RC23633 (W0411-172) November 19, 2004
Computer Science

IBM Research Report

Exploiting Fine-Grained Memory Locality with Predictive Dispatch

Michael Gschwind, John-David Wellman
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Exploiting fine-grained memory locality with predictive dispatch

Abstract

Future process technologies promise to complicate and eventually break the scaling of traditional 6T SRAM cell memory arrays. In this work, we present an analysis of the microarchitectural impact, and propose possible solutions based on workload characteristics of a broad range of workloads (SPEC, transaction processing, desktop and media).

We identify the sensitivity of the workloads to both cache latency and bandwidth, and consider how different SRAM design choices will impact workload performance. We show that, while the SPECcpu benchmarks are primarily throughput dominated, transaction processing and many desktop workloads show a higher sensitivity to cache access latency. We propose a method for dynamically exploiting fine-grained memory locality by reusing data stored in sense amp latches to improve available memory bandwidth. Finally, we propose and evaluate a predictive dispatch strategy, to recover cache bandwidth in a latency-focused design with a instruction flush and re-issue recovery policy.

1 Introduction

As technology scaling becomes increasingly difficult, several new issues have become important. These issues affect not only the logic and circuit design, but also have impact on the microarchitecture. It is well known that process technology is not scaling uniformly – some technology aspects are scaling better than others. In particular, the interconnects (wires) are not scaling as well as the logic gates.

A second, and somewhat newer concern, is that for the current and future device sizes, the presence or absence of a few atoms in a deposition layer can make a significant difference in terms of the gate performance. This means that the manufacturing variability across the chip is becoming increasingly significant, particularly for circuits where carefully tuned or balanced devices are required. In this work, we will examine some of these impacts on the 6T SRAM cell, and explore possible microarchitectural solutions to deal with the potential performance loss and/or design limitations that they impose.

This work is primarily concerned with alleviating the negative performance impact on very high-frequency designs, especially in future technologies where these high core frequencies can greatly exceed the frequencies attainable by other elements of the system. As explained above, the memory hierarchy, especially the lower-level SRAM caches, are of particular concern.

Traditionally, the CPU market has been separated into two distinct markets with very different market economics. First is the consumer market segment, where product differentiation is based on (often software-controlled) functionality differentiation and the overall systems often have relatively lower performance. Parts in this market segment are typically not sorted, and to control costs all (or substantially all) working parts must typically ship in the same product offering, i.e. very high yields are required.

In the PC and workstation segment, differentiation is more generally based on performance running user-installed software, offering no room for software based device differentiation. Premium products are typically identified by higher performance. To achieve this performance segmentation and exploit the distribution of parts on the performance curve, frequency sorting is employed to offer a wide range of performance-differentiated parts.

These two markets typically also differ in choice of core microarchitecture, due to a combination of factors including NRE cost, time to market, and achievable yield. Where consumer electronics often rely on in-order cores, which represent the majority of shipped microprocessors worldwide, a significant portion of the PC and workstation market is served by out-of-order cores.

The last few years have seen a rise in demand for high-performance consumer electronics devices to power more advanced media processing applications and game consoles. This requires higher performance cores than have traditionally been available in the consumer market segment, yet still requires the cores to hold to stringent cost constraints. These cost constraints translate into requirements on chip area and yield, and prohibits area-intensive designs and/or complex structures which are difficult to design or to yield.

As a result, simpler in-order designs operating at high processor frequency are attractive for this market space. However, the emphasis here is on efficient designs which deliver high performance and good manufacturability. An example of such a design is the recently announced CELL Broadband Processor Architecture, which is based on multiple high-frequency in-order cores [ACG+00], [P+05], [Hof05], [GHF+05].

The contributions of this work are as follows: (1) we explore SRAM design issues and their impact on future microarchitecture, (2) we evaluate workloads for their latency and throughput sensitivity, (3) we propose a novel scheme for taking advantage of fine grain memory reference locality by exploiting data retention in sense amp latches, (4) we propose and explore same-line access predictors for microarchitectures with high upset recovery cost due to wire latency.

In this work, section 2 describes the impact of the near-future technology on SRAM arrays, and section 2.1 discusses two possible array implementations: one optimized for array throughput, and the other for access latency, and the performance implications of these non-pipelined array access (latency-optimized) versus pipelined array access using small sub-arrays (throughput-optimized) are considered in section 2.2.

Section 3 examines the microarchitectural issues surrounding the various near-future technology SRAM cache implementations, and section 4 proposes a microarchitectural method to merge accesses to the same cache line, in an effort to recover throughput within a latency-optimized implementation. Section 5 evaluates the performance of the Same-Line Access Merge technique for several implementations, we discuss related work in section 6, and section 7 presents our conclusions.

2 Impact of Technology Limitations on SRAM Array Design

The technology scaling issues outlined in the introduction impact 6T cell SRAMs in two main ways:

Non-uniform scaling increases the wire delays relative to the logic speed (among other effects). This can have a significant impact on the time it takes to access an SRAM cell within an array, as both word and bit lines represent a significant portion of the array access time.

Increased manufacturing variability makes it impossible to build minimally-sized 6T SRAM cells at very high frequencies. Because the 6T SRAM cell relies on balanced (or matched) devices to reliably hold data, the minimally-sized 6T cell, at current and near-future scales, requires fabrication depositions accurate potentially to the same number of atoms in a layer. This is currently beyond the capabilities of large-scale production fabs.

Because random dopant fluctuations make manufacture of reliable, minimally-sized 6T SRAM cell in near-future technology extremely difficult, a number of solutions have been considered [CFH+05]. One option is to increase the size of the matched transistors used for building the basic 6T SRAM cell, which increases the number of atoms in the device layers, and thereby reduces the impact of slight variations in the deposition during the manufacturing process. Another option is to use a slower clock rate, or a more stable cell design (e.g., one using more than six transistors per memory cell).

Note that both the impact of the different scaling rate for wires, and the increased impact of manufacturing variability serve to reduce the effective frequency at which an SRAM memory array can be accessed. These effects point towards a likely increase in the time needed to access the SRAM array relative to the processor clock rate, very possibly to the point where the array access time will exceed the processor clock rate, and thus the latency of a single pipeline stage delay.

2.1 Possible Solutions

To maintain overall processor cycle time, several solutions are possible from a cache array design perspective:

- Allocate the equivalent of two pipeline stage delays (i.e., two processor cycles) to the SRAM array access. This will reduce the array access throughput to one access every other cycle, as the actual array access will be non-pipelined.
- Organize the SRAM array as a large collection of small sub-arrays, each of which can be accessed in one cycle, and select from the outputs of these many subarrays using pipelined multiplexing logic. This solution will recover fully-pipelined access to the SRAM array, but will reduce the array density, as each subarray will need to duplicate some logic, including address decoding logic, drivers, and sense amps, and the overall array will include additional routing and multiplexing logic, plus any pipeline registers within the pipelined multiplexor logic. This density reduction may in turn force longer wiring, requiring even deeper pipelining to overcome that additional delay. For a given area budget, this will also reduce the size of the memory that can be supported.

2.2 Performance Implications

Either solution can degrade the performance of some workloads. In figure 1, we show the impact of either increasing first level

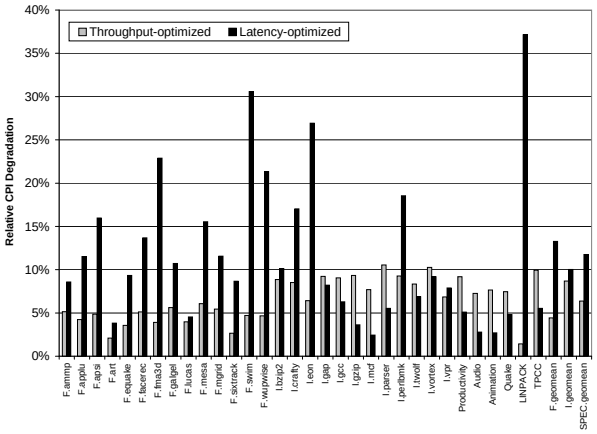

Figure 1: Memory access latency vs bandwidth tradeoffs for first level data cache hierarchy

cache access latency (i.e. optimized for throughput), or alternatively reducing the throughput of the array to every-other cycle (i.e. optimized for memory access latency) by plotting the percent CPI degradation for several SPEC workloads, a TPC-C benchmark version, several other applications, and the geometric means for SPECint2000, SPECfpu200 and the entire SPEC-cpu2000 benchmark suite. The other workloads of figure 1 represent some additional areas of interest, including productivity desktop applications, multimedia applications, the Quake open-source game, and the LINPACK high performance computation benchmarks.

As figure 1 shows, a number of latency-sensitive applications, such as many SPECint applications, transaction processing and some of the desktop applications show a significant degradation with increased cache latency (i.e. when the latency of cache access is increased by one cycle, even with full pipelining of cache accesses). Conversely, throughput-oriented workloads, such as SPECfp, show significant performance degradation when memory access bandwidth is reduced (i.e. when the number of cycles to access the cache is not increased, but the cache is reduced from full pipelining to every-other-cycle access).

The data demonstrates that the TPC-C benchmark is much more latency sensitive, which is understandable from the frequent, often exposed dependence chains which would be directly impacted by the increased latency of the fully pipelined array. In contrast, the SPECfp2000 program binaries exhibit more parallelism, often having independent memory operations scheduled back-to-back, and longer load-to-use distances to take advantage of the increased throughput and more robust performance in the face of increased cache access latencies.

Interestingly, the latency sensitive applications in figure 1 prefer to suffer a 2x reduction in memory bandwidth for a one-cycle latency advantage (effectively 14% of the overall LSU latency including address generation, array access, etc.). This seems to indicate that some applications are very sensitive to the memory access latency.

3 Microarchitectural Implications of Array Design

As shown in section 2.2, there are some potentially large performance degradations possible with the selection of either a degraded-throughput or an increased latency cache memory array. A number of approaches are possible to solve the issues related to the tradeoffs between degraded throughput and increased array latency.

Choosing a processor implementation with deeply pipelined, longer latency arrays, Chishti and Vijaykumar [CV04] evaluate the use of simultaneous multithreading to turn latency-bound problems into throughput bound problems with regard to the register file latencies. A similar approach can be used to resolve data cache array latency. This, however, requires workloads that can efficiently exploit the multi-threaded processing hardware effectively, and thus is little help when a single execution thread is targeted.

Alternatively, when a different application segment is targeted as the primary optimization point, a less deeply pipelined array with reduced throughput can employ techniques to recover throughput via microarchitectural means. One known approach used in memory subsystems is the use of banking, where back to back accesses can proceed when referring to different banks.

3.1 Exploiting Access Bursts

In this work, we propose and evaluate a new microarchitectural approach to exploit fine grained memory locality to recover some memory bandwidth in a non-pipelined array design. The pipeline representation of Figure 2 illustrates a load/store unit in which the caches can be accessed in a fully-pipelined manner, i.e. the cache tags array, memory cell array, etc. can all be accessed at the same frequency as the core processor operates.

The pipeline illustration of Figure 3 represents a load/store unit in which the core processor frequency is double the access rate for important cache units, e.g., the cache array. In a design like that of Figure 3, the load/store units must be restricted to prevent back-to-back accesses (i.e. one cache access per cycle) as this

Figure 2: A basic, fully-pipelined LSU design

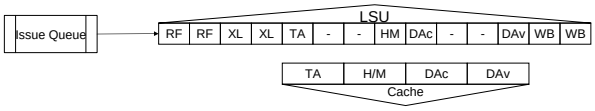

Figure 3: A basic non-fully-pipelined LSU design

would exceed the ability of the cache to serve those requests. This therefore represents a latency-optimized design (giving up some throughput) with a high-density memory array.

Our approach to recover throughput in such a latency-optimized high-density memory array is based on exploiting program access patterns that show burst accesses to successive memory locations. Such burst-access behavior is not uncommon in applications, including accesses to different fields in the same data structure, function call and return, block copy, and unrolled loops.

When such codes are executed, successive memory access instructions usually refer to data words in close proximity to one another, most notably often in the same cache line. To size this opportunity in some existing workloads, Figure 4 depicts a histogram of memory access sequences to the same 128-byte cache line. In this example, the cache line size of 128 bytes was chosen as this is a typical current cache line size, e.g., the cache lines in the IBM Power4^{TM} family of processors include 128 bytes of data [TDF^{+}02].

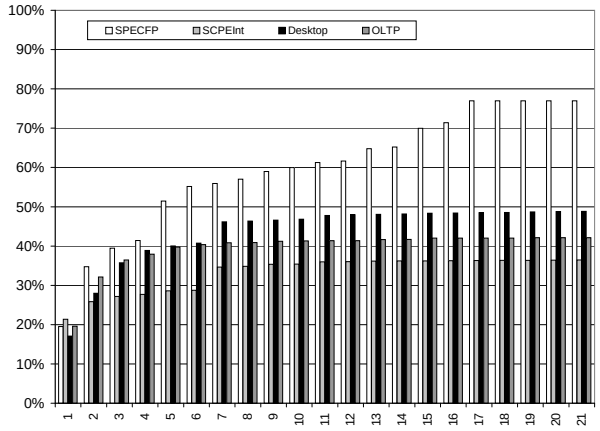

Figure 4: Accumulated percentage of loads to the same cache line by successive access string length

In this figure, the percentage of all loads that are members of a string of (i.e. a sequence of successive) loads that access the same cache line is plotted by accumulating the total opportunity. Thus, the first group of bars at x-axis position 1 indicate the percentage of loads in the execution trace that access the same cache line as the immediately preceding load, and which are immediately followed by a load to a different cache line. Figure 4 shows that approximately 20% of the loads fall into this category.

The next group of bars adds to this the percentage of loads that are members of 3-load sequence of accesses to the same cache line, which brings the accumulated percentage of loads close to 30%. As one looks across the width of the plot in figure 4, the height of the bars therefore indicates the total opportunity, were the hardware capable of exploiting same-line access merging for that number of successive loads. Looking to the extreme right of the graph, one finds that the total opportunity to exploit locality of reference to a cache line is fairly significant, falling between 35% and 77% depending on the workload set. While this opportunity analysis is fairly simplified, i.e. it does not take into account the actually timing (i.e. cycles of separation in execution) of the loads, nor memory coherence actions, etc. it does show that there is real opportunity in some workloads to exploit same cache line access. Note also that these workloads were not compiled and scheduled with this kind of feature in mind – these are existing workloads taken "as-is" and thus the potential may be greater if the compiler is made aware of this opportunity in the hardware.

In order for the hardware to exploit this locality, accesses must be identified which refer to the same cache line as a previous memory operation; in the most restricted case, accesses which refer to the same-line as an access issued in the immediately previous cycle. If such proximity can be established, then a memory load instruction following another memory load instruction from the same cache line does not have to perform a new data array access. Instead, a new set of bytes can be selected from the data recovered by the previous access, which can be retained in the sense amp latches.

Implementing such an approach, in a core that effectively requires multiple cycles for the cache array access, allows the recovery of back-to-back memory access operations whenever the opportunity presents itself. Specifically, because a cache generally stores data in a cache line, which is usually larger than the requested datum size (e.g., the cache lines in an IBM Power4^{TM} family of processors include 128 bytes of data, whereas the largest single-item request size in PowerPC^{TM} is only 16 bytes), the cache array will actually access some amount of data surrounding the requested datum [TDF^{+}02].

3.1.1 Base Cache Elements

Figure 5 illustrates a somewhat abstracted cache design that still illustrates the important elements of a cache implementation, and identifies some of the key components of the cache.

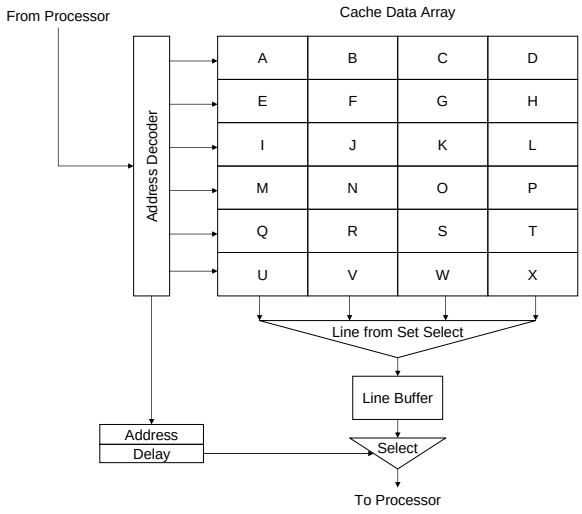

Figure 5: A general illustration of the significant elements of a cache array.

Looking at figure 5 one notes that the cache is accessed with a request address. This address is used to select a unique row of the cache array, where each row of the cache array can constitute multiple distinct cache lines (i.e., a set of cache lines, or congruence class) in which the datum is equally likely to reside. The tags of each of the cache lines of the set (corresponding to the selected row of the array) are checked to determine which cache line holds the requested datum (or whether none of the cache lines currently in the cache hold the datum, which results in a cache miss). Assuming there is a cache hit, the proper cache line is brought into a cache line buffer (i.e., the array's data cells are used to drive appropriate amplifier circuits, whose output is thereafter captured in the state-saving cache line buffer) from which the proper, requested datum is selected (by multiplexing) and forwarded to the processor.

In the case where the cache can be accessed once per processor cycle, figure 6 shows the optimal throughput flow of instructions through the LSU pipeline. For each cycle, starting at cycle *T0* and progressing through cycle *T16*, the instruction currently occupying each of the LSU pipeline stages is shown. The load/store unit pipeline stages are labeled in the boxes across the top of figure 6, where the short-hand notations correspond as follows:

RF is register file access,

		LSU					
Cycle	IssueQ	RF	XL	TA	HM	DAc	DAv	WB
T0	A	-	-	-	-	-	-	-
T0+1	B	A	-	-	-	-	-	-
T0+2	C	B	A	-	-	-	-	-
T0+3	D	C	B	A	-	-	-	-
T0+4	E	D	C	B	A	-	-	-
T0+5	F	E	D	C	B	A	-	-
T0+6	G	F	E	D	C	B	A	-
T0+7	H	G	F	E	D	C	B	A
T0+8	I	H	G	F	E	D	C	B
T0+9	J	I	H	G	F	E	D	C
T0+10	K	J	I	H	G	F	E	D
T0+11	L	K	J	I	H	G	F	E
T0+12	M	L	K	J	I	H	G	F
T0+13	N	M	L	K	J	I	H	G
T0+14	O	N	M	L	K	J	I	H
T0+15	P	O	N	M	L	K	J	I
T0+16	Q	P	O	N	M	L	K	J

Figure 6: Best-case throughput for a core-rate accessible cache

XL is address translation,
TA is the tags access
HM is the hit/miss notification
DAc is the data array access
DAv is the data available at the processor (which includes formatting, etc.)
WB is the data writeback by the LSU

The issue queue (labeled IssueQ), which is not a part of the load/store unit, is included for illustrative purposes, and here is assumed (for space in the figure) to be limited to contain no more than four instructions on any given cycle. Each of the letters in figure 6 represents a unique load instruction being executed in the load/store pipeline, and represents the position of that load operation on the given cycle, where the advance of time is from the top to the bottom. This figure shows an optimal throughput because it does not include any hazards in the operation of the loads – there are no misses, address translation exceptions, or data alignment exceptions shown in the pipeline flow.

Note that in figure 6 and figure 7 that the address translation (**XL**) and tag access (**TA**) are illustrated as successive actions performed sequentially. There are cache design which combine or overlap these actions by utilizing virtually-indexed, physically-tagged caches, which access the tag access and the array using the untranslated virtual address, and then compare the access tags using the translated physical address. Use of such a cache, which overlaps these fully pipelined stages, would not significantly alter any of the findings in this work.

3.1.2 Throughput-Limited Cache

Figure 7 illustrates a similar optimal throughput case where the core processor pipeline operates at a finer granularity (i.e., the core is "superpipelined") to provide a faster processor clock rate. If the cache is unable to accept load accesses at the full processor clock rate, e.g. the cache timings are as illustrated in the example of Figure 3, then the issue queue must issue load instructions to the load/store pipeline only every other cycle.

		LSU													
Cycle	IssueQ	RF	RF	XL	XL	TA	-	-	HM	DAc	-	-	DAv	WB	WB
T0	A	-	-	-	-	-	-	-	-	-	-	-	-	-	-
T0+1	B	A	-	-	-	-	-	-	-	-	-	-	-	-	-
T0+2	CB	-	A	-	-	-	-	-	-	-	-	-	-	-	-
T0+3	DC	B	-	A	-	-	-	-	-	-	-	-	-	-	-
T0+4	EDC	-	B	-	A	-	-	-	-	-	-	-	-	-	-
T0+5	FED	C	-	B	-	A	-	-	-	-	-	-	-	-	-
T0+6	GFED	-	C	-	B	-	A	-	-	-	-	-	-	-	-
T0+7	HGFE	D	-	C	-	B	-	A	-	-	-	-	-	-	-
T0+8	HGFE	-	D	-	C	-	B	-	A	-	-	-	-	-	-
T0+9	IHGF	E	-	D	-	C	-	B	-	A	-	-	-	-	-
T0+10	IHGF	-	E	-	D	-	C	-	B	-	A	-	-	-	-
T0+11	JIHG	F	-	E	-	D	-	C	-	B	-	A	-	-	-
T0+12	JIHG	-	F	-	E	-	D	-	C	-	B	-	A	-	-
T0+13	KJIH	G	-	F	-	E	-	D	-	C	-	B	-	A	-
T0+14	KJIH	-	G	-	F	-	E	-	D	-	C	-	B	-	A
T0+15	LKJI	H	-	G	-	F	-	E	-	D	-	C	-	B	-
T0+16	LKJI	-	H	-	G	-	F	-	E	-	D	-	C	-	B

Figure 7: Best-case throughput for half-core-rate accessible cache

Comparing Figures 6 and 7 one can see the prodigious drop-off in the optimal throughput, at least in terms of loads per processor cycle. In fact, the throughput per unit wall-clock time may be identical: in this case, figure 6 shows that load "A" is written back in cycle *T0+7*, whereas in figure 7 instruction A is not written back until cycle *T0+14*, but note that the cycle time of the design for figure 7 is assumed to be twice that of figure 6, resulting in identical elapsed time to execute the load operation.

3.1.3 Recovering Throughput

Our solution to this problem, where cache accesses cannot be pipelined at the one per processor cycle, is to take advantage of the additional data available within a cache line – effectively to exploit the spatial locality of temporally near-by cache accesses. Considering the general cache structures of Figure 5, one notes that on each cache access, the cache stores, in a cache line buffer, an entire cache line, from which the multiplexing elements are used to select the requested datum which (alone) is forwarded to the processor. In this work, we use an access granularity of an entire cache line as our baseline.

Depending on cache array design choices made, a number of bit lines may share a single sense amplifier, or a line may consist of several sectors sharing a bit line, limiting the data retrieved by a single cache access to a subline. If such an implementation is chosen, the subline or sector would be the granularity at which data accesses can be shared. We have experimented with a number of subline access sizes, which have confirmed the trend described here. While shorter access sizes reduce the scope of locality to a smaller data area, the approach explored here concentrates on accesses in extremely close proximity (in back to back cycles), and tends to refer to data in close proximity, e.g., sequential accesses. In this work, we concentrate on our results obtained with 128B cache line accesses for brevity.

If a subsequent load access to the cache were made to some other datum *within this same cache line*, then no additional accesses to the cache data array would be necessary. Instead, such an access could be served (assuming that the requested datum is completely contained within the cache line) simply by altering the settings on the datum selection multiplexers and forwarding the new data to the processor. This effectively reduces the required number of cache data array accesses per requested load from one to some value less than one, dependent upon the workload and code schedule.

In its simplest form, same-line access behavior of successive memory instructions can be exploited by issuing a load instruction every cycle. When a memory access instruction immediately follows another memory instruction issued in the preceding cycle, a test is performed to identify if the two accesses refer to the same cache line. If this is the case, the load instruction can be issued with the array access suppressed; only the computation of byte select masks and appropriate shift and sign-extension controls will be performed, and bypassed to the data formatting stage. When the load instruction for which such a same-line access merge has been performed reaches the data formatting stage, a set of bytes will be retrieved from the sense amplifier latches using the newly computed byte selects and data formatting controls, and then retired to the register file.

Figure 8 shows the performance of exploiting fine grained memory locality by testing for opportunities to perform a same line access merge after address generation. If addresses refer to the same-line, instructions will be issued back to back; if addresses refer to distinct lines, then an issue stall cycle will be introduced. Because of the low penalty for wrongly merging loads (a one-cycle stall in the best case), there is relatively little need to provide any intelligence in the issue of the loads. Note that there are some situations where additional intelligence could be gainfully employed (e.g., in an out-of-order issue design, the issue order of loads could be influenced by the likelihood for their

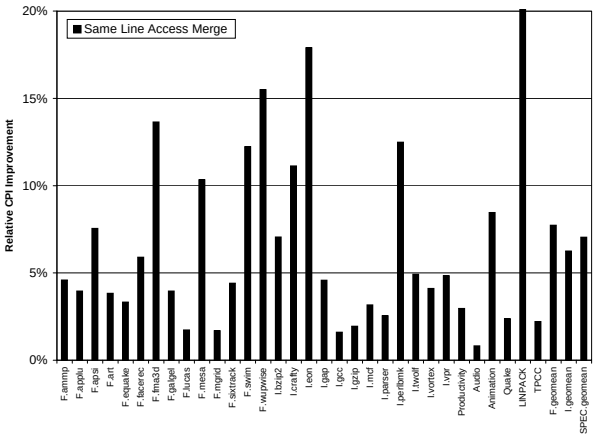

Figure 8: Performance based on aggressive same-line access merging after address generation phase

combining).

We also note that figure 4 shows good locality of reference in data access patterns for SPECfp, but the performance advantage obtained by the SPECfp workloads is only slightly better than that of SPECint workloads. While figure 4 only considered strings of references regardless of relative instruction distance, figure 8 is based on simulation using specific issue rules and policies of an assumed processor implementation. Savings only accrue for back to back accesses referring to the same line, limiting the benefit of the longer same-line reference strings found in SPECfp.

4 Same-Line Access Merging

While the scheme outlined so far shows high effectiveness, performing instantaneous stalls at the processor, or even function unit or pipeline level, is becoming increasingly difficult due to the increasing importance of wire length delays outline above.

While the scheme outlined in section 3.1.3 shows high effectiveness, it requires the ability to introduce stalls throughout the processor execution resources after the load/store unit performs the address generation, propagating the stall signal throughout the processor within a single cycle. In current and future high-frequency designs, particularly with the relatively poorer scaling of wires, performing instantaneous stalls at the processor, or even function unit or pipeline level, is becoming increasingly difficult.

Microarchitects, therefore, will likely adopt a scheme that does not stall, but instead aborts a problematic load instruction, and re-injects it into the pipeline at a later time. This re-injection may occur from either the instruction cache [Gsc99], or the issue queue [GKA01]. Unfortunately, the performance lost in repeatedly cancelling and reissuing instructions that the issue logic has overly aggressively scheduled together can result in a significant performance penalty, due to the high recovery (re-injection) cost, as shown in figure 9.

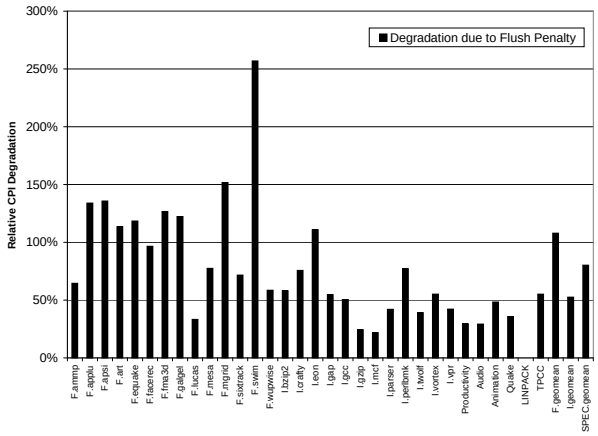

Figure 9: Aggressively issuing memory operations back-to-back and flushing prematurely issued instructions causes significant performance degradation across all workloads.

Thus, while in a stalling microarchitecture, all loads could be issued back-to-back in the hope that some successor loads will be to the previously accessed cache line, a more sophisticated issue scheme is required for those designs that suffer a higher penalty when loads are incorrectly merged. The exact cost (in number of cycles) of the penalty is obviously implementation-specific, but there are really two main options:

- assuming the issue queue retains issued loads for some number of cycles, the second load instruction (and all successors) can be flushed, and the load reintroduced to the LSU from the issue queue, or,
- assuming that the issue queue does not retain the issued loads, then the load instruction (and all successors) must be flushed and re-introduced from some other structure, e.g., from an instruction buffer (should one retain the instructions sufficiently throughout their execution) or by being re-fetched into the processor.

The penalty for an incorrect issue is clearly greater should the processor need to re-fetch the load and successor instructions than were it to recirculate from the issue queue(s), but there is still a significant penalty in either case. In order to keep the overall cost of speculatively issuing (merging) loads, in expectation

that they will hit the same cache line as their predecessor load, we introduce a prediction mechanism, which is used within the issue queue logic, to guide the decision whether to inject a successor load immediately after a predecessor load, or to wait an additional cycle (at which time it need not combine with the predecessor load, but can access the cache normally).

Figure 10 illustrates an optimized flow of loads through an LSU pipeline where loads that can merge are issued back-to-back. In Figure 10, the *"Comb?"* column indicates the predictor's output with regard to whether a successor load is predicted to be combinable with a predecessor (in the figure, an entry like "AB" indicates that A and B can combine). In this illustration, the predictor is assumed to have always made a correct prediction, producing the maximum throughput for this sequence of loads, and this (in-order) issue policy.

			LSU													
Cycle	Comb?	IssueQ	RF	RF	XL	XL	TA	-	-	HM	DAc	-	-	DAv	WB	WB
T0	-	A	-	-	-	-	-	-	-	-	-	-	-	-	-	-
T0+1	AB	B	A	-	-	-	-	-	-	-	-	-	-	-	-	-
T0+2	-	C	B	A	-	-	-	-	-	-	-	-	-	-	-	-
T0+3	-	DC	-	B	A	-	-	-	-	-	-	-	-	-	-	-
T0+4	CD	ED	C	-	B	A	-	-	-	-	-	-	-	-	-	-
T0+5	DE	FE	D	C	-	B	A	-	-	-	-	-	-	-	-	-
T0+6	-	GF	E	D	C	-	B	A	-	-	-	-	-	-	-	-
T0+7	-	HGF	-	E	D	C	-	B	A	-	-	-	-	-	-	-
T0+8	-	IHGF	F	-	E	D	C	-	B	A	-	-	-	-	-	-
T0+9	-	JIHG	-	F	-	E	D	C	-	B	A	-	-	-	-	-
T0+10	-	KJIH	G	-	F	-	E	D	C	-	B	A	-	-	-	-
T0+11	GH	LKJI	H	G	-	F	-	E	D	C	-	B	A	-	-	-
T0+12	HI	MLKJ	I	H	G	-	F	-	E	D	C	-	B	A	-	-
T0+13	-	MLKJ	-	I	H	G	-	F	-	E	D	C	-	B	A	-
T0+14	-	NMLK	J	-	I	H	G	-	F	-	E	D	C	-	B	A
T0+15	-	NMLK	-	J	-	I	H	G	-	F	-	E	D	C	-	B
T0+16	-	ONML	K	-	J	-	I	H	G	-	F	-	E	D	C	-

Figure 10: Best-case throughput for half-core-rate accessible cache with SLAM prediction.

Looking at Figure 10, one notes that the number of empty pipe stages (those signified by "–") is significantly reduced in the case where several successor loads can be combined, and thus issued back-to-back. This illustrates a potential performance benefit which could be exploited in a system which supports this concept of combining loads.

4.1 Implementation

The implementation of same-line access merging involves three functions:

- an enhanced issue function (for the LSU) that decides when two accesses are expected to be to the same cache line, e.g. as shown in Figure 11
- a line buffer from which a second access to the same cache line can be recovered (without access to the main cache array), e.g. as in Figure 12
- hardware to verify that the accesses were to the same line, and to trigger recovery if they are found to be to different lines.

In general, these elements are similar to other prediction hardware implemented in modern microprocessors, e.g. branch predictors, etc.

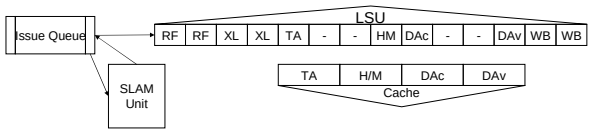

Figure 11: A SLAM-enabled non-fully-pipelined LSU designs

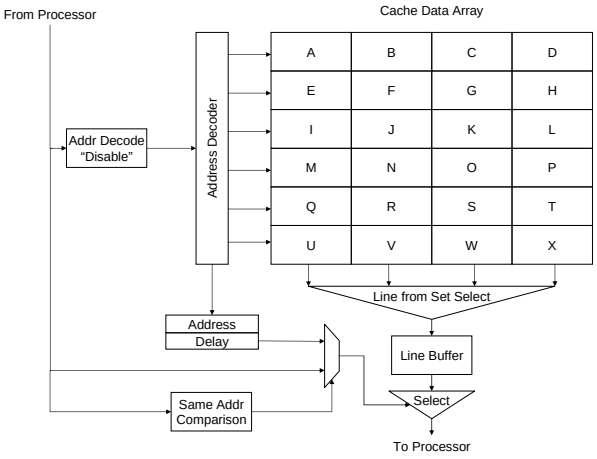

Figure 12: A general illustration of a SLAM-enabled cache array.

The same-line access predictor, which is used to supplement the issue queue logic and predict when two successive loads are expected to access the same cache line, can implement any one of a variety of prediction schemes, including n-bit saturating counters, local and/or global history based predictors, and so forth. Such a predictor can also be accessed in a number of different ways, including:

- using the instruction address of the candidate for same-line access,
- using the instruction addresses of two candidates for combining (i.e., the previously-issued and the next issue candidate)
- using instruction-word information (e.g. the base register

number)

- using a combination of the instruction address and instruction word information, from either or both of the candidates,

among many other possible schemes. In this analysis, we primarily concentrate on simple prediction mechanisms, in an effort to keep the overall hardware costs as low as possible.

The validation mechanism, used to determine whether two back-to-back accesses are actually to the same line, simply requires that the previous access address and the current instruction's access address map to the same cache line. This comparison must be made after the address of the later access instruction has been computed (i.e., after register-file access and any addition required to generate the address), which could readily be served by retaining the address of the prior access in a "same-address validation" register for comparison to the address of the successor access. In some cases, these comparisons may need to be delayed until the second access address is translated (if the address comparisons are made using the physical or real addresses). In the case that back-to-back accesses are not to the same line, an exceptional condition must be indicated and properly handled. Generally, this would employ the same kind of hardware as other exceptional conditions in the load/store pipeline, e.g. the hardware to handle address misalignment, etc.

A determination whether two accesses refer in fact to the same line can be made at earliest after the address generation phase. A single register stores the effective address of the previous cycle's array access. When a memory request has been issued to extract alternate bytes from the same line, an address check is performed at this point. If the address check succeeds, the extraction and format control is generated, and forwarded using the same pipeline registers as for a normal array access. However, the array access *per se* is suppressed, and the pipeline register after the sense amps do not record a newly sensed data line. Instead, the previously formed control word is used to extract a second set of data from the cache line.

If the address check fails after the address generation phase, a mis-prediction is indicated. In this case, the over-aggressively issued instruction, as well as all of its successor instructions are immediately flushed, and subsequently must be reissued by the frontend. This can be accomplished either by refetching the data from the cache, by retaining speculatively issued instructions in the issue queue (which requires an in-order issue queue which can issue from locations other than the front ranks), or by using a dedicated recirculation buffer or re-issue queue as described in [GKA01].

Using the effective address to determine whether two accesses refer to the same line ignores the – infrequent – possibility of having two effective addresses on different pages refer to the same physical line, as this requires translation by the effective to real address table (ERAT).[1] However, this is a conservative assumption, and allows recovery to be started several cycles earlier.

While retaining data in the sense amp latches creates caching effects, no special coherence mechanisms must be implemented[2]. Since no memory operation to another address can intervene between two same-line merged memory accesses (as this would invalidate the data stored in the sense amp latches), updates to the global memory space are not observable by a program. Instead, the execution behavior will always resemble that of the second load having been performed before any potential remote memory update. Because no intervening load can establish any global time ordering, this mechanism is sequentially consistent [Lam79, AG95].

In our design point, we simulate a single issue load/store unit, and the same line merge functionality is only engaged when the array is unavailable during a second cycle in which a load access is predicted to the same line as an immediately preceding (prior cycle) load operation, thereby eliminating the danger of issuing a store between two successive loads. This approach might be extended to retaining data for a larger number of cycles, either to reduce array access activity, and thereby cut the power consumption, or because an array may only be available one out of a larger number of cycles. If so extended, more synchronization must be provided.

Generally, more elaborate cache coherence policies can also be applied to the line buffer and SLAM predictor, but the implementation cost might exceed the marginal performance benefit. When synchronizing instructions are detected, which would cause the line buffer to be invalidated an implementation may choose to modify successive predictions based on the knowledge that the line buffer has been invalidated until another memory which would fetch data to the line buffer is encountered. [3]

5 Experiments and Evaluation

To evaluate the effectiveness of same-line access prediction, we have modeled the scheme with our advanced, modular PowerPC microprocessor simulation environment. The environment mod-

[1] The ERAT is a merged segment and page table lookaside buffer in PowerPC implementations.

[2] In some architectures, memory barrier instructions should not be issued in the same cycle as a predicted same-line access as they might establish a global event ordering.

[3] If the core implements bypassing from a store, then the invalidate needs to occur only when the entry is retired from the store buffer. Evidently, an implementation could also decide to update the line buffer, but the cost would likely be significant in terms of logic complexity.

els a deeply pipelined in-order PowerPC processor design with a stall-free backend. Instructions are issued speculatively to intercept the result from the producing instruction when they can be bypassed at the earliest point in time. When a result cannot be produced at the predicted time, e.g. because a cache miss has occurred, the dependent instruction and all its successor instructions are flushed, and they must be re-fetched and re-issued [Gsc99]. This is the most conservative implementation (from a CPI degradation perspective), so other recirculation schemes may offer better performance than the data shown in this evaluation.

For these experiments, we configure the latencies of the different pipeline stages as shown in table 1. We evaluate two basic machine configurations which fetch, decode and dispatch in-order, up to 4 instructions per cycle. We have explored two base machine configurations, varying the number of execution resources to establish the impact of available parallelism on the the effectiveness of this scheme. We have found that relative CPI impact of this scheme tracks closely for the machines studied. Here we will concentrate on a base machine configuration implementing one branch, two fixed point, two floating point, one load/store and one VMX unit.

Both the baseline cores includes separate instruction and data caches, each 32 Kbytes in size with a 128-byte linesize. The instruction cache is 2-way set associative, while the data cache 4-way set associative. Instruction execution is performed in-order, with a stall-free back end microarchitecture. Instructions which have been issued speculatively, but cannot proceed, must be flushed, re-fetched and re-issued.

Unit	Latency (Cycles)
fetch	3
decode/dispatch	5
register file access	1
FXU	1
LSU	2 + array access
FPU	6

Table 1: Base latencies of the simulated processor

To assess the impact of different design options, we have explored the respective performance impact of reduced data cache throughput and longer latency data cache accesses. Given the constraints on array design, the choices are to pipeline the actual array access to allow one access per cycle, at the cost of longer latency, or to have an unpipelined long-latency array access, which restricts successive accesses to the array to every-other cycle.

We have also analyzed the impact of having to reduce the cache size to meet a given area budget when going to a smaller sub-array size. Given a 4-way set associative cache, the smallest area reduction possible is to move to a 3-way set associative design, reducing the area to 3/4 size of the original area.

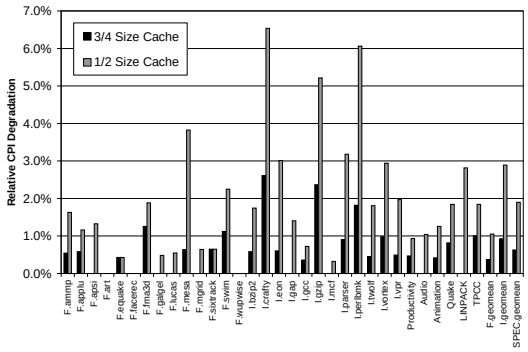

Figure 13: Performance degradation from reducing cache size to maintain area of a lower-density implementation

Figure 13 shows the performance degradation suffered by the core when the reduced density of the throughput-optimized cache design requires a reduction in the total cache storage (to maintain a similar area footprint). The impact from reducing the cache to 3/4 the original size, by reducing from a 32 Kbyte data cache to a 24 Kbyte cache, is not particularly large, peaking at about 2.5%. In this case, the reduction was implemented simply by stripping one of the set entries for each set, i.e., by moving from 4-way associative to 3-way set associative. We also considered the common technique of reducing the cache area to half the original version, in order to maintain nice power-of-two relationships (i.e., the new cache is 16 Kbyte with 2-way set associativity).

While these relative CPI degradations may seem small at first glance, they are consistent with other analyses of these kinds of cores. It is important to note that the base processor design is not a low-CPI out-of-order processor, but a relatively higher CPI in-order design built to exploit high frequencies (rather than ILP) for greater overall performance. Thus, while the relative CPI degradations may seem smaller than for an out-of-order design, the absolute increase in number of cycles of execution is not necessarily smaller, as the higher baseline CPI results in a smaller relative CPI change for the same number of cycles impact. Furthermore, considering the workloads, the SPEC CPU2000 programs are not particularly cache or memory intensive, and other applications which are memory intensive, e.g., TPC-C, are gen-

erally much more influenced by the L2 cache than the L1 size.

Given the potential for exploiting locality, but the inability to derive address information, we have turned to predictive techniques to establish a set of memory operations which are likely to be legally issued back to back.

Our evaluations have included local 1-bit and 2-bit saturating counter predictors, to explore the effect of hysteresis. To determine the opportunity for each benchmark, we also use an oracle prediction based on the observed addresses. An initial analysis shows that a 1-bit predictor leads to performance degradation in most cases, with only a minor overall performance improvement using the 2-bit predictor. This can be traced to a large number of mispredictions, as illustrated by a closer examination of the detailed simulation statistics.

A closer analysis for the *mesa* benchmark shows that the 1-bit predictor correctly predicted 7.06M instructions as eligible to be satisfied from the line accessed by the immediately preceding memory operation. In 1.78M cases, the predictor did not recommend merging the accesses when they could have been satisfied from the line buffered in the sense amp latches. This conservatism can be attributed to a combination of cold-start ("learning") phase for the predictor, and natural operation following a misprediction. A total of 1.1M accesses were incorrectly (overaggressively) predicted to be candidates for same-line access merge, but subsequently accessed a different cache line. This results in a respectable 86.5% success rate in correctly predicting that a load can be satisfied from the sense amp latches. Note, however, that the cost model is highly skewed against mispredictions, as a correct prediction saves approximately one cycle (depending on the schedule, etc.), but a misprediction incurs a full pipeline flush and re-fill.

Specifically, for *mesa*, the 1-bit predictor incurred an overall total performance penalty of 720000 cycles over the full 100M instruction execution segment, despite a relatively successful 86.5% correct prediction rate. The 86.5% correct accesses saved 11.7M cycles. This comes to an actual saving of 1.56 cycles per back to back access. This is accounted for by the ability to eliminate some issue group splits, and queuing effects which can either hide or magnify the cost of such a prediction. The cost for overly aggressively predicting 1.04M same-line accesses which do not in fact refer to the same cache line, however, incurred a penalty of 12.3M cycles, which is a penalty of 11.25 cycles per wrong prediction.

By adding hysteresis, the 2-bit predictor makes fewer of these overly eager mispredictions, avoiding some fraction of the expensive recovery penalties. Unfortunately, simply moving to a 2-bit predictor does not provide a very large boost in the performance, as it still applies merge predictions too aggressively overall. Based on the high asymmetry of the cost model, we decided to experiment with predictors which are skewed to reflect the asymmetry in the relative costs of decisions. To this effect we experimented with a 2-bit saturating counter predictor which has 3 non-merge predicting states, and only a single prediction state, as shown in figure 14.

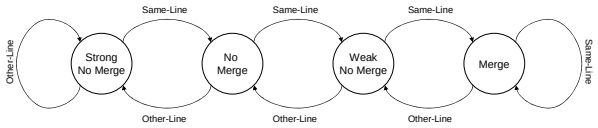

Figure 14: Asymmetric predictor FSM

This predictor is based on the observation that a correct prediction will improve performance much less than an incorrect prediction will degrade performance, and thus require at least two observations of valid pairings before making a positive prediction. By skewing the interpretation of the 2-bit counter states, the predictor will be considerably more conservative in its predictions, making fewer merge predictions overall, but most noticeably reducing the mispredictions (which suffer ten times in penalty the cycles that a correct prediction saves).

The prediction array is indexed by low order bits of the instruction address in parallel with instruction fetch. The prediction is used when making issue decisions to determine whether instructions should be issued directly after a memory operation issued in a preceding cycle, or to stall the memory operation for one cycle. If the instruction is stalled (the initial default of the predictor), it is marked as a candidate for potential same-line access.

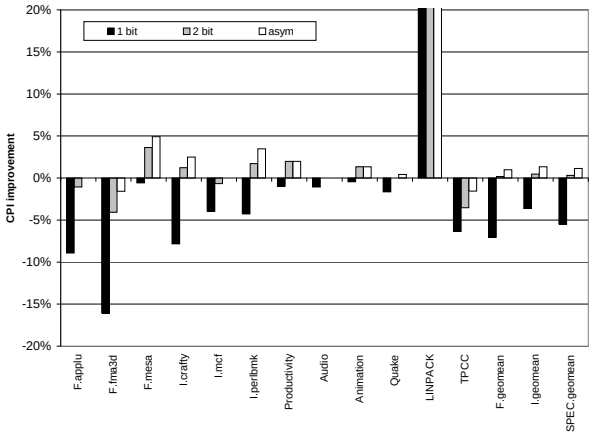

Figure 15: 1-bit, 2-bit, and asymmetric prediction performance improvements

The predictors shown in figure 15, each have 1024 entries accessed by the instruction address, without tag matching. The performance in figure 15 shows that the 1-bit and 2-bit predictors are often overly aggressive, particularly when the recovery costs are high. The 1024-entry asymmetric predictor improves greatly upon the performance of the 1-bit and 2-bit predictors, generally providing a positive performance improvement even under the same large misprediction penalties.

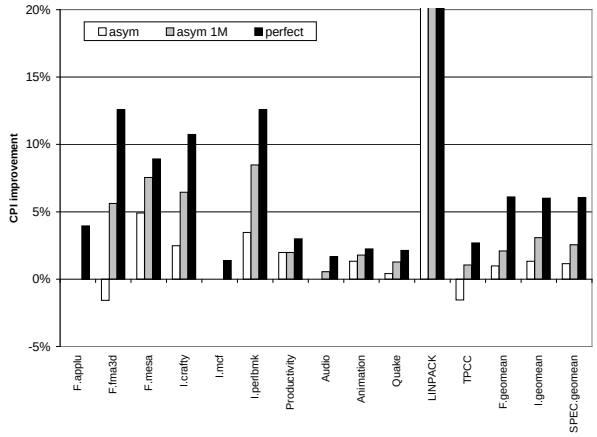

Figure 16: Asymmetric predictor, large asymmetric, and perfect prediction performance improvements

A control experiment with a 1M-entry asymmetric predictor was also performed to establish the impact of table size and false aliasing. This was supplemented with a further control experiment using a perfect predictor. Figure 16 illustrates the relative performance (CPI) improvement afforded by use of a 1024-entry asymmetric same-line access predictor, a large one-million entry asymmetric predictor, and a perfect predictor. The LINPACK benchmark shows a uniform 40% improvement for all three predictors.

From the data, we see that the perfect predictor provides real performance boosts for several workloads, and approximately six percent overall on the SPECcpu set. The large, one-million entry asymmetric predictor provides approximately 50% of the performance of the perfect predictor across the full SPECcpu suite, though it is much closer for some of the workloads that benefit most. The small, 1024-entry asymmetric predictor always performs noticeably better than the 1-bit and 2-bit predictors, but in some cases still results in performance degradation, and overall only achieves about one quarter of the performance improvement of the perfect predictor for the full SPECcpu benchmarks.

We find an overall performance improvement of 2% to 5% over the conservatively issued case for non-pipelined arrays, either mitigating some of the performance degradations experienced by throughput-bound applications, or giving additional improvement for latency-sensitive workloads by increasing the available throughput when feasible.

6 Related work

A number of researchers have considered various approaches to address issues surrounding cache accesses. L0 caches have been proposed to exploit locality of reference at a fine granularity. Several architecture proposals for such L0 or filter caches have been made. Adding a full level of cache with tag comparisons, line select, and so forth, can add significant overhead. Increased bandwidth might be obtained by a deeper pipelining of such a cache. In many aspects, this is akin to downsizing the L1 cache appropriately and adding another level of cache beyond the L1.

An L0 cache might also be sized small enough to allow back to back execution of load operations. However, when the L0 effectiveness falls below a given hit rate (especially in an in-order pipeline), the overall pipeline flow will quickly reflect the cache latency of the the sum of the first two levels of cache, plus any penalties for latency and bandwidth limitations in queueing to the next level cache. Additionally, in a stall-free back-end implementation such as described here, a miss in the L0 cache will incur a pipeline flush when a dependent operation has been issued before the miss condition is detected. To avoid the flush penalties for such a design, either the core must assume worst-case (L0 miss) timing, or predict which cases will miss the L0 cache, and control the issue of those instructions to occur at the L0 miss timing. Such a design is indistinguishable from the design proposed here, where multiple line buffers are retained.

Su and Despain [SD95] describe a block buffer for low power cache operation. In many aspects, this approach is similar to an L0 cache, and the data retention aspects described in this work. By checking the block buffer tag before accessing the cache, the cache latency is effectively lengthened. Since as no information about which memory accesses can be satisfied without cache access, proper scheduling choices can be made and high re-issue penalties will cause both power and performance degradation in design with high upset event recovery cost. In conjunction with the predictive dispatch described in this work, the power savings potential of block buffers can be obtained in high performance microprocessor implementations.

Wilson et al. [WOR96] describe several structures to efficiently exploit cache ports. The suggest a ‘load all’ strategy to satisfy multiple accesses to the same data item, and a ‘load

all wide' variant which allows multiple cache access buffers by replicating tag match logic and selectors in each cache access buffer to satisfy their requests from a single line. This implementation will require significant hardware resources to route the data and tags to multiple access buffers, perform the appropriate selection. They also suggest a wide L0 cache structure organized as FIFO.

Cache banking is a well-known technique and has been employed in several generations of architectures. Beyond just banking a cache, Yoaz et al. [YERJ99], provide a description and evaluation of of bank prediction, wherein the processor uses a predictor to identify when two accesses may collide in the same cache bank. This is clearly related to same-line access prediction, though its focus is on larger cache units (banks) and in preventing these collisions, while same-line access prediction identifies opportunities to access the same cache entity (as a favorable action).

In a similar vein, Kessler [Kes98] describes the use of hit/miss prediction in instruction issue logic. The goal of this work was to predict those accesses which were likely to miss in the cache access, in order to better schedule dependent operations that source the result of a load that is predicted to miss. This allows the processor to avoid some expensive recovery actions on those loads which are predictably misses, and allows a more aggressive speculative issue of the dependent operations for those predicted to hit.

Another direction of cache research has considered speeding-up the overall impact of cache access is to predict the memory access address [GG97]. Predicting the access address allows the processor to bypass the address-generation logic (and even translation, should the prediction return physical addresses) which can save some load/store pipeline execution stages. This technique does not reduce the number of cache accesses, as each predicted address must still be used to access the memory hierarchy.

Another area of cache-related research focuses not on predicting the hit or miss behavior, or the cache access locations, but in actually predicting the cache access results. Widigen et al. [WSM96] propose an Operand Prefetch Cache (OPC) which is used to predict the operand memory read results, thereby providing the operand earlier (in the decode stage) than otherwise available, and potentially eliminating the operand read latency entirely (at least from the critical operation path). Performance studies in [WSM96] show that a 12 Kbyte OPC can predict approximately one third of memory reads with 95% accuracy.

Effectively, the work of [WSM96] is an implementation in the field of *value prediction*, which has been heavily studied. Value prediction [LWS96] predicts the value that would be returned by a load access, allowing the dependent instructions to take the predicted value and speculatively begin execution using that data. This technique may remove load accesses from the critical execution path, but does not reduce the number of cache array access or load instruction executions (which are needed for verifying the predictions).

In a similar approach, John et al. [JTMC98] use a code coalescing unit to retain stored values in a store register rename buffer. Subsequent load accesses to addresses associated with values in the store register rename buffer can then be satisfied by accessing those registers, at register-file speeds. Accesses to this register file are handled using an extension to the already-implemented register renaming hardware. By interposing this enhanced store-buffer, they are able to eliminate some set of load accesses to the data cache; their experiments indicated that between 25% and 42% of the load accesses could have been eliminated with an appropriately-sized buffer. It is not clear how this technique would be applied in a processor that did not implement out-of-order and register-renaming hardware.

Other research work has addressed means for transforming latency-bound benchmarks into more latency-tolerant forms. Chishti and Vijaykumar [CV04] evaluate the use of simultaneous multithreading to turn latency-bound benchmarks into throughput-bound benchmarks. While this is interesting research, it is really outside the scope of our work, which does not address the transformation of the workloads, and really focuses on single-threaded applications (e.g., SPECcpu). Workloads which exhibit thread parallelism provide would generally be throughput-sensitive (rather than latency-sensitive) and thus our techniques would be less directly beneficial.

Cho et al. [CYL01] predict *independent* data access streams, which they then direct to different memory pipelines in an effort to increase overall memory bandwidth.

Bursts of store traffic, which exhibit a similar spatial locality in many cases, are usually automatically smoothed within the processor core by use of *store combining logic*, either in the store queue, or in a separate store combining unit between the processor and the caches. Furthermore, stores are generally not on a critical execution path, and thus do not require the same attention as loads.

The concept of retaining the output of the memory array for speedy access to the same memory line has been used extensively in DRAMs as *"(fast) page mode"* and its derivative modes (e.g., *extended data out/EDO*, *burst EDO*, etc.) [Gui98]. This is used in DRAMs primarily to reduce latency by bypassing the array access and sensing phases.

Capturing write locality to nearby data locations is a common feature in many processors today. Many microprocessors use combining store queues to combine multiple *write* requests

that fall within the storage capacity of a single store queue entry (subject, of course, to the processor architecture's coherence model). Such combining store queues subsequently perform a single write of the combined data to the cache.

7 Conclusion

We have evaluated workloads for throughput and latency sensitivity for different cache design options. We have shown how to increase throughput for non-pipelined array accesses, and proposed to use predictors to establish valid issue pairings. Our results show that workload characteristics for SPEC and a variety of image filtering workloads benchmarks favor high throughput cache design options, whereas TPC-C and several desktop workloads (productivity, video and audio processing) can achieve performance benefits even when trading only 1 cycle reduction in latency for a reduction of bandwidth by a factor of 2.

We have proposed a method to exploit fine-grained memory locality by suppressing cache array access and select data from sense amp latches of the SRAM array in successive cycles. This optimization allows to obtain performance improvements of about 5% to 15% percent, with an average of about 7% performance improvement across SPECcpu2000 workloads by explotining back to back memory execution for cache structures which do not support such throughput rates otherwise. [4]

In architectures with high upset event cost, such as non-stalling pipelines, this performance improvement potential is outweight by the upset event cost when the line buffer cannot satisfy a subsequent access, and lead to performance degradation up to 250%, with an average performance penalty of 85% for SPECcpu2000 workloads.

To address this penalty, we have proposed the use of a predictive dispatch technique, and evaluated several predictors for their effectiveness in predicting valid pairings, establishing a performance improvement potential of 2% to 5% over a conservative alternate cycle dispatch policy in a microprocessor using a flush and re-issue based upset event recovery policy, and preventing performance degradation on any of the benchmarks being evaluated. In our simulations, this is performance equivalent to an increase of 8KB for the first level data cache (25% size change), using a 1000 entry 2bit predictor.

[4] While this work has primarily focused on a full-width line buffer for a 128-byte cache line, the technique described also is also beneficial to designs which include an effective line buffer width that is greater than the typical memory access size. Analysis of address reference streams indicate that even with a 32 byte (sub-)line buffer, there is substantial locality of reference to data to provide opportunity for subsequent accesses to hit the same line buffer data as a preceding access.

Although beyond the scope of this evaluation which put an emphasis on workload performance, the prediction scheme described could be used aggressively to reduce the amount of array accesses necessary to reduce power spent on array accesses. The perfect prediction bar establishes higher potential than has been achieved by any of the predictors evaluated, suggesting that more aggressive predictors could yield even better results. Given that the predictors can be accessed in parallel with instruction caches, more elaborate predictors are possible without impacting the timing-critical dispatch stages.

Report Availability

The publication approval date of this report does not indicate the date of its public availability.

References

[ACG+00] E. Altman, P. Capek, M. Gschwind, H. Hofstee, J. Kahle, R. Nair, S. Sathaye, and J-D Wellman. Symmetric multi-processing system with attached processing units being able to access a shared memory without being structurally configured with an address translation mechanism. US Patent 6779049, December 2000.

[AG95] Sarita Adve and Khourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 1995.

[CFH+05] Leland Chang, David Fried, John Hergenrother, Jeffrey Sleight, Robert Dennard, Robert Montoye, Lidija Sekaric, Sharee McNab, Anna Topol, Charlotte Adams, Kathryn Guarini, and Wilfried Haensch. Stable SRAM cell design for the 32 nm node and beyond. In *VLSI Symposium on Technology*, Kyoto, Japan, June 2005.

[CV04] Z. Chishti and T. Vijaykumar. Wire Delay is not a Problem for SMT (in the Near Future). In *31st International Symposium on Computer Architecture*, München, Germany, June 2004.

[CYL01] Sangyeun Cho, Pen-Chung Yew, and Gyungho Lee. A High-Bandwidth Memory Pipeline for Wide Issue Processors. *IEEE Transactions on Computers*, 50(7):709–723, 2001.

[GG97] José González and Antonio González. Memory Address Prediction for Data Speculation. In *3rd International Euro-Par Conference (Euro-Par'97)*, pages 1084–1091, 1997.

[GHF+05] Michael Gschwind, Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. A novel SIMD architecture for the CELL heterogeneous chip-multiprocessor. In *Hot Chips 17*, Palo Alto, CA, August 2005.

[GKA01] M. Gschwind, S. Kosonocky, and E. Altman. High Frequency Pipeline Architecture using the Recirculation Buffer. IBM Research Report RC23113, March 2001.

[Gsc99] M. Gschwind. Pipeline Control Mechanism for High-Frequency Pipelined Designs. US Patent 6192466, January 1999.

[Gui98] Tom's Hardware Guide. Motherboards and Ram: Ram Guide. http://www6.tomshardware.com/ motherboard/19981024/index.html, October 1998.

[Hof05] Peter Hofstee. Power efficient processor architecture and the Cell processor. In *11th International Symposium on High-Performance Computer Architecture*. IEEE, February 2005.

[JTMC98] Lizy John, Yin Teh, Francis Matus, and Craig Chase. Code Coalescing Unit: A Mechanism to Facilitate Load Store Data Communication. In *International Conference on Computer Design*, pages 550–557, Austin, TX, October 1998.

[Kes98] R.E. Kessler. The Alpha 21264 Microprocessor: Out of order execution at 600 MHz. In *Hot Chips 10*, August 1998.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LWS96] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value Locality and Load Value Prediction. In *Architectural Support for Programming Languages and Operating Systems*, pages 138–147, 1996.

[P+05] Dac Pham et al. The design and implementation of a first-generation CELL processor. In *International Solid-State Circuits Conference Technical Digest*, February 2005.

[SD95] Ching-Long Su and Alvin Despain. Cache design trade-offs for power and performance: A case study. In *Proceedings of the 1995 International Symposium on Low Power Design*, Dana Point, CA, 1995.

[TDF+02] J.M. Tendler, J.S. Dodson, J.S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, January 2002.

[WOR96] K. Wilson, K. Olukotun, and M. Rosenblum. Increasing Cache Efficiency for Dynamic Superscalar Microprocessors. In *23rd Internationcal Symposium on Computer Architecture*, pages 147–157, 1996.

[WSM96] Larry Widigen, Elliott Sowadsky, and Kevin McGrath. Eliminating operand read latency. *Computer Architecture News*, pages 18–22, December 1996.

[YERJ99] Adi Yoaz, Mattan Erez, Ronny Ronen, and Stephan Jourdan. Speculation Techniques for Improving Load Related Instruction Scheduling. In *Proc. of the 26th International Symposium on Computer Architecture*, pages 42–53, May 1999.