related to the LLA prefetching mode.

3. Experimental Methodology

3.1. Power6 System Parameters

Experiments were performed on a 4-core 4.7 GHZ IBM POWER6 p570 server, running AIX 6.1. The POWER6 design contains two cores per chip, each including separate 64KB private L1 instruction and data caches. The 8-way set associative data cache is optimized for low latency access, allowing for a single-cycle load-to-use penalty for fixed-point operations, and a zero-cycle penalty for floating point operations. It is a write-through design, backed by a 4MB unified L2 cache with a 5 ns access latency, which is private per core and inclusive of the L1 d-cache. An off-chip 32 MB 16-way set-associative L3 victim cache is shared by the two cores, with a 40 ns access latency. Cache line size is fixed in all cache levels at 128B.

Each chip contains two memory controllers, each responsible for one half of the address space, interleaved by cache line. The characteristics of the memory system are heavily influenced by the populated capacity, since DRAMS are daisy-chained in high-capacity configurations. POWER6 systems support up to 8TB of 800MHZ DDR2 DRAM. The system used for these experiments contained 22.5GB of memory, with a memory acess latency of 110ns. Le et al. describe further details of the POWER6 design [19].

3.2. Applications

We used the SPEC CPU 2006 benchmark suite, as well as a handful of Java applications from the SPEC JVM 2008 benchmark suite, and SPECjbb2005.

The C/C++ SPEC benchmarks were compiled using the XL C/C++ Compiler, Enterprise Edition V9 for AIX. Fortran benchmarks we compiled with the XL Fortran Enterprise Edition V11.1 for AIX. All benchmarks were compiled with peak optimization flags, identically to their configuration for submission of SPEC CPU results, and measured using the reference input set. A full listing of benchmark compiler switches is included in Appendix A.

For the Java benchmarks, we used IBM's J9 Java Virtual Machine, version 1.6 (SR2). The measurement interval is preceded by a 30 second warm-up period to eliminate startup and JIT compilation overhead from our measurements. In addition, to compensate for the indeterminism and cross-run variability in Java programs, the numbers we report are averaged across five separate runs.

3.3. Measurements and Metrics

In the analysis that follows, data is presented for the following six different prefetching configurations.

- **nopref**: No prefetching enabled. Software prefetch instructions are treated as no-ops, and both hardware-based sequential and LLA prefetching are disabled.
- **swpref**: Software-only prefetching is enabled. LLA and hardware-based prefetching are disabled. Enhanced dcbt instructions, which provide hints to the hardware prefetcher, are ignored.
- **hwpref**: Hardware-based Sequential prefetching is enabled. LLA prefetching and all software prefetching, including enhanced dcbt instructions, are disabled.
- **hwswpref**: Both hardware-based sequential prefetching, and software prefetching (including enhanced dcbt hints) are enabled. LLA prefetching is disabled.
- **llapref**: LLA prefetching is enabled, hardware-based stream preftcher and software prefetching is disabled.
- **allpref**: All prefetching mechanisms enabled. This is the default system configuration.

All experiments were performed using a single application thread, so a single core was utilized with SMT disabled. For each prefetching configuration, we measure performance, either in terms of execution time or throughput, and a number of relevant micro-architectural metrics using the the core's performance monitoring unit (PMU). The PMU uses dedicated, per-thread counters to measure up to four performance events concurrently. In addition to counting events, the PMU counters can also be programmed to attribute processor stalls to specific events.

4. Analysis

In this section we present our analysis, primarily based on data collected using hardware performance counters. We start with a look at the magnitude of the data cache stall problem to assess the potential of data prefetching techniques, and then investigate the performance of the different techniques currently available.

4.1. Data Misses and Prefetching Potential

We used the *nopref* configuration, in which none of the prefetching techniques are enabled, to gauge the opportunity for prefetching techniques based on the amount of time spent in data miss related stalls. These include stalls due to data cache misses and data address translation misses. Figure 1 shows the percentage of cycles spent in data miss related stalls for all of our benchmarks, with different colors designating separate benchmark suites. The C language benchmarks, particularly those belonging to the SPEC FP suite, show greater opportunity with eight of the 27 benchmarks spending more than 50% of their time stalled on data misses. The worst of our Java benchmarks, on the other hand, spend about 20% of their time stalled on data misses. On an average, the time spent in data miss related stalls, and hence the size of the opportunity for data prefetching techniques, is a significant 31%. As we can see from the figure, there are also a few benchmarks in every suite that do not suffer significantly from data misses, and are therefore not interesting from a data prefetching perspective. Consequently, the following experiments omit data for the following benchmarks, which spend fewer than 15% of their time stalled in data cache and data translation misses: perlbench, bzip2, gobmk, hmmer, sjeng, h264ref, gamess, gromacs, namd, povray, calculix, compiler, and mpegaudio.

4.2. Comparing Prefetching Performance

Figure 2 shows the speedup due to each prefetching configuration relative to the baseline *nopref* configuration in which all prefetching is disabled. The hardware stream prefetcher demonstrates significant performance improvements across the board, causing speedups as high as 3.7 (libquantum), with average speedup of 1.73. A few applications, including all of the Java apps, do not benefit at all, but most benefit greatly. For this set of benchmarks, software prefetching is generally ineffective, showing no performance improvements for the vast majority of benchmarks (recall that the compiler suppresses prefetching in loops that are predicted to be amenable to hardware prefetching). The milc application from SPEC FP is the only app shown here with significant performance gain, a 67% speedup. The only other two benchmarks exhibiting any improvements from software prefetching are GemsFDTD and derby, 1% and 2% respectively. On average, software prefetching alone is worth 2% for these applications, with this milc outlier responsible for skewing the average up to 2%. Combining hardware and software prefetching provides additional benefits for milc, while other benchmarks are unaffected.

Load lookahead prefetching, while less beneficial than the hardware stream prefetcher, does provide significant performance benefits for many applications, as much as 36% (relative to nopref) for GemsFDTD, and 10% on average. LLA prefetching outperforms the hardware stream prefetcher in only two benchmarks (SPECjbb2005 and omnetpp), but in these cases the performance improvements are very small (2% and 4% respectively).

When combined with hardware and software prefetching (*allpref*), load-lookahead continues to provide benefits, on

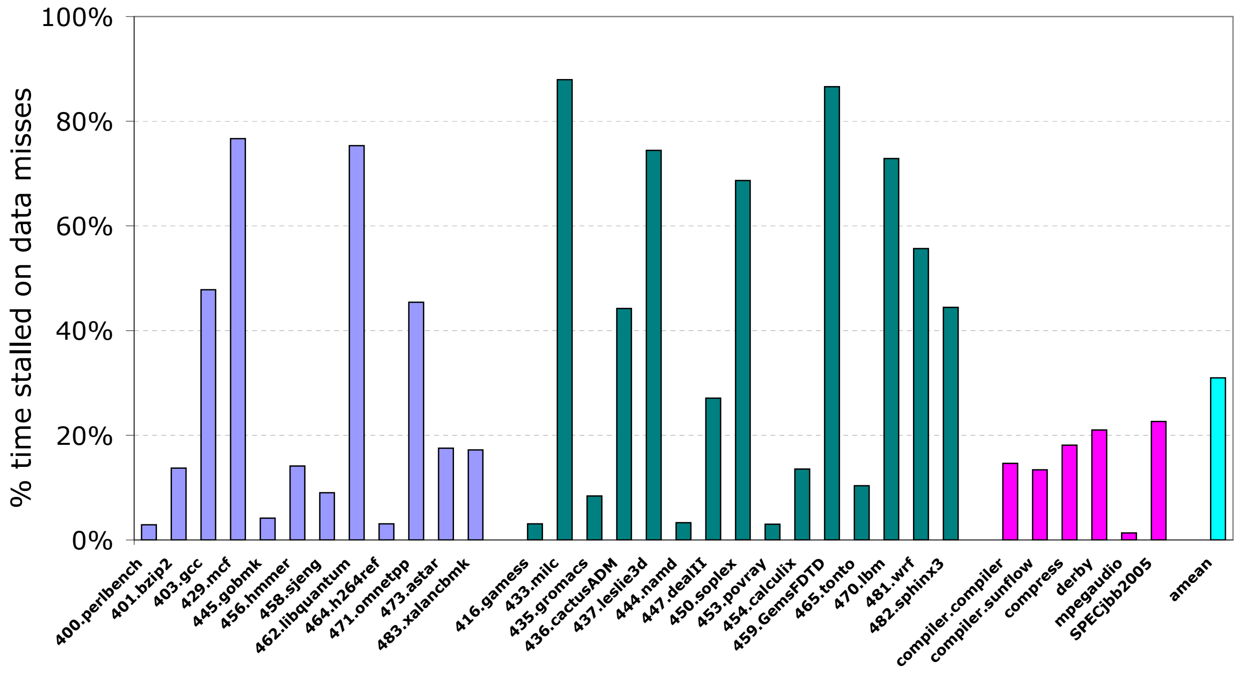

Figure 1: Dispatch stalls caused by L1 data cache or data translation misses, without any prefetching enabled.

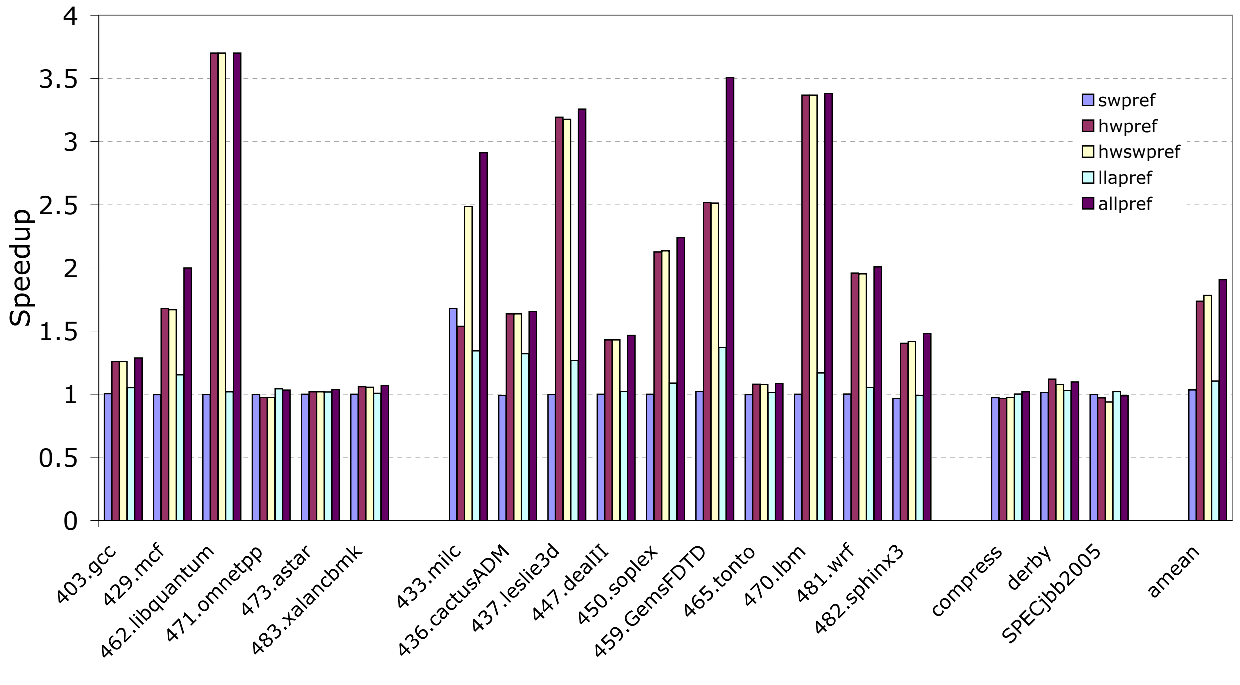

Figure 2: Speedup Due to Different Prefetch Combinations (relative to all prefetching disabled).

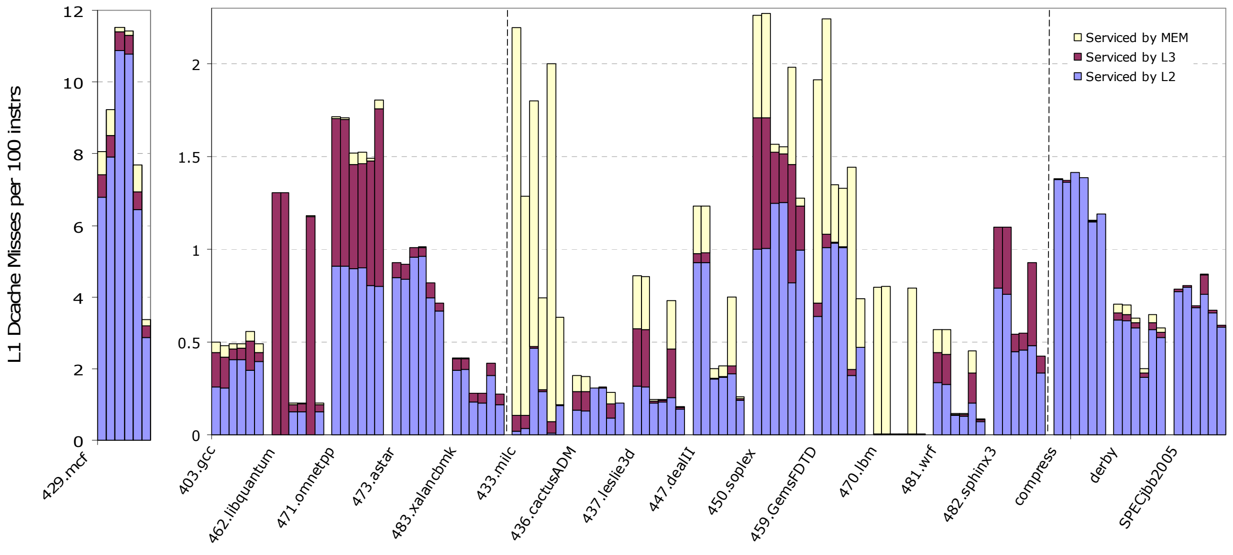

Figure 3: L1 Data Cache Misses and where they are serviced from. Each cluster of bars represents one benchmark. Individual bars in a cluster represent the *nopref, swpref, hwpref, hwswpref, llapref,* and *allpref* configurations in that order from left to right. Note that the Y-axis scale for mcf and the rest of the benchmarks is different.

average 6% and over 15% for 3 of the benchmarks. In the best case (GemsFDTD), it adds 39% performance improvement when used in conjunction with the other forms of prefetching.

Effect on Data Cache Misses. We next take a closer look at the effect of different prefetching techniques on data cache miss behavior. Figure 3 shows the L1 data cache miss rate and the location from which L1 data cache misses are serviced. For each of the benchmarks analyzed, we plot the L1 data cache miss rate for all six configurations in the following order from left to right: *nopref, swpref, hwpref, hwswpref, llapref,* and *allpref*. The total height of a stacked bar denotes the total number of L1 data cache misses per 100 instructions. These are further broken down into categories based on the number of misses that were serviced from the on-chip L2 cache, the off-chip L3 cache, and from memory. Note that this data pertains to demand misses only.

In keeping with the speedup numbers discussed in the previous section, we observe that hardware prefetching is successful in reducing the miss rate for most of our benchmarks. Moreover, it reduces the number of high latency transfers required to service L1 data cache misses. The latter is sometimes even more effective than reducing the miss rate. For *mcf*, for example, the L1 d-cache miss rate with hardware prefetching enabled is the highest across all configurations, but also involves the least number of L3 and memory accesses. The speedup achieved by hardware prefetching is also the highest for *mcf*. For two of our three Java benchmarks, LLA prefetching is most effective in reducing the miss rate. Hardware, software, and LLA prefetching also seem to share a synergy, so that the cumulative effect of all three, as observed in the default *allpref* configuration often yields higher benefits than any one technique applied in isolation.

4.3. Load Lookahead Analysis

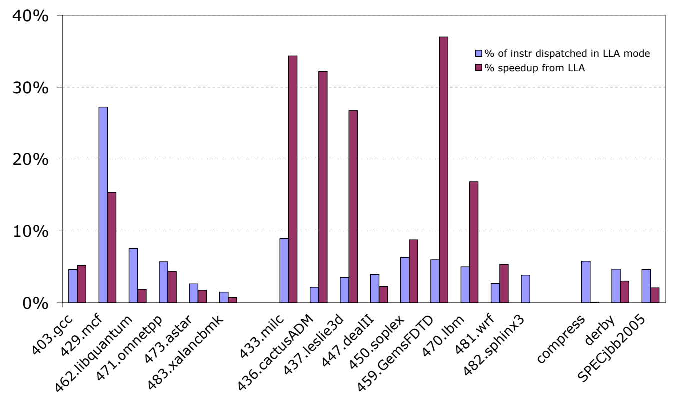

Figure 4: Percentage of instructions dispatched in LLA mode vs. LLA speedup (relative to nopref)

Figure 4 shows the percentage of instructions that are dispatched in LLA mode, versus its performance benefits. The two do not appear to be at all correlated. Of the benchmarks shown, six gain more than 15% performance due to LLA, but of those six the percentage of instructions dispatched in LLA mode varies between 2% and 26%.

In addition to the top level parameters for toggling LLA mode, the system also contains several other switches for controlling the behavior of LLA mode, with which we also experimented. Lookahead can also be optionally used to warm branch prediction history structures. When enabling this option, we found that the benefits were negligible for most applications. The system also includes a switch that disables prefetching and branch history warming in LLA mode, inducing the overhead associated with switching between LLA and normal mode at each d-cache and DERAT

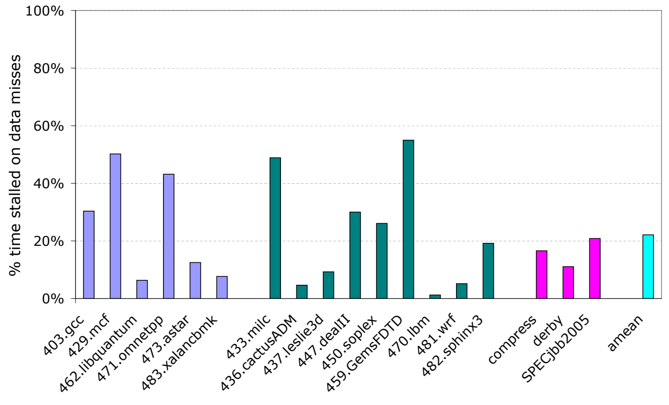

Figure 5: Dispatch stalls caused by L1 data cache or data translation misses when all currently available prefetching techniques are enabled.

miss, without any of the benefits. We tested this mode on a subset of the benchmarks to study the inherent overhead of the LLA mechanism, and found that this LLA-induced overhead contributed to less than 0.2% runtime, so the mechanism is itself efficient; the observed benefits are not weighed down by any significant overhead.

4.4. The Potential for Further Prefetching Gains

After enabling each of the prefetching mechanisms, we are left with a familiar picture. Many benchmarks continue to spend a large fraction of their time waiting on data misses to be serviced. Figure 5 shows the percentage of time each benchmark spends with the pipeline's dispatch stage stalled waiting on a L1 data cache or data translation miss. (This chart is analogous to Figure 1, which shows stall cycles before applying prefetching.) While the average stall time has dropped significantly from 30%, it remains high, at 22%. Some benchmarks(omnetpp, astar, xalancbmk, compress, SPECjbb2005)) were improved by only a small amount using the implemented prefetching mechanisms. Others were improved dramatically (mcf, libquantum, milc, leslie3d, dealII, soplex, GemsFDTD, lbm, wrf, sphinx3). Of these benchmarks, further prefetching opportunity is marginal for a few (libquantum, leslie3d, lbm, wrf), but significant for the rest.

Obviously, ample opportunity remains for improving performance through novel prefetching techniques. While this opportunity may be offset by latency-tolerating mechanisms like multithreading, we believe that single-thread performance will continue to be important for many applications.

5. Related Work

Jouppi proposed the first hardware prefetcher capable of detecting streaming, which is the basis for many prefetchers found in commercial designs today, including POWER6 [16]. Numerous variations have been proposed, each correlating prefetches with a variety of dynamic information. Baer and Chen describe a prefetching mechanism that uses instruction address to predict subsequent data accesses [2]. Charney and Reeves first proposed general hardware-based correlated cache prefetching[6]. Joseph and Grunwald determine correlations using a Markov prediction process[15]. Lai et al. correlate prefetches with the prediction of dead cache blocks [18]. Cooksey et al. base correlations on the contents of incoming cache blocks [11].

A number of other papers have explored correlations of misses among a fixed region of memory, triggering multiple prefetches when a miss in that region occurs [5][17][26][32]. Hu et al.[13], and Nesbit and Smith[22], develop area and complexity-effective mechanisms for tracking correlations. Another area of prefetching research has explored pre-execution of part or all of the program using a parallel helper thread [10][24][25][30][34]. A more thorough summary of prior work in hardware prefetching can be found in Vanderwiel and Lilja's survey [31].

A lot of research has also gone into software techniques that employ static or dynamic program analysis to perform compiler-enabled data prefetching [20], [4], [29], [9], [33], [27], [14]. These techniques range from exploiting regular data access patterns resulting from loops and array accesses, and statically analyzing linked data structures to more complex dynamic schemes that use profiles gathered at runtime to infer data access patterns. Modern compilers, including JIT compilers for Java, incorporate some of these techniques.

The area of runahead execution also has a rich recent history, starting from its introduction by Dundas and Mudge, who demonstrated its benefits for increasing memory-level parallelism in an in-order pipeline[12]. Mutlu and Patt, and Barnes et al. subsequently demonstrated the benefits of runahead execution in the context of out-of-order [21] and EPIC [3] microarchitectures, respectively. There have been a number of subsequent enhancements since then involving more sophisticated checkpoint mechanisms [1] as well as the ability to runahead without requiring the re-execution of subsequent instructions [28].

While the POWER6 design represents the first commercially available processor supporting runahead execution, it has been well documented that the Rock Processor from Sun Microsystems was also slated to support an aggressive microarchitecture including both a runahead mode involving re-execution (Scout mode) and an enhancement in which instructions need not be re-executed (SST mode)[7][8]. In attempting to compare our own results to the performance improvements reported in those studies, we note two experimental differences that prevent an apples-to-apples comparison: 1) with double the L1 cache capacity, and a 32MB L3 cache, the POWER6 system should observe fewer data related stalls to begin with, and consequently less opportunity for prefetching benefits, and 2) it is unclear whether a separate conventional hardware prefetcher was used in those studies. Consequently, a true comparison is a subject of future work.

Many of the factors that make fair comparisons of different evaluations of microarchitectural techniques difficult were described by Perez et al.[23], whose study also included a head-to-head comparison of a variety of proposed prefetching techniques. While their work focused on a number of proposed prefetching mechanisms in the context of an experimental simulation environment, we compare prefetching mechanisms that are sufficiently mature to have been implemented in a commercial product. Hopefully, our study may be useful as a reference in future studies similar to Perez et al.

6. Conclusions and Future Work

In this paper, we have taken a snapshot of the prefetching mechanisms employed by one high-performance commercial server design, have examined the benefits of each prefetching form independently and together, and gauged opportunity for further performance improvements due to prefetching. We find that therunahead prefetching feature delivers significant performance improvement when used in conjunction with the other prefetching mechanisms, on average 6% and at most 39%, for the cache-sensitive benchmarks studied. However, in a head-to-head comparison, a more conventional sequential hardware stream prefetcher outperforms runahead prefetching significantly, across nearly all applications. Software prefetching benefits are very inconsistent; when effective it appears to be extremely effective, demonstrating 67% performance improvement on milc, but unfortunately all other applications show negligible benefits. In future work, we hope to study the benefits of software prefetching in the XL compiler while disabling its mode of suppressing prefetches in the presence of a hardware prefetch engine; although this does not represent a realistic system configuration, we are interested to see software prefetch benefits relative to the other prefetching mechanisms.

The runahead prefetching results demonstrate that the feature is an effective form of prefetching for some applications when used in isolation or in conjunction with other types of prefetchers. Because its effectiveness is very application-specific, however, we do not view this technique as a replacement for conventional hardware prefetchers, or as a replacement for cache capacity. As shown in our evaluation, a sequential hardware prefetcher out-performs this particular implementation of runahead prefetching in nearly all applications. However, many (6 of the 19 cache intensive workloads evaluated) are not improved significantly by any of the forms of prefetching studied. Other implementations of runahead may overcome some of the limitations of the POWER6 implementation, but their ability to provide robust performance across a variety of workloads remains to be seen. Consequently, we view runahead execution as a useful addition, not a replacement, to large caches and stride-based hardware prefetching.

In conclusion, despite much improvement from these forms of prefetching, we find that ample opportunity remains for further gains from reducing stalls due to data cache misses. We believe that prefetching will continue to be a fertile ground for research, particularly the development of complexity, area, and bandwidth-effective prefetchers (including improved forms of runahead) for which adoption by industry is economical.

Acknowledgments

The authors would like to thank Hubertus Franke and Pratap Pattnaik for their support of this work. We appreciate the help of Joefon Jann in configuring various platform registers for controlling prefetching. Yaoqing Gao provided much insight into the use of software prefetching in the XL compiler suite.

References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 423–432, December 2003.

[2] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, pages 176–186, 1991.

[3] R. D. Barnes, S. Ryoo, and W.-m. W. Hwu. "flea-flicker" multipass pipelining: An alternative to the high-power out-of-order offense. In *Proceedings of the 38th International Symposium on Microarchitecture*, pages 319–330, December 2005.

[4] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in java. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 280–291, Washington, DC, USA, 2001. IEEE Computer Society.

[5] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Stealth prefetching. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274–282, October 2006.

[6] M. J. Charney and A. P. Reeves. Generalized correlation-based hardware prefetching. Technical Report EE-CEG-95-1, School of Electrical Engineering, Cornell University, 1995.

[7] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-performance throughput computing. *IEEE Micro*, 25(3):32–45, 2005.

[8] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Simultaneous speculative threading: a novel pipeline architecture implemented in Sun's Rock processor. In *Proceedings of the 36th International Symposium on Computer Architecture*, pages 484–495, June 2009.

[9] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 199–209, New York, NY, USA, 2002. ACM.

[10] J. D. Collins, H. Wang, D. M. Tullsen, C. J. Hughes, Y.-F. Lee, D. M. Lavery, and J. P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 14–25, 2001.

[11] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–290, 2002.

[12] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th International Conference on Supercomputing*, pages 68–75, 1997.

[13] Z. Hu, M. Martonosi, and S. Kaxiras. TCP: Tag correlating prefetchers. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture*, pages 317–326, 2003.

[14] T. Inagaki, T. Onodera, H. Komatsu, and T. Nakatani. Stride prefetching by dynamically inspecting objects. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 269–277, 2003.

[15] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 252–263, 1997.

[16] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, 1990.

[17] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 357–368, 1998.

[18] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. pages 144–154, 2001.

[19] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, 2007.

[20] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 222–233, New York, NY, USA, 1996. ACM.

[21] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 2003.

[22] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *Proceedings of the Tenth International Symposium on High-Performance Computer Architecture*, pages 96–105, 2004.

[23] D. G. Perez, G. Mouchard, and O. Temam. Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In *Proceedings of the 37th International Symposium on Microarchitecture*, 2004.

[24] A. Roth and G. S. Sohi. Speculative data-driven multithreading. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 37–, 2001.

[25] Y. Solihin, J. Torrellas, and J. Lee. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 171–182, 2002.

[26] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi. Spatio-temporal memory streaming. In *Proceedings of the 36th International Symposium on Computer Architecture*, pages 69–80, 2009.

[27] S. W. Son, M. Kandemir, M. Karakoy, and D. Chakrabarti. A compiler-directed data prefetching scheme for chip multiprocessors. In *Proceedings of the 14thSymposium on Principles and practice of parallel programming*, pages 209–218, 2009.

[28] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–119, 2004.

[29] A. Stoutchinin, J. N. Amaral, G. R. Gao, J. C. Dehnert, S. Jain, and A. Douillet. Speculative prefetching of induction pointers. In *Proceedings of the 10th International Conference on Compiler Construction*, pages 289–303, London, UK, 2001. Springer-Verlag.

[30] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, 2000.

[31] S. VanderWiel and D. Lilja. HPPC-96-05: A survey of data prefetching techniques. Technical report, University of Minnesota, 1996.

[32] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems. Guided region prefetching: a cooperative hardware/software approach. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 388–398, 2003.

[33] W. Zhang, B. Calder, and D. M. Tullsen. A self-repairing prefetcher in an event-driven dynamic optimization framework. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 50–64, 2006.

[34] C. B. Zilles and G. S. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 2–13, 2001.

Appendix

Benchmark	Compilation flags
SPECINT 2006	
400.perlbench	-bmaxdata:0x50000000 -qpdf1(pass 1) -qpdf2(pass 2) -O4 -qlargepage -qenablevmx -qvecnvol -D_ILS_MACROS -qalias=noansi -qfdpr -blpdata
401.bzip2	-bmaxdata:0x4ffffffc -qpdf1(pass 1) -qpdf2(pass2) -O5 -qlargepage -qenablevmx -qvecnvol -D_ILS_MACROS -qfdpr -blpdata
403.gcc	basepeak = yes
429.mcf	basepeak = yes
445.gobmk	-qpdf1(pass 1) -qpdf2(pass 2) -O4 -qlargepage -qenablevmx -qvecnvol -D_ILS_MACROS -blpdata
456.hmmer	-O5 -qlargepage -D_ILS_MACROS -qfdpr -blpdata
458.sjeng	-qpdf1(pass 1) -qpdf2(pass 2) -O5 -qlargepage -qenablevmx -qvecnvol -D_ILS_MACROS -qfdpr -blpdata
462.libquantum	-qpdf1(pass 1) -qpdf2(pass 2) -O5 -qlargepage -qenablevmx -qvecnvol -D_ILS_MACROS -q64 -qfdpr -blpdata
464.h264ref	-qpdf1(pass 1) -qpdf2(pass 2) -O5 -q64 -D_ILS_MACROS -qenablevmx -qvecnvol -qfdpr -bdatapsize:64K -bstackpsize:64K -btextpsize:64K
471.omnetpp	-bmaxdata:0x20000000 -qpdf1(pass 1) -qpdf2(pass 2) -O5 -qlargepage -qenablevmx -qvecnvol -D_ILS_MACROS -qalign=natural -qrtti=all -qinlglue -blpdata
473.astar	-bmaxdata:0x20000000 -qpdf1(pass 1) -qpdf2(pass 2) -O5 -qlargepage -D_ILS_MACROS -qfdpr -qinlglue -qalign=natural -blpdata
483.xalancbmk	-bmaxdata:0x20000000 -qpdf1(pass 1) -qpdf2(pass 2) -O5 -qlargepage -D_ILS_MACROS -qinlglue -D__IBM_FAST_VECTOR -blpdata
SPECFP 2006	
433.milc	-bmaxdata:0x40000000 -O5 -qlargepage -D_ILS_MACROS -qalign=natural -qfdpr -blpdata
470.lbm	-O5 -qlargepage -D_ILS_MACROS -qfdpr -q64 -blpdata
482.sphinx3	-qpdf1(pass 1) -qpdf2(pass 2) -O4 -qlargepage -qenablevmx -qvecnvol -D_ILS_MACROS -qfdpr -blpdata
444.namd	-qpdf1(pass 1) -qpdf2(pass 2) -O5 -D_ILS_MACROS
447.dealII	-bmaxdata:0x50000000 -O5 -qlargepage -D_ILS_MACROS -qrtti=all -D__IBM_FAST_VECTOR -blpdata
450.soplex	basepeak = yes
453.povray	-qpdf1(pass 1) -qpdf2(pass 2) -O5 -qlargepage -qenablevmx -qvecnvol -D_ILS_MACROS -qalign=natural -qfdpr -blpdata
410.bwaves	-bmaxdata:0x50000000 -O5 -qlargepage -qenablevmx -qvecnvol -qfdpr -qsmallstack=dynlenonheap -blpdata
416.gamess	-bmaxdata:0x40000000 -qpdf1(pass 1) -qpdf2(pass 2) -O5 -qalias=nostd
434.zeusmp	-bmaxdata:0x40000000 -qpdf1(pass 1) -qpdf2(pass 2) -O3 -qarch=auto -qtune=auto -qlargepage -qenablevmx -qvecnvol -qxlf90=nosignedzero -blpdata
437.leslie3d	-O4 -qlargepage -q64 -blpdata
459.GemsFDTD	basepeak = yes
465.tonto	-bmaxdata:0x20000000 -qpdf1(pass 1) -qpdf2(pass 2) -O5 -qlargepage -blpdata
435.gromacs	-qpdf1(pass 1) -qpdf2(pass 2) -O5 -qlargepage -qenablevmx -qvecnvol -qfdpr -D_ILS_MACROS -blpdata
436.cactusADM	-bmaxdata:0x60000000 -qpdf1(pass 1) -qpdf2(pass 2) -O2 -qarch=auto -qtune=auto -qlargepage -qenablevmx -qvecnvol -qfdpr -qnostrict -D_ILS_MACROS -blpdata
454.calculix	-qpdf1(pass 1) -qpdf2(pass 2) -O4 -qlargepage -D_ILS_MACROS -blpdata
481.wrf	-bmaxdata:0x30000000 -O5 -qlargepage -qalias=nostd -D_ILS_MACROS -blpdata
Java	
compiler, compress, derby, mpegaudio, SPECjbb2000	-Xmx1024m -Xlp