On High-Bandwidth Data Cache Design for Multi-Issue Processors

Jude A. Rivers, Gary S. Tyson, Edward S. Davidson
Advanced Computer Architecture Laboratory
The University of Michigan
{jrivers,tyson,davidson}@eecs.umich.edu

Todd M. Austin
MicroComputer Research Labs
Intel Corporation
taustin@ichips.intel.com

Abstract

Highly aggressive multi-issue processor designs of the past few years and projections for the next decade require that we redesign the operation of the cache memory system. The number of instructions that must be processed (including incorrectly predicted ones) will approach 16 or more per cycle. Since memory operations account for about a third of all instructions executed, these systems will have to support multiple data references per cycle. In this paper, we explore reference stream characteristics to determine how best to meet the need for ever increasing access rates. We identify limitations of existing multi-ported cache designs and propose a new structure, the Locality-Based Interleaved Cache (LBIC), to exploit the characteristics of the data reference stream while approaching the economy of traditional multi-bank cache design. Experimental results show that the LBIC structure is capable of outperforming current multi-ported approaches.

1. Introduction

Improvements in microprocessor performance continue to surpass the performance gains of their memory subsystems. Higher clock rates and increasing numbers of instructions issued and executed in parallel account for much of this improvement. By exploiting instruction level parallelism (ILP), these processors are capable of issuing multiple instructions per cycle, which places a greater demand on the memory system to service multiple requests per cycle. As microprocessor designers push for more performance, the trend to aggressively exploit more and more ILP will continue; targets of up to 16 instructions (i.e. an average of three basic blocks) per clock cycle are already being explored [1]. With about a third of a program's instruction mix being memory references [2], an average of 6 load/stores per cycle is necessary in order to sustain a 16-wide issue rate. With such demands, current single- and dual-ported cache implementations are clearly inadequate. There is a need to explore low-cost techniques for increasing the effective number of ports in order to meet the need for sustainable cache bandwidth.

Currently, multiple cache ports are implementable in one of four ways: either by conventional and costly ideal multiporting, by time division multiplexing, by replicating multiple single-port copies of the cache, or (with lower performance and possibly lower cost) by interleaving the cache through multiple independently-addressed banks. Conceptually, ideal multi-porting requires that all p ports of a p-ported cache be able to operate independently, allowing up to p cache accesses per cycle to any addresses. However, ideal multiporting is generally considered too costly and impractical for commercial implementation for anything larger than a register file. Current commercial multiporting implementations therefore use one of the remaining three techniques.

The time division multiplexed technique (virtual multi-porting), employed in the IBM Power2 [14] and the new DEC Alpha 21264 [6], achieves dual-porting by running the cache SRAM at twice the speed of the processor clock. As data access parallelism moves beyond 2, extending this technique by running SRAMs p times as fast as the processor will become infeasible as a multiporting solution. Consequently, we do not explore this technique any further in this paper.

The data cache implementation in the DEC Alpha 21164 [5] provides an example of multi-porting through multiple copy replication. The 21164 implements a two-ported cache by maintaining identical copies of the data set in each cache. To keep both copies coherent, every store operation must be sent to both cache ports simultaneously, thus reducing the effectiveness and scalability of this approach relative to ideal multi-porting. Another major cost of this approach is the die area necessary for cache replication.

A 2-bank (interleaved) data cache is found, for example, in the MIPS R10000 [4]. A simultaneously served pair of data references must address different banks. With a well balanced and well scheduled memory reference stream, this approach can boost data access parallelism and deliver high bandwidth. With the wrong memory reference stream, however, bank access conflicts can seriously degrade the delivered performance toward single-ported, single bank performance. Although dividing a

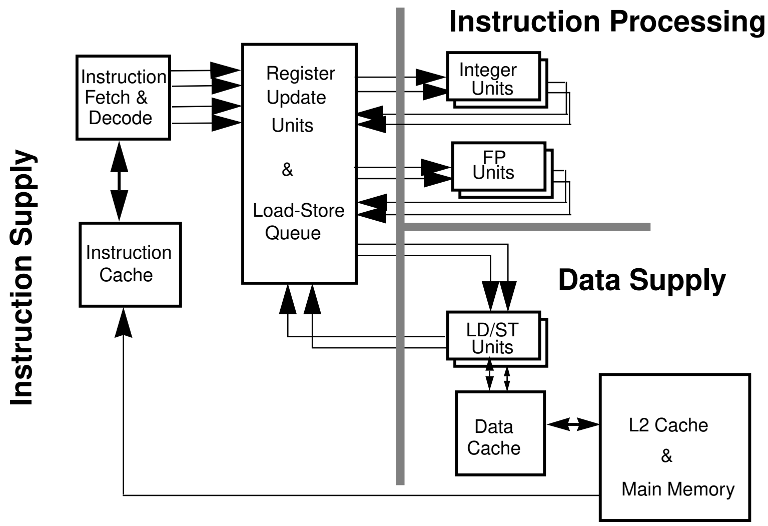


Figure 1: The schematic structure of our simulated dynamic superscalar processor. Our focus is on the bandwidth requirements of the data supply portion

cache into banks can be economical, the cost of the crossbar between the load/store units and the cache ports grows superlinearly as the banks (and ports) increase.

Even with high hit ratios, these multi-porting techniques fall short of the ideal performance threshold due to the need for broadcast writes in the replicated design, and the existence of bank conflicts in the multi-bank approach. In this study, we evaluate these implementations and their scalability to ideal multiporting as data access parallelism increases. Though each technique has significant costs and drawbacks, we find that multi-banking holds the key to a low cost cache memory design that can cope with increasing degrees of instruction level parallelism. A look at the memory reference stream reveals that a substantial number of bank conflicts are caused by references to the same cache line. In this paper, we propose and evaluate the Locality-Based Interleaved Cache (LBIC), an enhanced multi-bank design, that employs a single line multi-ported buffer per bank to reduce bank conflicts. The LBIC, by exploiting same line locality, scales well toward ideal multiporting with an implementation cost close to traditional multi-banking.

In the next section, we describe the architectural assumptions made in this study, the simulation environment, and the characteristics of the benchmarks used. In Section 3, we examine the three multiple cache port approaches: ideal multi-porting, replicated multi-porting and multi-banking. Section 4 presents characteristics of the memory reference stream, and the effects on multi-banking performance. Our analysis in Sections 3 and 4 serve as the basis for the introduction of the Locality-Based Interleaved Cache (LBIC) in Section 5. An evaluation of the LBIC design is given in Section 6, and Section 7 concludes this work.

2. Methodology

Given the emerging trend toward aggressive ILP exploitation through multiple instruction issue, we anticipate that there will be an increasing burden on the data cache to service ever more numerous requests per cycle. To capture this hypothesis in our experiments, we model a dynamic superscalar processor with a very high issue rate. Higher issue rates demand even higher-bandwidth instruction supply and increased resources for instruction processing. More resources for instruction processing basically entails adding functional units (made possible by VLSI advances) whereas high-bandwidth instruction supply requires sophisticated branch prediction techniques, more complex decode/issue logic, and high-bandwidth fetch strategies (e.g. the Fill Unit [19][20] and the Trace Cache [3]), which are relevant, but beyond the scope of this paper.

Modern branch predictors are already quite accurate and more accurate branch predictors, capable of predicting across multiple basic blocks, are expected in the near future. Future processors are also bound to exploit more parallelism through more aggressive speculation across multiple basic blocks. These developments will place a far greater demand on the data memory than current processors do. We must note that more accurate branch prediction for dynamic superscalar processors has not decreased the resource requirements needed for processor execution, but has rather increased the useful work completed per cycle.

2.1. Architectural Assumptions

Our processor architecture is an extended version of the SimpleScalar [17] sim-outorder simulator which performs

Fetch Mechanism	fetches up to 64 instructions in program order per cycle
Instruction Cache	perfect cache, 1 cycle hit latency
Branch Predictor	perfect branch prediction
Issue Mechanism	out-of-order issue of up to 64 operations per cycle, 1024 entry re-order buffer (RUU), 512 entry load/store queue (LSQ), loads may execute when all prior store addresses are known
Functional Units	64-integer ALU, 64-FP adders, 64-integer MULT/DIV, 64-FP MULT/DIV, varying # of L/S units.
Functional Unit Latency (total/issue)	integer ALU-1/1, integer MULT-3/1, integer DIV-12/12, FP adder-2/1, FP MULT-4/1, FP DIV-12/12, load/store-1/1
Data Cache	32KB direct-mapped, write-back, write-allocate, 32 byte lines, 1/4 cycle hit/miss latency, (varying # of ports), non-blocking, supports one outstanding miss per physical register

Table 1: Baseline Processor/Memory Model

out-of-order issue, execution and completion on a derivative of the MIPS instruction set architecture. Figure 1 provides a schematic picture of the simulated processor. Almost all recent (e.g. the Intel P6 [16], the MIPS R10000 [4]) and new architectures (e.g. the HP PA-8500 [15], the DEC 21264 [6]) rely on out-of-order issue and/or out-of-order execution for exploiting parallelism, and this trend is likely to continue. In particular, these architectures are using increasingly large register update units (or instruction windows), e.g. 56 entries in the HP-PA8500, and enabling *memory re-ordering* techniques (e.g. allowing loads to execute before stores) in exploiting parallelism through dynamic execution ordering.

Since our study focuses on handling the bandwidth requirements of the data supply portion of future multiple issue processors, we designed a near perfect front end (i.e. the instruction supply portion) to our processor simulator. In addition, we provided adequate resources to the instruction processing phase so as to highlight the effect of bandwidth in data supply. In particular, we assume a perfect branch predictor and a considerable number of functional resources. Table 1 details our chosen parameters and architectural assumptions. Though a 64-way superscalar processor appears unlikely to be implemented any time soon, removing issue constraints, except for those required by the semantics of the benchmark programs we study, helps to extract the highest level of data access parallelism that exists within program constructs and thereby highlights the effects of the cache.

Our simulated processor uses a register update unit (RUU) [21] to keep track of instruction dependencies; and a load/store queue (LSQ), an address reorder buffer that prevents loads from bypassing stores to the same address. Loads, with their effective addresses computed and dependencies resolved, are sent from the LSQ to the cache at issue time, while stores are actually written to the cache at commit time. Loads to same address as an earlier store in the LSQ can be serviced with zero latency by the corresponding store.

Our memory subsystem consists of separate instruction and data caches, a secondary data cache, and main memory. The primary instruction cache is perfect and responds in a single cycle. The primary data cache is a nonblocking, 32K byte direct-mapped cache, with 32 byte lines and a single cycle access time. The L2 cache is a 512 KByte four-way set-associative cache with 64 byte lines and a four cycle access time. Accesses from L1 to L2 are fully pipelined, and a miss request can be sent every cycle, up to 64 pending requests. The main memory access latency is just 10 cycles. As this study seeks to stress bandwidth instead of latency to memory, our interest lies in exposing the degree of data access parallelism that exists across various code constructs, rather than the effect of the memory latency seen by misses to the cache.

2.2. The Instructions Per Cycle Metric

IPC, instructions per cycle, is a common metric of choice for evaluating multi-issue processors and their extensions. True IPC, as a measure, must evaluate three phases of instruction level parallelism: the issue phase, the execution phase and the completion phase.

Typically, IPC is taken as the number of instructions that the processor completes per cycle. Such a metric, however, fails to consider some of the activities and resource requirements that occur at the front-end of the processor. For example, at the issue phase, a speculative processor may issue numerous instructions that access the data cache, but are later discarded for mis-speculation. In such instances, IPC as a measure of instructions completed per cycle, fails to expose the data resource requirements for the work done on the wrong path. This could lead to a

Program	Instr. Count (Mil.)	Mem Instr. (%)	Store-to-Load Ratio	L1 Miss Rate (32KB)
SPEC95 INT Benchmarks				
Compress	35.69	37.4	0.81	0.0542
GCC	264.80	36.7	0.59	0.0240
Go	548.12	28.7	0.36	0.0271
Li	956.30	47.6	0.59	0.0084
Perl	1,500.00	43.7	0.69	0.0265
SPEC95 FP Benchmarks				
Hydro2d	967.08	25.9	0.30	0.1010
Mgrid	1,500.00	36.8	0.04	0.0402
Su2cor	1,034.36	32.0	0.32	0.1307
Swim	796.53	29.5	0.28	0.0615
Wave5	1,500.00	31.6	0.39	0.1103

Table 2: The ten benchmarks and their memory characteristics

seemingly realistic argument that for an IPC of 3, assuming that a third of an application's instruction mix is memory load/stores, the memory system only needs to support 1 access per cycle. However, 2 or 3 accesses will appear in some 3-instruction groups, and many instructions may be executed speculatively, and later discarded. Hence, it may take a peak rate of 2 or 3 memory operations per cycle to achieve an IPC of 3.

It is valid, however, to use IPC as the metric for comparison among the various high-bandwidth organizations. For our results, there is no speculation effect on IPC since our processor does not speculate.

2.3. Benchmarks

We selected ten programs (5 integer and 5 floating point) from the SPEC95 benchmark suite for this study. In choosing benchmarks for analysis, we looked for programs with varying memory requirements. To expose the extent of data parallelism that exist across different code segments, it is necessary to look at programs with large and small data sets as well as programs with high and low reference locality. In addition to exhibiting those characteristics, programs from SPEC95 also stress the memory system much more than their counterparts from the SPEC92 set.

As Table 2 shows, we simulated each benchmark either to completion or to the first 1.5 billion instructions. We believe that simulating samples or tiny portions of a program is risky for a high-bandwidth data study. In particular, memory reference patterns can vary among different phases of program execution, which is likely to result in burst data accesses at some points during program execution. A sampled or a minimal partial simulation may fail to capture such a trend and is therefore likely to present a distorted picture of a program's memory reference behavior and its memory resource requirements.

Table 2 lists the memory characteristics of the benchmarks. The store-to-load ratio column provides the available stores per each load reference. We also provide the miss rates of the benchmarks for the 32KB direct-mapped L1 cache used for this study.

3. Conventional Multi-Ported Solutions

Multi-banking and multi-porting by replication appear to be the practical approximations to ideal multi-porting as data accesses per cycle increase. Whereas the multi-bank approach can work well for applications that lack locality because of the statistically independent nature in which memory references are presented to the cache, it could hurt the performance of applications with good spatial locality if the reference pattern is such that consecutive references map to the same bank, especially where the data layout is *cache line interleaved*[a]. For most current single-ported multi-bank implementations, multiple accesses to the same cache line may not proceed in parallel. Replicated caches, on the other hand, do not scale to the ideal performance because of the need for broadcasting stores. In the next two subsections, we describe the pros and cons of replicated and multi-bank design implementations, and attempt to explain their performance variations and how they scale to ideal multi-porting.

3.1. Multi-Ported Caches

Figure 2a illustrates a multi-ported cache. Each port is provided with its own data path to every entry in the cache, implemented either by replicating the entire single-ported cache for each port and maintaining the same contents in each copy, or by multi-porting the individual cache SRAM cells (ideal multiporting). The ideal multiporting implementation appears to deliver the highest bandwidth. However, practical implementations of these designs suffer from several drawbacks. One is the circuit complexity resulting from increasing capacitance and resistance load on each access path as the number of ports increases, which can lead to longer access latency for ideal multi-porting, and longer store time for replicated designs. Another concerns the chip area overhead necessary for replication and the many extra wires and additional com-

[a] Line interleaved refers to banking the cache such that a cache line resides completely in one bank and consecutive cache lines reside in successive banks. Vector supercomputers are usually 8-byte *word interleaved*, where a single memory block spreads across different banks. Word interleaving is efficient for reducing bank conflicts but costly due to the need for tag replication in each bank or multi-porting the tag store to allow simultaneous access from all banks.

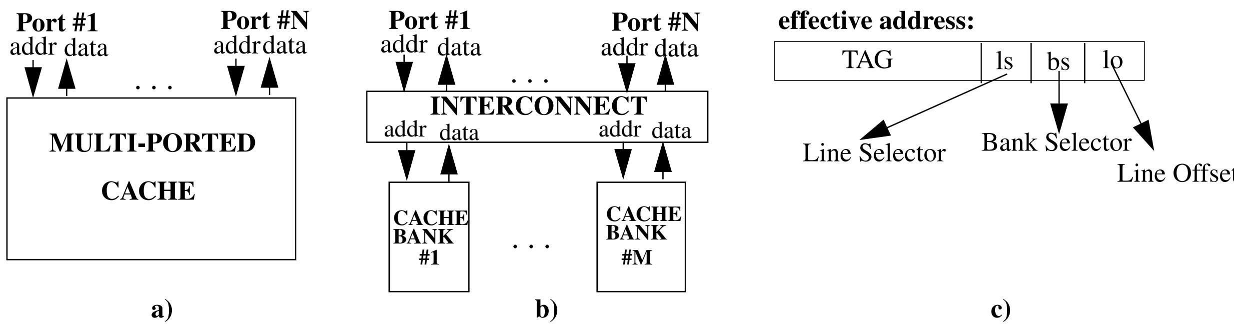


Figure 2: Conventional high-bandwidth cache memory organizations: a) True multi-ported, and b) Multi-bank (interleaved). c) Effective address decomposition for multi-bank caches. We use the bank selector (bs) bits of the effective address for selecting the appropriate bank.

Ports --->	**1**	**2**			**4**			**8**			**16**		
Programs	~	True	Repl	Bank	True	Repl	Bank	True	Repl	Bank	True	Repl	Bank
Compress	2.66	5.22	4.08	3.95	7.41	5.15	5.12	7.83	5.55	5.86	7.83	5.68	5.96
Gcc	2.65	4.80	4.03	4.15	6.19	4.99	5.23	6.27	5.29	5.61	6.27	5.35	5.70
Go	3.44	5.62	5.32	4.80	6.82	6.53	5.87	7.13	6.95	6.45	7.17	7.02	6.67
Li	2.10	4.17	3.42	3.78	6.58	4.76	5.84	6.58	5.33	6.34	6.58	5.43	6.48
Perl	2.25	4.48	3.52	3.51	7.08	4.67	4.57	7.25	5.29	5.85	7.25	5.49	6.30
SPECint Ave.	**2.55**	**4.80**	**3.98**	**3.99**	**6.79**	**5.14**	**5.28**	**6.97**	**5.62**	**6.01**	**6.98**	**5.73**	**6.20**
Hydro2d	3.76	7.19	6.32	6.41	9.94	8.96	8.64	10.6	9.88	9.24	10.7	10.1	9.70
Mgrid	2.67	5.11	5.07	4.97	9.64	9.49	7.90	16.6	16.2	9.32	18.6	18.6	10.2
Su2cor	3.01	5.93	5.21	5.29	9.04	7.75	7.41	10.3	9.39	7.83	10.8	10.2	8.45
Swim	3.20	6.36	5.46	5.46	10.0	8.53	6.19	12.8	10.7	6.82	13.6	11.2	6.90
Wave5	3.28	6.01	5.26	5.58	7.26	6.76	6.28	7.53	7.30	6.55	7.56	7.42	6.74
SPECfp Ave.	**3.14**	**6.04**	**5.43**	**5.50**	**9.05**	**8.18**	**7.16**	**10.8**	**10.0**	**7.78**	**11.2**	**10.5**	**8.16**

Table 3: IPC for ideal multi-porting (True), multi-porting by replication (Repl), and multi-banking (Bank).

parator logic needed to implement each port. Additionally, implementation by replication also sacrifices some bandwidth since store accesses must be sent simultaneously to all cache copies to ensure coherence. Consequently, a store cannot be sent to the cache in parallel with any other access. Except for **compress** (worse) or **mgrid** (better), about 1/4 to 1/3 of all memory operations are stores (see Table 2). Therefore this restriction can be quite limiting.

In this section, we discuss the performance of both ideal multi-porting and multi-porting by replication [5]. The performance of ideal multi-porting is offered as a basis for examining how many cache ports are adequate for various applications, and also to assess the performance degradation of the other designs. Table 3 presents the IPC data for ideal multi-porting (True), multi-porting by replication (Repl.), and multi-banking (Bank) for each benchmark as the number of ports increase from 1 to 16 in powers of 2. As expected, multiple cache ports are very important for processors with large issue widths. For our 32 KByte direct-mapped cache, increasing the number of ideal cache ports from one to two shows as high as 89% and 92% performance improvements for the average SPECint and SPECfp programs, respectively. Some individual programs exhibit higher performance gains: 96% for **compress**, 99% for **swim**, and 98% for **li**. From two to four ports shows an average performance gain of 41% for SPECint and 50% for SPECfp. Eight ideal ports appear sufficient for the SPECint benchmarks since increasing ports to sixteen returns a meagre average 0.12% performance improvement. For SPECfp, however, an increase from 8 to 16 ideal ports delivers about 4% improvement in performance.

For multi-porting by replication, each port is connected to its own 32 KByte direct-mapped cache. To maintain coherence, each store needs to be sent to all the caches simultaneously. Clearly, the degradation from this constraint is evident from the Repl. IPC numbers in Table 3. The performance improvement trend as cache ports increase is evident, but fails to scale to ideal multi-porting. For integer programs with high store-to-load ratios (like **compress** .81; **gcc** .59; **li** .59; and **perl** .69), the non-scalability with increasing cache ports can be significant; for **mgrid** .04, it is insignificant and 16 port performance is virtually indistinguishable from ideal.

3.2. Multi-Bank Caches

A multi-bank (or interleaved) cache, shown in Figure 2b, employs a crossbar interconnection to distribute the memory reference stream among multiple cache banks. Each bank of a multi-bank cache can independently service one cache request per cycle, and each bank in this study has a single port. As long as simultaneous accesses map to independent banks, this technique delivers high bandwidth access. This design has a lower latency and area requirement than the replicated cache, especially for large cache sizes. While its crossbar interconnect adds significant cost and latency to the access path, its smaller, single ported banks are less costly and faster; thus a direct cost/latency comparison is highly dependent on the implementation technology and configuration. Using an omega network rather than a crossbar would alter this tradeoff, increasing latency, but reducing cost for larger configurations.

A bank selection function is necessary for mapping memory reference addresses onto corresponding cache banks. This function can affect the bandwidth delivered by the multi-bank implementation since it influences the distribution of the accesses to the banks. An inefficient function may increase bank conflicts, reducing the delivered bandwidth. Many bank selection functions have been proposed and shown effective in the literature [10][11], especially for dealing with large multi-bank memories for vector processors. Efficient functions must, however, be weighed against implementation complexity and the possibility of lengthening the cache access time. This therefore renders accurate, but complex selection functions highly unattractive for cache design. In our experiments, we use *bit selection* [8] (see Figure 2c), a simple function which uses a portion of the effective address as the bank number; data layout in cache is thus line interleaved. As we see shortly, the choice of a selection function may not be as critical as we thought since much of the loss of bandwidth due to same bank collisions map to the same cache line.

Table 3 reports the multi-bank IPC numbers for each benchmark as the number of cache banks increase. We assume a single cache port per bank, and do not add additional time for traversing the crossbar interconnect. Actual multi-bank designs can be pipelined to hide some of the interconnect latency. For these experiments, the number of banks is limited to sixteen. While the data in Table 3 demonstrates reasonable performance with increasing number of banks (and cache ports) for the multi-banking approach, the performance peaks at an average 6.202 IPC for the 16-bank cache which significantly trails the 6.791 IPC of the ideal 4-port cache for SPECint. Even with 16 banks, performance remains lower than that of an ideal 4-port cache (except for **mgrid**).

Compared to multi-porting by replication, however, multi-bank performance should scale better with increasing ports due to the data access serialization brought about by stores in replicated caches. As the number of ports increase, the performance of multi-banking overtakes multi-porting by replication, particularly for store-intensive programs like **compress** and **gcc**. For instance, a replicated 8-port cache simulating **compress** achieves only 95% of an 8-bank cache performance. For programs like **mgrid** and **swim** where this trend appears not to hold, bank conflicts appear to be responsible for the multi-bank performance degradation as we see in the next section.

Though bank conflicts do limit multi-bank performance, the multi-bank approach appears to offer a better cost/performance approximation to ideal multi-porting as processor issue widths widen. In particular, as we approach wider issue widths with large instruction windows for superscalar designs, the memory accesses needed per cycle are likely to spread across more than a single cache line. Intuitively, a multi-bank design that places consecutive cache lines in different banks offers the possibility for parallel multiple cache accesses.

4. Characteristics of the Memory Reference Stream

To understand why the performance of the multi-bank cache significantly lags behind ideal multi-ported cache performance, we must understand the nature of the memory reference stream as presented to the cache structure. We explore how the nature of the memory reference stream can promote bank conflicts, and techniques that can help alleviate conflicts in multi-bank designs.

Figure 3 shows the likelihood of bank conflicts between consecutive cache references in a four-bank cache. We collected these statistics by assuming an infinite size four-bank cache with 32 byte lines. These numbers are meant to serve as an upper bound for distribution of accesses among the banks. In a dynamic superscalar processor environment like the one we consider for this study, some of these accesses will be satisfied within the LSQ, without ever reaching the cache. Each column in the figure shows the dynamic frequency distribution for one of the benchmarks, while SPECint Ave. and SPECfp Ave. represent the average distribution of these SPEC integer and floating point programs respectively. The columns are separated into 5

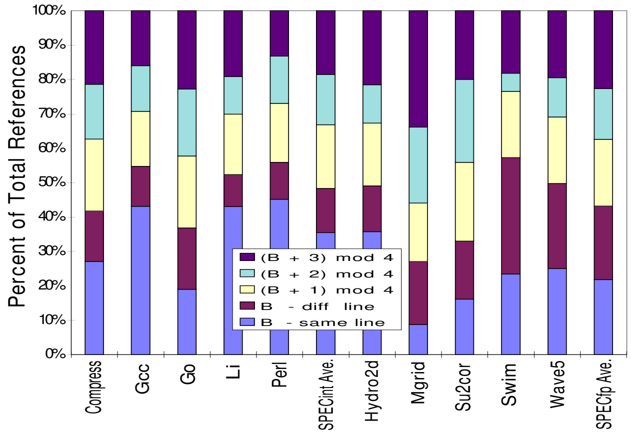


Figure 3: Consecutive memory reference mapping analysis for an infinite 4-Bank Cache structure

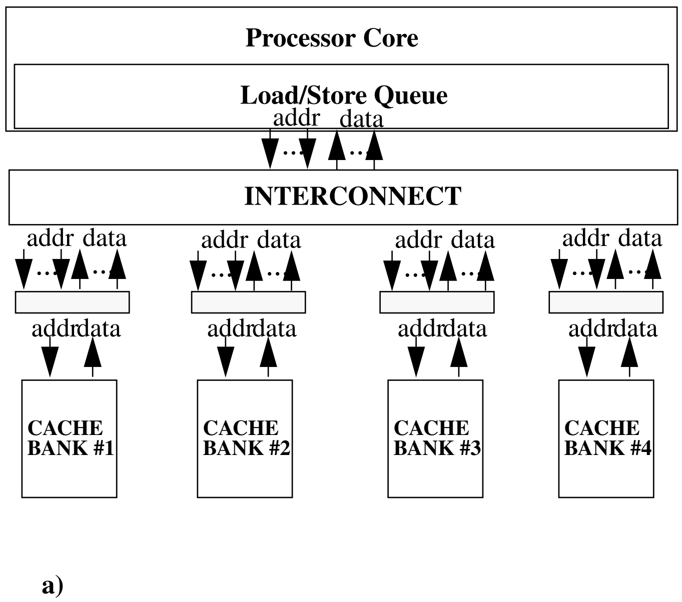
segments, the first two segments correspond to one bank while the other three correspond to the other three banks in the cache structure. The 2 lowest segments, *B-same line* and *B-diff line*, show how frequently a reference's immediate successor maps into the same bank, and which is more likely to cause a conflict. The other three segments, *(B + i) mod 4* with *i = 1, 2, 3*, show how frequently the immediate successor maps to each of the other three banks: the next bank, the bank after that, etc.

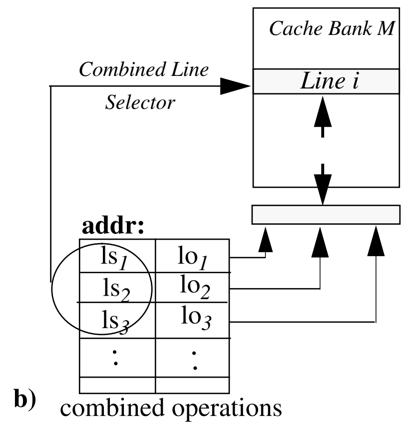
For a uniform, independent reference stream distribution, the probability for each of the four segments would be 0.25. However, most applications show a skewed probability toward same bank (i.e. *B-same line* plus *B-diff line*) - averaging 49% across the integer benchmarks, 44% across the floating point benchmarks, and 50% or more in **gcc**, **li**, **perl**, and **swim**. This clearly limits the effectiveness of the multi-bank approach. While bank conflicts can be substantially reduced with a large number of word interleaved memory banks (as is done in large vector processors like the Hitachi S-3800 (512 banks) [22]), this approach may not work well for caches for two reasons. First, a large number of banks will result in larger and slower interconnects, which could further lengthen the time to access the cache. Second, word interleaving is costly since the tag store would need to be replicated or multi-ported. For line interleaved data layout, a cache line of 8 words carries a single tag, but 8 copies are needed for word interleaving. Increasing the number of banks without word interleaving however, may not reduce bank conflicts significantly. As we show below, our analysis suggests that even with an infinite number of banks, a substantial fraction of the bank conflicts we see in these programs could remain since they are caused by items mapping to the same cache line.

Due to the restrictions of the multi-bank model, consecutive references that access the same bank but different lines cannot proceed in parallel, and need to be serialized. A higher *B-diff line* probability, like 33.81% for **swim** and 24.73% for **wave5**, therefore indicates a greater difficulty in improving multi-bank performance through bank conflict reduction, although increasing the number of banks may help. Interestingly, the integer programs appear less susceptible to same bank with different lines conflicts, averaging only 12.85% in *B-diff line* probability, whereas the floating point benchmarks have a 21.42% average *B-diff line* probability. This behavior by the floating point benchmarks is quite appropriate, given that our experiment considers a cache line size of just 32 bytes (or 4 double-words). Floating point programs, with non-unit strides, will therefore tend to have consecutive references map to different cache lines, even when both are in the same bank.

The *B-same line* probabilities demonstrate the inherent spatial locality that exists in the benchmarks. For the SPECint benchmarks and **hydro2d,** more than half of the consecutive references that map to the same bank also map to the same cache line, as evident from Figure 3. On average, same line accounts for 35.4% of the SPECint references, although only 21.8% of the SPECfp references. For programs like **gcc**, **li** and **perl**, more than 40% of all consecutive references access the same line in the same cache bank. This inherent spatial locality in program reference patterns can be exploited to improve multi-bank delivered bandwidth through *access combining*.

Access combining, a technique developed concurrently by Wilson et. al. [7] and Austin and Sohi [8], attempts to combine references to the same cache line into a single request. Combining devotes additional cache resources to areas in the design that can best exploit spatial locality. Combining works as follows: Accessing stored data in a conventional cache can be viewed as an indexing operation into a two dimensional matrix, using the cache line selector and line offset fields of the effective address. Combining incorporates additional logic in the load/store queue (address reorder buffer), along with limited cache line multi-porting, to improve access throughput. Ideal multi-porting, we must recall, allows multiple requests to be processed regardless of the relationship among their addresses. For a multi-bank approach therefore, incorporating combining logic enables multiple references to the same cache line to collapse into one request for a single cache line, with multiple line offsets. This allows multiple references to the same line while requiring only a single multi-ported line buffer to be included in the bank implementation. When combining is implemented in each bank of a multi-bank cache, the throughput can be increased significantly. In a four-bank cache with two-port combining logic in each bank, up to eight references can be processed in each cycle.





c)

addr:

Ref	Bank	Line	Offset
store	0	12	0
load	1	10	4
load	1	10	8
store	0	12	12

Figure 4: a) A schematic diagram of a 4x*N* locality-based interleaved cache (4x*N* LBIC), 4 banks each with an *N* ported buffer, b) Basic combining logic for a single bank in an LBIC design, c) A data access pattern for performance comparison.

5. The Locality-Based Interleaved Cache (LBIC)

Our analysis in the previous section has shown how the nature of program reference patterns contribute to bank conflicts, and the corresponding effect that this has on multi-bank cache performance. The use of access combining appears theoretically favorable for improving multi-banking performance to levels closer to ideal multi-porting. Another useful technique that helps reduce bank conflicts is memory re-ordering, as shown in [13]. Efficient memory re-ordering ensures that independent references can be assembled to provide enough accesses to as many banks as possible in each cycle. Since our simulator does memory re-ordering through a large LSQ, our IPC numbers in Section 3 already reflect that optimization, except that the traditional multi-bank cache fails to benefit from this. Access combining, as a technique, facilitates efficient exploitation of spatial locality in a program's data set. In this section, we introduce a multi-bank cache design that uses these two concepts for performance improvement. The concept of memory re-ordering is already implemented, as address re-order buffers or load/store queues, in current dynamic superscalar processors [4]. Thus, current processors already implement an optimization scheme that can help multi-banking to scale well. In those processor designs, very little additional logic is necessary to perform access combining in the LSQ - the comparison circuit already exists for determining whether a memory conflict occurs.

5.1. The LBIC Structure

We propose the Locality-Based Interleaved Cache (LBIC) for two reasons: to handle multiple data accesses per cycle, and do so with lower implementation cost. There are two design goals: assembling as many independent references as possible through memory reference re-ordering, and exploiting the spatial locality within an application's data through access combining. An LBIC structure thus consists of a traditional multi-bank cache with a single line multi-ported buffer on each bank.

An LBIC structure is described by the *M*x*N* configuration, where *M* denotes the number of banks in the cache structure, and *N* is the maximum number of ports to any single cache line in a bank. The cache structure, consisting of the *M* banks, remains the same as the traditional multi-bank cache. Cache configuration parameters are flexible, except that the data layout in the multi-bank design must be cache line interleaved. This requirement is necessary as a word interleaved or a sub-block interleaved layout necessitates replicating or multiporting the tag storage. An *N*-ported single line buffer is associated with each bank. This line buffer is not meant to hide access latency but rather to act as a distribution center for the simultaneous supply of up to *N* accesses from one cache line. A total of *N* line offset requests to the same cache line in a bank can be handled simultaneously. There are several low cost techniques for implementing a multi-ported single line buffer. One approach is to employ the technique used in multi-porting a register file. This approach is a well understood problem with true multi-porting solutions that do not lie in the processor's critical path. Figure 4a illustrates a schematic diagram of an *MxN* LBIC structure, where $M = 4$. With four banks and an *N*-ported single line buffer per bank, the 4x*N* LBIC can potentially process up to 4*N* data cache accesses in parallel per cycle.

5.2. LBIC Implementation Issues

An LBIC implementation requires a memory reorder buffer or a LSQ. The main control mechanism of the LBIC design resides in the LSQ, and performance of the scheme depends on the depth of the LSQ. Deeper LSQs will help to minimize possible performance degradation due to insufficient data requests for combining.

LSQs in current processors use comparison circuits for dynamic memory disambiguation. To implement the LBIC combining scheme, this logic can be extended so that when a load/store (leading) request is launched, the bank and line selector bits of its effective address are simultaneously compared with pending ready load/store requests in the LSQ. Alternatively, to reduce the compare circuitry and to ensure a fast compare time, items inserted in the LSQ can be sorted out into the respective bank queues as their effective addresses become known. With this approach, the line selector bits of the leading request are compared with the ready LSQ entries mapping to the same bank. When the leading request hits in the cache, the cache line is gated into the multi-ported line buffer of that bank. Up to *N-1* matching requests (in addition to the leading request) may send their line offsets in parallel to that bank. The load request offsets select data from the *N*-ported buffer.

Unlike the replicated cache structure where store requests can cause serialization of accesses, the LBIC relies on a store queue in each bank, as some current multi-bank implementations do [18], for handling multiple stores or any combination of matching stores and loads per cycle. The store queue in each bank is assumed to be a structure that can hold up to some number of words of store data. The store queue uses idle cycles, as used in the HP PA8000 implementation [18], to perform stores to the data cache. With such a structure, when a cache line selector pulls a cache line into the line buffer, the matching load offsets select their data from the buffer and the matching store offsets deposit their data in the store queue simultaneously. This approach even allows a load followed by a store to the same memory location to be accepted in the same cycle. A matching load closely following a store is satisfied in the LSQ, and does not even reach the cache structure.

Figure 4b illustrates the basic operation of the LBIC combining logic for three requests mapping to Line *i* in Bank *M*. In this example, the three requests logically combine into a single cache line operation. The line selector, ls_1, of the leading request is sent to Bank *M* to gate the data in Line *i* into the line offset buffer. Simultaneously, the other two load/store requests compare their *ls* fields with ls_1 and since they match, the three line offsets (lo_1, lo_2, lo_3) are sent in parallel to select their data from the buffer (for load requests). Matching store requests, if any, simply deposit their data into the store queue accompanying Bank *M*.

The LBIC control, in this preliminary study, does not prioritize with regard to which LSQ accesses are issued in a cycle. Instead, it relies on the LSQ memory scheduling logic and attempts to combine accesses with the leading request at the head of the LSQ. One possible enhancement for further improving the LBIC's performance is to add some selecting LSQ logic that attempts to find the largest group of combinable ready accesses in the LSQ. Larger access groups can therefore be given priority over smaller groups. Furthermore, the smaller groups may grow larger by the time they are selected. Special priority may be given to critical requests that may cause processor stalls. We settled on the leading request because we believe it is fair and simple. The sorting logic for the enhancement above may be costly.

The LBIC combining scheme should have no adverse impact on the cache access latency, beyond that of traditional multi-banking. In addition, area costs are limited to the multi-ported line buffer per bank, the necessary hit signal gates, and multiplexors necessary for selecting appropriate data from the line buffers. In addition, with any heavily multi-ported cache design, there is an increase in the required number of buses necessary for transferring data between the cache structure and the processor.

6. Effectiveness of the LBIC

Theoretically, the performance of the LBIC scheme should scale well towards ideal multi-porting since the scheme has the ability to both exploit independent random references and extract the spatial locality that exists at specific points within an application's data set. In addition, the LBIC suffers less of the traditional multi-bank conflict problems, and no store coherence problems as in multi-porting by replication. For some reference patterns, the LBIC approach is a natural choice for providing high bandwidth.

Consider the references in Figure 4c to be ready entries in a LSQ. Whereas a 2-way multi-bank cache will require two cycles to execute these load/stores (assuming that the items hit in the cache), a multi-ported cache by replication will use three cycles (one cycle per store, plus one for the

Programs	2x2	2x4	4x2	4x4	8x2	8x4
Compress	4.608	4.741	5.521	5.567	5.985	5.991
Gcc	5.256	5.510	5.680	5.716	5.765	5.775
Go	5.849	6.151	6.528	6.640	6.800	6.844
Li	5.805	6.437	6.505	6.515	6.526	6.529
Perl	4.715	5.087	5.905	6.221	6.687	6.722
SPECint Ave.	**5.194**	**5.513**	**6.000**	**6.102**	**6.326**	**6.344**
Hydro2d	9.168	10.215	9.953	10.355	10.163	10.391
Mgrid	8.537	11.292	11.851	15.026	14.301	16.582
Su2cor	7.645	8.287	8.395	8.832	8.955	10.110
Swim	8.283	10.181	8.867	10.366	9.104	10.412
Wave5	6.780	6.993	6.995	7.106	7.082	7.213
SPECfp Ave.	**7.977**	**9.118**	**8.933**	**9.736**	**9.415**	**10.201**

Table 4: IPC for six *M*x*N* LBIC configurations.

two loads). A 2x2 LBIC, however, will be able to handle all four requests in a single cycle.

To evaluate the effectiveness of the LBIC design, Table 4 presents IPC numbers for six *MxN* LBIC configurations. For performance evaluation, we compare an *MxN* LBIC against *M*-port ideal, *M*-port replicated, and 2*M*-bank cache. A quick comparison with Table 3 reveals that the LBIC configurations are either as good as or superior to the comparable multi-ported structures in the table. With the exception of **compress**, the 2x2 LBIC outperforms the 2-port ideal cache. The 2x2 LBIC also performs better than the 2-port replicated cache, and the 4-bank cache. It must be noted that a large 2-port replicated cache costs about twice the 2x2 LBIC in die area. While the 4x4 LBIC, on average, achieves within only 90% of the 4-port ideal cache performance on SPECint, the 4-port ideal cache is fully outperformed when it comes to SPECfp and can only achieve 64% of the 4x4 LBIC's performance on **mgrid,** for example. The 4x4 LBIC also performs slightly better than the 8-bank cache for SPECint (6.102 IPC average for the former compared to 6.005 for the latter) and far better for SPECfp (9.7361 IPC average as opposed to 7.782).

The effectiveness of access combining in the LBIC is particularly apparent from the SPECfp IPC averages in Table 4. For the *M*x*N* configuration, keeping *M* fixed and increasing *N* from 2 to 4 yields an average IPC improvement of 10.3%. On the other hand, when *N* is fixed and *M* is increased from 2 to 4 or from 4 to 8, only an 8.5% average improvement is seen for *N* = 2 and 6.5% for *N* = 4. This clearly shows that the SPECfp programs benefit more from combining than from interleaving. Though our earlier analysis in Section 4 showed the SPECint programs to have high consecutive reference locality in same bank and same line, it appears that the constraints in their program semantics limit the gains from combining. The SPECint programs benefit more from doubling *M* than from doubling *N*.

7. Related Work

With increasing focus on high ILP exploitation, more studies are now dealing with cache bandwidth issues. In the first major study in this area [2], Sohi and Franklin present evaluations to show the relevance of using non-blocking caches to increase bandwidth for superscalar processors. They propose different configurations, combinations and implementations of multi-ported and multi-bank caches as solutions for high bandwidth, but do not provide any evaluations for these designs.

Motivated by multi-porting cost, in die area and access latency, Wilson et al. [7] provide a comprehensive evaluation of several buffering techniques for improving the bandwidth of a single cache port, and Austin and Sohi [8] explore many designs in detail for achieving high bandwidth for TLB devices. These studies concurrently developed access combining, a technique that we extend in our design of the LBIC structure.

Finally, Juan et. al. present results of their work on high bandwidth data caches for superscalar processors in [9]. While they provide much breadth, touching on several designs, our focus has been on more depth in designing the LBIC structure, which is capable of handling multiple accesses per cycle with performance close to ideal multi-porting and economy of cost near traditional multi-banking.

8. Conclusions

As the trend toward exploiting higher levels of parallelism through multiple instruction issue continues, more and more demand will be placed on the data cache memory. With increasing numbers of data items needed from the cache per cycle, the current single- and dual-ported cache implementations threaten to become a major bottleneck for future processors. There is a need therefore for cost-effective cache designs that can handle multiple data accesses simultaneously. In this study, we have explored the performance and limitations of the two commercially-implemented ways of achieving multi-ported caches: multi-porting by replication and multi-banking.

Whereas the multi-banking technique suffers substantial performance degradation due to bank conflicts, multi-porting by replication is die area limited and does not scale well to ideal multi-porting performance due to the need for broadcast stores for coherence. Interestingly, analysis of the memory reference stream reveals that a substantial portion of all conflicts in a multi-bank cache are caused by consecutive references that map into the same cache line of the same cache bank.

Our proposed Locality-Based Interleaved Cache (LBIC) is built on traditional multi-banking and employs limited multi-porting to a single-line buffer per bank to exploit same cache line spatial locality. Our detailed execution driven-simulations suggest that the LBIC approach is a good choice for handling multiple accesses. In particular, the LBIC design scales better toward ideal multi-porting performance than either of the currently available implementations, and is competitively cost-effective. Furthermore, the memory scheduling logic of the load/store queue can be enhanced to further increase the bandwidth delivered by the LBIC.

9. Acknowledgments

The authors at the University of Michigan are grateful to the Intel Corporation for its support through the Intel Technology for Education 2000 grant. Jude Rivers is funded by a University of Michigan Graduate Fellowship. This work was also supported in part by a gift from the IBM Corporation.

Bibliography

[1] Y. N. Patt, S. J. Patel, M. Evers, D. H. Friendly, and J. Stark, "One Billion Transistors, One Uniprocessor, One Chip," *IEEE Computer,* 30(9):51--57, September 1997.

[2] G. S. Sohi and M. Franklin, "High-Bandwidth Data Memory Systems for Superscalar Processors," *Proceedings of ASPLOS-IV*, pp. 53--62, April 1991.

[3] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," *Proceedings of MICRO-29*, pp. 24--34, December 1996.

[4] K. Yeager et. al., "R10000 Superscalar Microprocessor," *Hot Chips VII*, 1995.

[5] Digital Equipment Corporation, Maynard, MA. *Alpha Architecture Handbook*, 1994.

[6] Digital Equipment Corporation, Maynard, MA. *Alpha Architecture Handbook*, 1996.

[7] K. M. Wilson, K. Olukotun, and M. Rosenblum, "Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessors," *Proceedings of ISCA-23*, pp. 147--157, May 1996.

[8] T. M. Austin and G. S. Sohi, "High-Bandwidth Address Translation for Multiple-Issue Processors," *Proceedings of ISCA-23*, May 1996.

[9] T. Juan, J. J. Navarro, and O. Temam, "Data Caches for Superscalar Processors," *Proceedings of ICS*, July 1997.

[10] W. Oed and O. Lange, "On the Effective Bandwidth of Interleaved Memories in Vector Systems," *IEEE Transactions on Computers*, C-34(10):949--957, October 1985.

[11] B. R. Rau, "Pseudo-Random Interleaved Memory," *Proceedings of ISCA-18*, June 1991.

[12] J. E. Smith, "Decoupled Access/Execute Computer Architectures," *Proceedings of ISCA-9,* June 1982.

[13] G. S. Tyson, Evaluation of a Scalable Decoupled Microprocessor Design, Ph.D. Dissertation, University of California at Davis, January 1997.

[14] Microprocessor Report, Vol. 7, No. 13. *IBM Regains Performance Lead with Power2,* October 1993.

[15] G. Lesartre and D. Hunt, "PA-8500: The Continuing Evolution of the PA-8000 Family," *Proceedings of COMPCON'97*, March 1997.

[16] Microprocessor Report, Vol. 9, No. 17. *Intel Boosts Pentium Pro to 200 MHz*, November 1995.

[17] D. Burger and T. M. Austin, "Evaluating Future Microprocessors: the SimpleScalar Tool Set," Tech. Report #1342, University of Wisconsin, June 1997.

[18] D. Hunt, "Advanced Performance Features of the 64-bit PA-8000," *Proceedings of COMPCON'95*, pp. 123--128, March 1995.

[19] S. W. Melvin, M. C. Shebanow, and Y. N. Patt, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines," *Proceedings of MICRO-21*, pp. 60--66, December 1988.

[20] M. Franklin and M. Smotherman, "A Fill-Unit Approach to Multiple Instruction Issue," *Proceedings of MICRO-27*, pp. 162--171, November 1994.

[21] G. S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," *IEEE Transactions on Computers*, 39(3):349--359, March 1990.

[22] K. Kitai, T. Isobe, T. Sakakibara, S. Yazawa, Y. Tamaki, T. Tanaka, and K. Ishii, "Distributed Storage Control Unit for the Hitachi S3800 Multivector Supercomputer," *Proceeding of ICS*, pp. 1--10, July 1994.