Caching Values in the Load Store Queue

Dan Nicolaescu, Alex Veidenbaum, Alex Nicolau
Department of Computer Science
University of California, Irvine
{dann,alexv,nicolau}@ics.uci.edu

Abstract

The latency of an L1 data cache continues to grow with increasing clock frequency, cache size and associativity. The increased latency is an important source of performance loss in high-performance processors. This paper proposes to cache data utilizing the Load-Store Queue (LSQ) hardware and data paths. Using very little additional hardware this inexpensive cache improves performance and reduces energy consumption. The modified Load/Store Queue "caches" all previously accessed data values going beyond existing store-to-load forwarding techniques. Both load and store data are placed in the LSQ and is retained there after a corresponding memory access instruction has been committed. It is shown that a 128-entry modified LSQ design allows an average of 51% of all loads in the SpecINT2000 benchmarks to get their data from the LSQ. Up to 7% performance improvement is achieved on SPECInt2000 with a 1-cycle LSQ access latency and 3-cycle L1 cache latency. The average speedup is over 4%.

1. Introduction

Modern wide issue out–of–order processors use higher and higher clock frequencies in order to increase performance. Memory system performance is very important in such systems and they typically support a deep memory hierarchy – three levels of cache is not uncommon today. The L1 data cache is a large set associative cache with a number of independent read/write ports. Such L1 caches have a growing latency due to high frequency, communication delays and logic complexity leading to increased load to use latency. For example Alpha 21264 [9] has 3 and 4 cycles load to use latency for integer and floating point loads, respectively. The latencies are 2 and 6 cycles for Intel Pentium4 [8]. The latencies more than double with a doubling of the clock frequency [19].

High load to use latency is detrimental to performance because the instructions depending on the load cannot be executed until the load completes. Aggressive out–of–order instruction issue hides part of the L1 latency, but cannot mask all of it. It is not easy to quantify the effect of longer L1 latency on performance. One rule of thumb from the Intel PentiumPro [17] designers was that an additional cycle of load latency costs about 1/2 of what it costs in a strict in-order architecture. They also observed that 15% of micro-ops that are critical–path loads take an extra clock cycle to complete when the latency is increased by a cycle. Thus a lower-latency L1 is quite desirable. However, this typically means a smaller or simpler L1 cache that has higher miss rates.

The latency vs miss rate trade-off is resolved differently depending on a system type – desktop or server. Most system designers opt for a larger L1 at the expense of latency, e.g. the Alpha 21264 and the AMD Athlon[1]. The extra latency is deemed acceptable. The Alpha designers reportedly estimated a 4% performance penalty from an extra cycle of cache latency [7], while PentiumPro designers estimated it to be 7.5%.

One possible solution that would decrease the latency is to add another level to the hierarchy, an L0 cache. This has been proposed in the context of reducing energy consumption[10], but was actually shown to lead to a performance loss. The performance loss stems primarily from an additional latency on L0 misses although it was not evaluated for out–of–order processors. It may be possible to find an L0 size that would reduce the average L1 latency, but it depends on the respective L0 and L1 latencies and miss rates and thus will not work for all programs. And the addition of an extra cache level would significantly increase the complexity of an already complex part of the CPU.

This paper proposes a design that provides the benefit of a decreased load to use latency of the L0 cache without the associated potential performance loss. This is accomplished with little additional hardware and with much lower complexity than a true L0 cache. The key observation is that the load/store queue(s) in the CPU already contains a tag structure similar to that of a cache and a tag comparison

Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)
1526-7539/04 $20.00 © 2004 **IEEE**

is performed on each load and store instruction to guarantee correct memory access ordering. This tag compare can be used as part of an L0-like access to avoid the latency increase typical of the L0 miss. What's more, the LSQ data paths are sufficient to implement an L0-like cache.

The LSQ designs in existing out–of–order processors mentioned above already perform a cache-like function when they implementation a store–to–load forwarding. The approach proposed in this paper goes beyond that in two important ways. First, it "caches" load data when a memory load completes and allows such cached values to be reused. Second, it keeps the data in an LSQ entry even after the corresponding instruction retires and until the LSQ space is needed. This is accomplished without the complexity of a true L0 cache and leads to a 5.5% average performance improvement on SpecINT.

The remainder of this paper is organized as follows. The related work is presented next, followed by a description of the functioning of a base LSQ design and that of an experimental setup used for evaluation. The proposed design is presented next with experimental results and possible improvements to the design and a comparison with the performance of a memory hierarchy containing a small L0 cache is presented to show some of the potential benefits of the approach as well as to demonstrate that a traditional L0 design has a problem.

2. Load/Store Queue Operation

An out-of-order processor can issue load and store instructions to the data cache in an order different from the program order. A load/store queue is the hardware structure that keeps loads and stores from issue until retirement. A memory access instruction waits in the LSQ until its address (and data for stores) is computed. In case of a load, the memory access order is checked and the load sent to cache. In case of a store, it is buffered in the LSQ until it is ready to retire and then written to the cache non-speculatively. The above is a very general description of the LSQ operation. A variety of possible implementations are possible. For instance, it can be implemented as separate load and store queues as in Pentium4 or Alpha 21264. Alternatively, a single queue structure can be used as in AMD's Athlon [6].

In all cases, the instructions are inserted in the queue(s) in program order and the correct memory access ordering is maintained. This cannot be done any earlier in the instruction pipeline as memory addresses only become known at this point. At the same time loads need to be sent to the cache as soon as possible to improve performance. In general, read–after–write, write–after–write, write-after–read, and read–after–read dependencies are all possible among memory accesses. Except for the read–after–read these dependencies need to be enforced. It is accomplished by using a CAM with an address tag in each LSQ each entry. This way a new address can be quickly compared to address tags of all pending entries.

Let us assume a single queue. In this case the program order of memory access is the same as the queue order. The way to achieve both ordering and performance in this case is to 1) send stores to cache in order, and 2) to guarantee that a load does not get sent to cache if any of the earlier stores overlap with its address. The latter implies that the load waits for all prior stores to have their addresses computed. Loads can still bypass earlier loads and even earlier stores once address independence is established.

A degree of speculation is possible in sending requests to cache. This allows a load to bypass an earlier store(s) prior to their address resolution. The load and all subsequent instructions are cancelled if a dependence is detected once the store address is known. This can be expensive. The Alpha 21264 uses this type of speculation but implements a predictor to reduce the frequency of incorrect load speculation. The 21264 also uses separate queues making things even more complicated. For simplicity, the rest of this paper will use a single queue design without speculative loads, as shown in Figure 1.

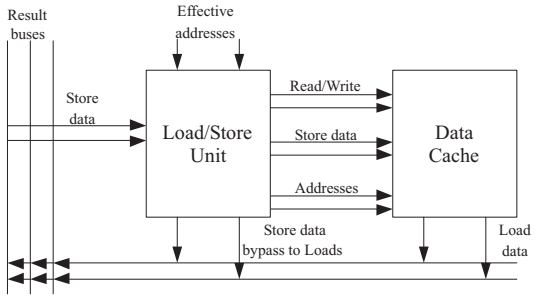

Figure 1: Load/Store Queue implementation in a dual–ported cache system

This queue design is similar to that used in the AMD K7 [6]. The LSQ is connected to a dual–ported cache and to result buses. The LSQ inputs are Load/Store addresses and data for store instructions. The LSQ outputs are addresses and store data for the data cache. It is also connected to the result busses in order to implement store–to–load forwarding. This LSQ organization will be called the *"base LSQ"*.

Store–to–load forwarding is invoked in the base LSQ when an address tag check detects a prior store with the same address. The corresponding data from the queue forwarded from the LSQ and the cache access is avoided. The LSQ size of a modern processor is growing to help explore the instruction level parallelism (ILP). For example, the Alpha 21264 has a total of 64 LSQ entries. This increases the forwarding opportunities from the LSQ. However, once a store retires its data is no longer available for forwarding.

1526-7539/04 $20.00 © 2004 **IEEE**

2.1. Related Work

Various techniques for reducing the cache access latency have been proposed, including pure hardware solutions, software/compiler approaches, or mixed approaches. Only some of these approaches, the most closely related to the ideas presented in this paper, are briefly discussed in this section.

The concept of load redundancy and how it can be exploited was presented in [4], [20] and [15]. It was shown that a significant number of load instructions access the same data in a short time interval. This type of locality can also be exploited by the design described in this paper with a smaller hardware budget, and fewer changes to the CPU design.

The concept of value locality, a high probability that a previously seen value will be used again soon, is used for load value prediction [13]. A load value predictor predicts the data read by a load instruction and avoids waiting for it to be returned by the cache, which is a longer latency operation. A predictor involves adding logic for dealing with mis–predictions and suffers from latencies in case of mis–predictions.

Silent stores [3],[12], [11] are store instructions that store to a location the same data that is already there. By eliminating silent stores, the memory traffic is decreased, allowing other memory accesses to proceed faster, and increasing performance. The approach presented in this paper deals with load instructions, which are more frequently executed, and have instructions depending on their result.

A filter cache [10] is a small and fast L0 cache. The L0 cache has a lower latency than an L1 cache on hits, but in case of a miss the latency is higher. It has a higher miss rate due to its small size. The filter cache is an approach used mostly for reducing the energy consumption because it can decrease the processor performance.

Another technique for reducing the load to use latency is presented in [2]. A pipeline organization is proposed that allows load instructions to complete prior to reaching the execute stage of the pipeline, allowing for an important speedup.

A read–after–read memory dependence prediction is presented in [14]. By converting series of load–use–load–use chains to a single load–use–use chain memory accesses can be eliminated to increase the program execution speed.

This paper differs from previous work by trying to improve the average cache access latency with only small modifications of the processor pipeline. It adds little extra hardware and complexity. At the same time, it is not using prediction and thus it incurs no mis–prediction penalty. Nor does it suffer the latency increase of an extra level of cache.

3. The *Cached* LSQ

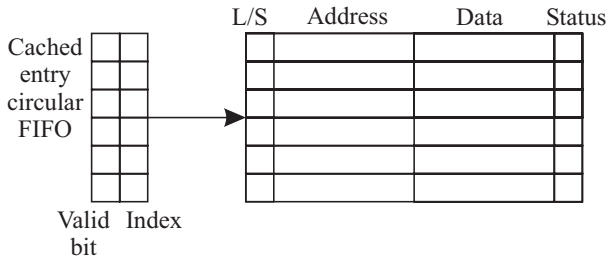

Figure 2: Cached Load/Store Queue

The LSQ design proposed in this paper and its differences from previously proposed LSQ designs are described in this section. Figure 2 shows the additional details of the LSQ entry organization. The *L/S* flag identifies the instruction as being a load or a store. The *"address"* contains a memory address to access, the *"data"* contains the data to store for store instructions, and *"status"* contains the execution state of the instruction (in progress, not issued, etc.). The addresses are stored in a CAM. The data part of the entry is maintained in a RAM.

It is clear that the LSQ is quite similar to a cache with a single-word (8 byte) block. The data paths connecting the LSQ to the rest of the system are the same as for the L1 cache. Access to the LSQ is also similar to a cache access: first a tag search, then possibly data forwarding (from store entries). The main difference is that the LSQ does not keep the data after the instruction retires nor does it keep the load data fetched by a load access. The approach proposed here adds the capabilities to perform both of these actions.

The modified LSQ design proposed in this paper introduces a new state for the LSQ entries. This state, called *"cached"* allows an entry for a completed instruction to be retained in the LSQ together with its data. Using the existing address tag search in the LSQ, a "cached" entry for an address can be detected with no additional time. Once detected, the data in such an entry can be forwarded to subsequent loads using the same mechanism as for store–to–load forwarding. The new design will be called a *Cached Load/Store Queue (CLSQ)*.

The *base LSQ* does not store load data in the entry. The CLSQ design changes that and stores the load data in the *"data"* part of a corresponding entry. This is accomplished by writing data into the LSQ as it is moved from the cache to the CPU over one of the result busses. As a result, both loads and stores now have data in their LSQ entries. This allows another type of forwarding: from load instructions to subsequent loads. The CLSQ design thus allows load instructions to get their data form either load or store instructions that have retired but with entries that are cached in the LSQ.

1526-7539/04 $20.00 © 2004 **IEEE**

A load in the *base LSQ* did not use the *"data"* field. The register file read the requested data directly from a result bus. The CLSQ design captures the load data and stores it in the CLSQ from the result bus at the same time as the register file. This is done utilizing the *base LSQ* input ports from a result bus used by store instructions. The LSQ port connection to a result bus is unused in this cycle. Thus no data path changes are required, only the control logic of the LSQ needs to be changed.

The CLSQ keeps entries after the corresponding instruction has retired and until the LSQ space is needed for new instructions. In order to distinguish the LSQ entries for retired instructions from other entries, the *"status"* field in the LSQ is augmented with a new, *"cached"* state. Additional control logic keeps track of the entries in *"cached"* state and allows them to be released when LSQ space is needed.

The design used in this paper is a circular FIFO that contains an index of the *"cached"* entry and a valid bit. When a load or store instruction is put in the *"cached"* state its LSQ index is pushed in the *"cached"* entry FIFO and marked as valid there. When a new instruction arrives in the LSQ and no entries are free, the oldest entry in *"cached"* state (as pointed to by the FIFO head) is used to store the new instruction. The entry is removed from the FIFO at the same time. Notice that the FIFO is only accessed when cached entries are added/removed from the LSQ.

On a store instruction, all entries in the *"cached"* state with an address matching the store are invalidated. This prevents multiple entries with the same address but possibly different data from being in the cached LSQ. No corresponding invalidation of *"cached"* entries is needed on load instructions. An additional coherence problem exists with respect to the L1 cache. An entry in the CLSQ may become "stale" with respect to the rest of the memory hierarchy. For instance, this would happen if a coherence request invalidated the data in L1. One approach for dealing with this problem is to invalidate a cached entry any time a corresponding coherence action occurs in the L1 cache. The inclusion principle can be extended to the CLSQ with the L1 cache maintaining the information. This will avoid unnecessary invalidation of CLSQ cached entries.

The CLSQ is virtually addressed, so in case of context switches its contents might need to be invalidated. The solutions usually applied to virtually indexed caches to avoid invalidations after context switches can be applied to the CLSQ too. It would also be possible to invalidate the contents of the CLSQ, its size is very small compared its access frequency.

In terms of timing, the LSQ address tag compare is performed when a new address becomes available. It will be assumed to take one cycle and this cycle is already accounted for in the base LSQ access. The L1 cache access starts in the next cycle, at the earliest. In the CLSQ case, therefore, the miss signal for the CLSQ is available early enough to avoid starting the L1 cache access. The same is true for the store–to–load forwarding in the base LSQ. Also, a conservative assumption is made that a data access in the CLSQ and the base LSQ is completed one cycle after the address compare is finished.

4. Experimental setup

The CLSQ was evaluated by implementing the proposed changes in the SimpleScalar–3.0c simulator [5] modified to support longer pipelines. An aggressive 64–bit high performance architecture was modeled. The SPECINT2000 benchmark suite was simulated (see Table 2). The benchmarks were compiled with the -O4 flag using the Compaq compiler targeted for the Alpha 21264 processor. The benchmarks were fast forwarded for 500 million instruction, then fully simulated for 5 billion instructions. For the SPECFP2000 benchmarks, although the hit rate in the CLSQ is significant the performance improvement is negligible due to the high L1 miss rate of those benchmarks, so they are not shown in this paper.

L1 Icache	32KB, 2 way, 64 byte/line, 1 cycle
L1 Dcache	32KB, 2 way, 64 byte/line, 3 cycle, 2 R/W ports
L2 cache	2MB, 8 way, 64 byte/line, 20 cycle
Issue	4 way out-of-order
Branch predictor	64K entry g–share, 4K-entry BTB
Reorder buffer	256 entry
Load/Store Queue	128 or 256 entry
Scalar Arithmetic	4 integer, 4 floating point units
Complex Units	2 INT, 2 FP multiply/divide units
Pipeline	15 stages

Table 1: Processor configuration

The base LSQ is a 128-entry single queue with store–to–load forwarding on pending entries. The data access latency for both the base LSQ and the CLSQ is one cycle. The memory hierarchy latencies are 3/20/200 cycles for the L1/L2/memory accesses, respectively. Other details of the processor and system model are given in Table 1. The CMOS process parameters for the simulated architecture were a 1.5GHz clock and a .10μm feature size.

4.1. Load NOPs

An interesting problem occurs in dealing with *nops*. The Alpha *"Universal NOP" – unop* instruction is encoded as a type of load (*ldq_u*) with the register hardwired to zero ($31) as the destination. The architecture specification allows for *unops* to be eliminated early in the pipeline. The instruction decoder needs to detect that the *unop* instruction is not a

load, even though it has the opcode of a load. The simulator should not perform a cache access for such an instruction or put it in the LSQ. This was not done by default in the original version of the simulator.

The figures 3 and 4 are comparison results for a simulator that properly detects *unops* with one that does not. Figure 3 shows that on average only 85% of the L1 data cache accesses are actually loads and stores, the rest are mis-interpreted *unops*. This would have an important impact for studies of the L1 energy consumption: for instance the dynamic energy consumed by the DL1 cache would be off by 40% for the gcc benchmark. The number of DL1 misses does not change for most benchmarks, but given that the number of accesses decreases, the miss rate increases. The number of misses changes for *perl* due to mis-speculated loads. The benchmark spends a significant amount of time in a small, data dependent, loop (a regular expression matcher).

Figure 4 shows that when properly detecting *unops* the IPC increases on average by 4%, but it can increase by 15% for *gcc*. The IPC improvement is due to resolving NOPs faster in the pipeline than loads. Loads occupy resources like: entries in the reorder buffer and LSQ, cache ports, have a long latency and they need to be committed in program order. The machine used in these simulations is resource rich, the impact on IPC would be even higher on a more resource constrained processor.

In the simulations in this paper the *unop* instructions are treated like proper NOPs, the do not perform a memory access.

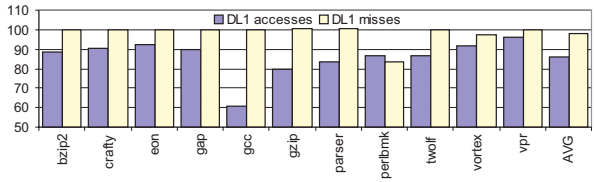

Figure 3: % DL1 accesses and misses of a simulator that properly detects *UNOPs* compared to one that does not

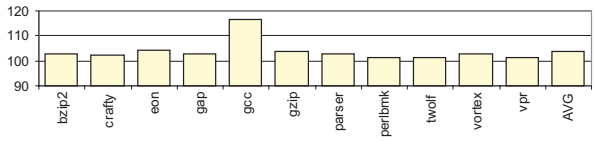

Figure 4: % IPC change for a simulator that properly detects *UNOPs* compared to one that does not

5. Experimental results

The CLSQ is basically a cache and as such can be evaluated using standard cache metrics such as hit rate. As a cache, its performance is largely determined by its size and the line size since it is fully associative (the impact of the line size is shown in Section 5.1. The size of this cache is a function of the number of valid entries in the base LSQ. Let us therefore start by investigating the average base LSQ occupancy and then look at the CLSQ hit rates.

bzip2	Compression
crafty	Game Playing: Chess
eon	Computer visualization
gap	Group theory
gcc	C optimizing compiler
gzip	Compression
parser	Word processing
perlbmk	PERL interpreter
twolf	Place and route simulator
vortex	Object–oriented database
vpr	FPGA circuit placement and routing

Table 2: Benchmarks and descriptions

Figure 5 shows the percentage of time when the base 128-entry LSQ was full, an average of 2.1% of the time. Only for the "gcc" benchmark the LSQ was full a significant amount of time: 15%. On average, therefore, a 128-entry CLSQ will be equivalent to a 128-line L0 cache with 8B lines.

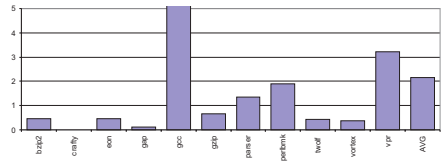

Figure 5: % of time the 128–entry base LSQ was full

Next, consider the CLSQ hit rate. Figure 6 shows how many load instructions hit only on entries in the "Cached" state. For integer benchmarks the average hit rate is 51% for a 128-entry CLSQ and 45% for a 64–entry one. While low by normal cache standards this is very good for a small L0-like cache. In addition, these rates do not include LSQ hits due to store–to–load forwarding. The significant number of hits in the CLSQ shows that the proposed design is an effective way to exploit locality.

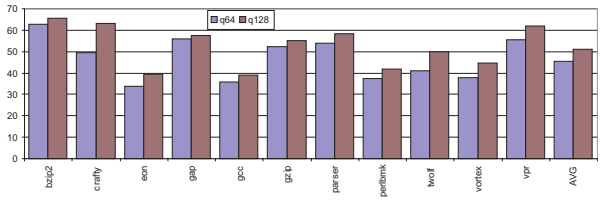

Figure 6: Load instructions that hit in the CLSQ

The CLSQ hits on "cached" entries occur in addition to the the base LSQ hits due to store–to–load forwarding. The latter are also exploited by the CLSQ but are shown separately since they are not unique to the CLSQ. Figure 7

shows the load hit rates in the base LSQ due to store–to–load forwarding. They are 6.3% for the 128-entry LSQ. This is significantly lower than the CLSQ hits on cached entries. The difference can be explained in large part by a much larger number of entries in the "Cached" state compared to the number of pending stores and by a shorter lifetime of such stores in the LSQ. The CLSQ hits due to store–to–load forwarding are not shown here, but they are within 0.2% of base LSQ.

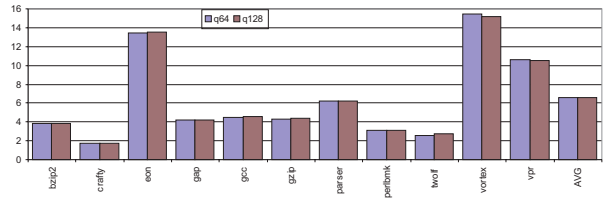

Figure 7: Load instructions that hit in the base LSQ (ie store–to–load forwarding)

One can ask if it is worth adding the load data to the CLSQ since this requires modification of the base LSQ design, especially if separate Load/Store queues are used. The answer is that, on average, over 50% of all the hits in the CLSQ are on data from load instructions (not shown here for space reasons). This justifies modifications to keep the load data in the CLSQ.

The use of the CLSQ should increase the execution speed due to reduced Load–to–use latency (1 cycle vs 3 cycles for L1 hits). Figure 8 shows the speedup of a system using the CLSQ in comparison with a classic LSQ. For the 64–entry CLSQ the speedup is on average 4.04% for a 32–entry and 4.09% for a 128-entry CLSQ. The speedup increases slightly with the size of the CLSQ as expected.

The difference in benchmark performance is clear if one looks at the average time spent in the base LSQ (Figure 9). For integer benchmarks the average time is around 50 cycles (and increases with the LSQ size).

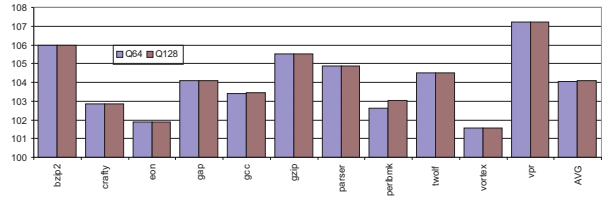

Figure 8: Speedup when using a CLSQ

Another effect of load–to–use latency reduction due to the CLSQ is that the total number of executed instructions is reduced, as can be seen in Figure 10. This effect is due to a faster branch resolution. For branches dependent on load instructions the CLSQ provides data to resolve the branch condition faster. When this happens for incorrectly predicted branches the result is fewer speculatively issued instructions.

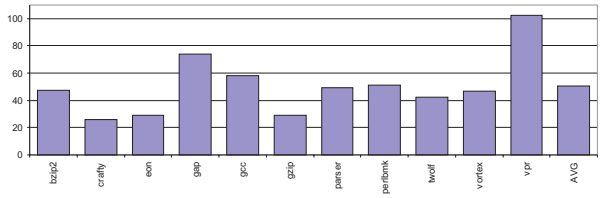

Figure 9: Average LSQ latency in cycles

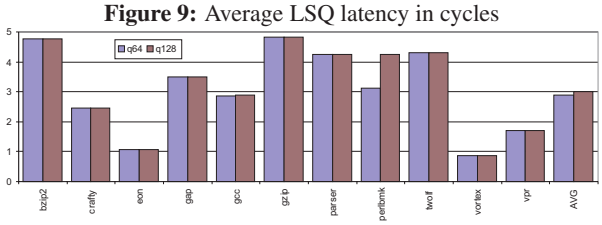

Figure 10: Reduction in the number of instructions executed when using a CLSQ

5.1. Exploiting Spatial Locality in the CLSQ

The CLSQ design described so far exploits primarily the temporal locality of data access. The only exception is the 32–bit integer benchmarks since the CLSQ data is 64 bits wide. It is well known that spatial locality is very important for good cache performance. This section describes the CLSQ design changes necessary and explores the effect of spatial locality on the CLSQ. Wide LSQ entries were also mentioned in [11] and [16].

The CLSQ can exploit spatial locality by increasing the size of the data field and modifying the address tag search mechanism. This is preferable to creating a separate CLSQ entry for the additional word(s) since increasing the CLSQ size will slow down the address tag comparison process. In effect, the design proposed here is a sectored CLSQ with a single address tag plus additional "valid" bits per word. It is called a *"Wide CLSQ"* (WCLSQ). The new address tag match process is illustrated in Figure 11.

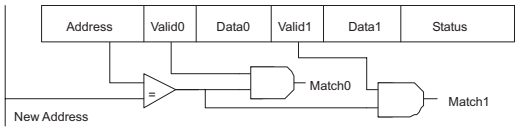

Figure 11: WLSQ entry structure and "match" logic

Increasing the LSQ line size may increase the complexity of the data fetch from the L1 cache. This is because additional words of the CLSQ "cache line" need to be fetched. One approach for doing this would be to treat this as prefetch. A separate queue for these "extra word" requests can be maintained and they can be scheduled when there are no "regular" fetch requests. This gives the extra words lower priority than access to CPU requested addresses but higher priority than stores. This approach does not require modifications to the L1 cache. It should work well if cache ports are sufficiently free. Figure 12 shows

1526-7539/04 $20.00 © 2004 **IEEE**

the average cache port utilization to be less than 40%, on average. It is low enough to make this approach feasible.

Another approach that appears simpler but requires changes to the L1 cache is to increase the fetch width from the L1 cache. This approach would deliver 2x or more words per request and work well with the wider CLSQ. This is the approach simulated in this paper. A 16–byte WCLSQ line, double the original 8–byte line size, is used in order to keep the system complexity under control. The bus width between the L1 cache and the WCLSQ is also changed so that the entire WCLSQ line can be fetched and moved in the same three cycles of the L1 cache access. The hit access time to the data part of the WCLSQ is still assumed to be one cycle.

Loads and stores are processed differently by the WCLSQ. For a load a 16B line is fetched from the L1 cache. For a store, only the 8-byte part of the line that is addressed by the instruction is written in the WCLSQ (8B access alignment is assumed). The same happens when writing the data to the L1 cache (although write combining is possible at this point as implemented in many cache systems). As a result, there is more data cached in the WCLSQ from load instructions than from store instructions.

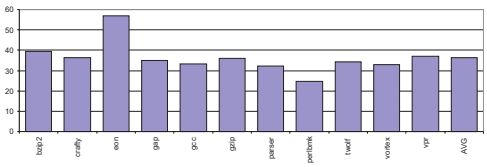

Figure 12: Average cache port utilization

Figure 13 compares the average number of loads that hit in WCLSQs with 16B and 32B lines relative to the hit rate for CLSQ with an 8B line. The number of entries is 128 in all cases. The average number of hits increases by 16% for a 16B CLSQ and by more than 26% for ad 32B CLSQ. The average speedups for the CLSQ and WCLSQ are shown in Figure 14. The speedup increases by about 2%.

Overall, the wide CLSQ delivers a significant increase in the load hit rate for all benchmarks. This is still low by data cache standards, however the overall cache size of the CLSQ is quite small. A larger LSQ, as postulated for some of the future designs, will perform better. It should also be remembered that the LSQ size determines its access time.

The 128-entry LSQ was at the limit of a 1-cycle access for the system modeled here. This was determined using the CACTI [18] model of the CLSQ for the parameters used in this paper. For a 3-cycle L1 cache access it was not considered worthwhile to have a 2-cycle CLSQ. Again, future systems will have slower L1 cache access and will make the CLSQ perform better.

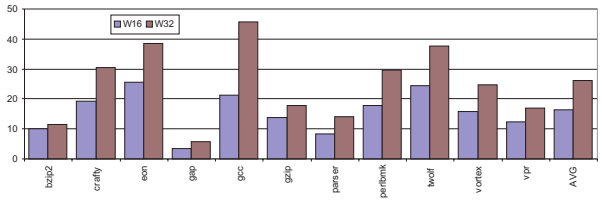

Figure 13: Load instruction hit rates in WCLSQ relative to CLSQ

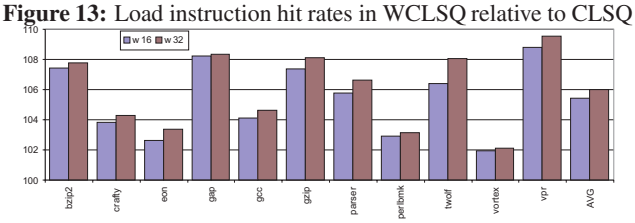

Figure 14: Speedups for CSLQ and WCLSQ

5.2. Impact Of Additional Resources

Future high-performance microprocessors will have more resources to better support instruction level parallelism. These will include a larger reorder buffer, L2 cache, and branch predictor as well as more functional units. Combined with a longer L1 cache latency these will make memory access more of a bottleneck and will make the use of the CLSQ more important because it will avoid the increased memory access latencies.

5.3. Reducing the number of cache ports

Considering the CLSQ avoids cache accesses, it can be envisioned that the CLSQ could allow for a decrease in the number of cache ports. Reducing the number of ports has a significant impact on performance. Figure 15 shows that the performance of the baseline system can decrease by up to 25% by using a single ported cache instead of a dual ported one.

It can be possible to build a system that would just access the CLSQ instead of the cache for some instructions, hence allowing 2 memory access instruction to be issued in parallel in a system with a single ported cache: one to the CLSQ and another one to the cache. Figure 16 shows the relative performance of a system with a single ported cache and a CLSQ that can be accessed simultaneously with the cache with a system with a dual ported cache. Some benchmarks have slowdowns of up to 12%, but on the average

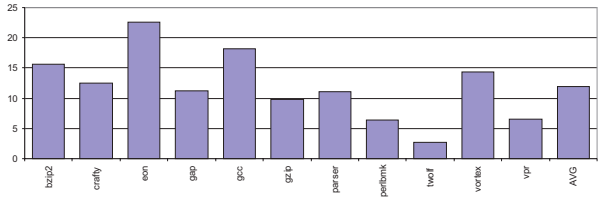

Figure 15: 1–port cache vs a 2–port cache slowdown

1526-7539/04 $20.00 © 2004 **IEEE**

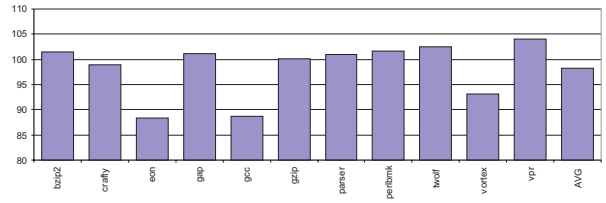

Figure 16: CLSQ with a 1–port cache vs a 2–port cache

the performance of the single ported cache CLSQ system is within 98% of the performance of the dual ported cache system. For such a CLSQ system the energy consumption of the cache would be much smaller. This can be important in the future, for higher issue width processors building fast multi-ported caches is non trivial, using a CLSQ to reduce the number of ports can be useful.

6 Conclusion

This paper presented the design of the CLSQ, a cached LSQ that allows data from both loads and stores to be retained in the LSQ entries after the corresponding instruction retires. The CLSQ allows the retained data to be reused by future load instructions. The access time to the data in CLSQ is faster than in the L1 cache and reduces the load–to–use latency for dependent instructions. This also leads to a reduction in the number of mis-speculated instructions when a branch condition depends of the load value.

The paper shows that the number of load instructions that hit in the CLSQ is significant. It is 51%, on average, for SPECInt benchmarks and leads to a small performance improvement that can be obtained almost for free. Floating point benchmarks have a lower hit rate and a much higher average load latency due to high L1 miss rates and do not benefit as much from the CLSQ. A CLSQ design with a longer line size but the same number of entries was also considered. It was found to further improve performance. The possibility to use a CLSQ to reduce the number of cache ports was also considered, this will have a bigger impact in the future for wider issue machines that would need more that 4 cache ports.

The proposed modifications to the LSQ are small and do not have a negative impact on performance. The CLSQ design makes use of existing hardware resources in the CPU and is thus very inexpensive. The cycle time is not changed. This makes it comparable to an L0 cache, which has a higher complexity. In addition, by using the address tag compare of the LSQ the CLSQ avoids an additional latency that might be incurred in the use of an L0 cache.

The CLSQ latency of one cycle considered here was only so much smaller than the three cycle L1 latency. The expected increase in cache latency for future processors as well as larger future LSQ sizes will further increase the speed–up obtainable by using the CLSQ.

Future extensions of this work include studying allowing hits in the LSQ on pending load instructions and the effect of squashing silent store that can be detected at the LSQ level by comparing the store data with the contents of the CLSQ.

References

[1] AMD Athlon Processor, x86 Code Optimization Guide. `http://www.amd.com/products/cpg/athlon/techdocs/pdf/22007.pdf`.

[2] T. M. Austin and G. S. Sohi. Zero-cycle loads: Microarchitecture support for reducing load latency. In *MICRO*, pages 82–92, 1995.

[3] G. B. Bell, K. M. Lepak, and M. H. Lipasti. Characterization of silent stores. In *PACT*, pages 133–142, October 2000.

[4] R. Bodik, R. Gupta, and M. L. Soffa. Load-reuse analysis: Design and evaluation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 64–76, 1999.

[5] D. Burger and T. M. Austin. The SimpleScalar tool set. TR-97-1342, University of Wisconsin-Madison, 1997.

[6] K. Diefendorff. K7 challenges Intel. *Microprocessor Report*, 12(14):1–7, Oct. 1998.

[7] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14):1–6, Oct. 1996.

[8] G. Hinton, and et all. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 4(Q1):13, Feb. 2001.

[9] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, Mar./Apr. 1999.

[10] J. Kin, M. Gupta, and W. H. Mangione-Smith. The filter cache: An energy efficient memory structure. In *MICRO*, pages 184–193, 1997.

[11] K. M. Lepak. Silent stores for free: Reducing the cost of store verification. Master's thesis, University of Wisconsin–Madison, 2000.

[12] K. M. Lepak and M. H. Lipasti. On the value locality of store instructions. In *ISCA*, pages 182–191, 2000.

[13] J. P. S. Mikko H. Lipasti, Christopher B. Wilkerson. Value locality and load value prediction. In *ASPLOS VII*, 1996.

[14] A. Moshovos and G. S. Sohi. Read-after-read memory dependence prediction. In *MICRO*, 1999.

[15] S. Onder and R. Gupta. Load and store reuse using register file contents. In *ICS*, pages 289–302, 2001.

[16] A. Pajuelo, A. Gonzalez, and M. Valero. Speculative dynamic vectorization. In *MICRO*, pages 271–280,, 2002.

[17] D. B. Papworth. Tuning the Pentium Pro microarchitecture. *IEEE Micro*, 16(2):8–15, 1996.

[18] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical report, DEC, 2001.

[19] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *ISCA*, pages 25–34, 2002.

[20] J. Yang and R. Gupta. Load redundancy removal through instruction reuse. In *ICPP*, pages 61–68, 2000.

[21] J. Yang and R. Gupta. Energy-efficient load and store reuse. In *ISLPED*, pages 72–75, 2001.

1526-7539/04 $20.00 © 2004 **IEEE**