Reducing Data Cache Energy Consumption via Cached Load/Store Queue

Dan Nicolaescu, Alex Veidenbaum, Alex Nicolau
Center for Embedded Computer Systems
University of California, Irvine
{dann,alexv,nicolau}@cecs.uci.edu

ABSTRACT

High-performance processors use a large set–associative L1 data cache with multiple ports. As clock speeds and size increase such a cache consumes a significant percentage of the total processor energy. This paper proposes a method of saving energy by reducing the number of data cache accesses. It does so by modifying the Load/Store Queue design to allow "caching" of previously accessed data values on both loads and stores after the corresponding memory access instruction has been committed. It is shown that a 32-entry modified LSQ design allows an average of 38.5% of the loads in the SpecINT95 benchmarks and 18.9% in the SpecFP95 benchmarks to get their data from the LSQ. The reduction in the number of L1 cache accesses results in up to a 40% reduction in the L1 data cache energy consumption and in an up to a 16% improvement in the energy–delay product while requiring almost no additional hardware or complex control logic.

Categories and Subject Descriptors

C.1.1 [**Processor Architecture**]: Load/Store Queue, Cache

General Terms

Design, Performance

Keywords

low power, low energy, memory, cache, LSQ, load queue, store queue, low latency

1. INTRODUCTION

The data cache energy consumption of modern wide issue out–of–order processors is growing due to increasing clock frequencies, support for higher degrees of associativity and an increased number of read/write ports. L1 data cache energy consumption can amount to 15% of the total processor energy consumption. Thus it is important to minimize it. Modern processor caches also have an increased load to use latency. For example Alpha 21264 [8] has 3 and 4 cycle load to use latency for integer and floating point loads, respectively. The latencies are 2 and 6 cycles for Intel Pentium4 [6]. High load to use latencies increase program execution time and the energy–delay product. Aggressive out–of–order processor designs can mask some of the L1 latency, but not all of it.

A decrease in the L1 data cache energy consumption and the energy–delay product can be achieved by reducing the number of L1 data cache accesses and by reducing the effective cache access latency. This paper proposes a technique to reduce the data cache energy consumption and to decrease the load to use latency. The technique uses resources already present in the processor, adding very little extra hardware.

The size of the LSQ has grown significantly in modern processors due to the need to better exploit instruction level parallelism. However, the average LSQ occupancy is not very high. This leads to the availability of the storage space in the unoccupied LSQ entries. The technique proposed in this paper exploits this temporarily unused storage space in the LSQ to store data values. This allows load instructions to *"hit"* in this storage space, thus avoiding a cache access. The energy dissipated in a LSQ access is much lower than that required to access the cache leading to significant energy savings. Actually, only a part of the LSQ access energy is newly spent, the rest is the same as in current designs. The new technique also does not require additional data storage or significantly change the cache access datapath. This avoids increasing the complexity of the already complex memory system of a CPU.

A commonly implemented non–speculative data cache access avoidance and latency reduction technique is store–to–load forwarding. The technique exploits store to load locality in programs. Store instructions waiting in the LSQ have their store data placed in the storage area of the queue when it becomes ready. All subsequent load instructions check their source address against the destination address of all store instructions present in the queue to maintain correct memory access ordering. If a load address is included in the address range that a store instruction covers, the corresponding data from the queue is used by the load instruction without performing a cache access. The queue access is faster than a cache access and the load instruction completes faster.

An LSQ entry is released when a corresponding load or store instruction retires. The design proposed here "caches" load/store data in the LSQ entries after the corresponding

ISLPED'03, August 25–27, 2003, Seoul, Korea.

instruction retires. Hits on such data will decrease the number of load instructions that need to access the data cache and reduce the average load latency. These "cached" LSQ entries are a temporary storage and can be freed if needed for new load/stores. Overall, there can be four types of retained data in the LSQ that can potentially be reused:

1. in–progress store instructions
2. retired store instructions
3. retired load instructions
4. in–progress load instruction.

The first type is already reused in existing systems, e.g. store–to–load forwarding. The last type is not explored in this paper, it is part of the ongoing work. This paper concentrates on reusing the data of retired instructions.

The main contributions of the technique proposed in this paper are:

- it allows data for load instructions to be stored in the LSQ entries
- it utilizes unused LSQ entries for holding previously accessed data
- it allows the LSQ to be used as a cache for data accesses by the CPU
- it shows that this approach leads to significant energy savings

The paper also shows that a significant reduction in execution time can be achieved due to reduced latency of the LSQ cache access as compared to the L1 data cache access. This speedup is due to both reduced load-to-use latency and reduction in the number of mis–speculated instructions. The reduced execution time leads to further improvement in the energy–delay product.

All this is achieved by better exploiting data locality. By allowing more data to be kept and reused in the LSQ, cache accesses are avoided and load instructions complete faster. Avoiding cache accesses reduces the energy consumption of the data cache. Completing load instructions faster speeds up program execution and reduces the energy–delay product. The approach has only minimal impact on system complexity since it largely uses resources already present in the processor.

1.1 Related Work

Techniques for reducing the cache energy consumption have been thoroughly researched in the literature. Various, very diverse techniques have been proposed, ranging from pure hardware solutions, pure software or pure compiler–based approaches, or mixed approaches. The approaches mentioned here are the ones most closely related to the ideas presented in this paper.

A filter cache [9] is a small and fast L0 cache. It has a lower energy consumption and latency for hits that the L1 cache. Due to its small size the filter cache has a high miss rate. Using a filter cache leads to an increase in program execution time due to the increased load latency in the case of a filter cache miss, but the overall energy consumption is decreased.

The concept of load redundancy is presented in [2]. It is shown that a significant number of load instructions access the same data in a short time interval. [14] shows an energy efficient way of removing load redundancy. It does so by adding an extra pipeline stage, extra storage and control logic.

A way–prediction scheme [7] uses a predictor with an entry for each set in a set–associative cache. Only the cache way predicted by the predictor is accessed, thus saving energy. In case of an incorrect prediction the access is replayed, accessing all the cache ways in parallel and resulting in additional energy consumption. The technique presented in [12] determines the precise cache way where the data resides before a data cache access, avoiding mis–prediction penalties in energy and time.

Way prediction for instruction caches was described in [8] and [13], but these mechanisms are not as applicable to D-caches.

A technique for reducing the load to use latency, and thus implicitly improving the energy–delay product, is presented in [1]. 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 [11]. 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.

The use of the LSQ to squash silent stores is explored in [10]. Energy savings can be obtained by avoiding to access memory for silent store instructions. But store instructions are less frequent than load instructions, hence the potential for improvement is smaller.

This paper differs from previous work by trying to achieve data cache energy savings and improvements of the energy–delay product by doing only small modifications of the processor pipeline, adding little extra hardware and complexity and by making better use of temporarily unused storage space in the LSQ.

2. THE CACHED LSQ

The LSQ design proposed in this paper and its differences from previously proposed LSQ designs are briefly described in this section. A more detailed description of an LSQ design can be found in [5].

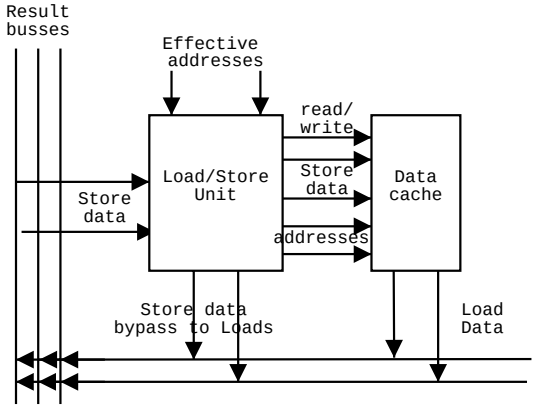

Figure 1: Load/Store Queue and its data paths

Figure 1 shows a Load/Store Queue used in the AMD K7 [5] and its connection to the rest of the system assuming a dual–ported cache. 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. The LSQ organization with a single tagged

queue for both loads and stores (called the *"base LSQ"* from now on) will be used to present the ideas proposed in this paper.

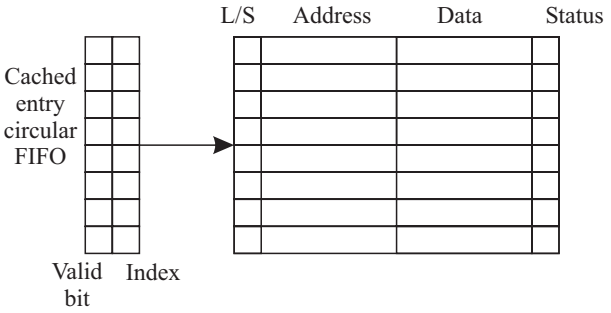

Figure 2: Cached Load/Store Queue

Figure 2 shows the details of the LSQ organization. The *L/S* flag identifies the instruction as being a load or a store. The *"address"* contains a memory address to access, *"data"* contains the data to store for store instructions, and *"status"* contains the execution state of the instruction (in progress, not issued, etc.).

An LSQ entry is allocated when a load or store instruction is issued. At some point the memory address computation completes and the address is entered into the "address" part of the LSQ entry. The addresses are stored in a CAM and are searched to detect load/store dependencies. The data part of the entry is maintained in a RAM. An entire LSQ entry is released when the corresponding instruction retires.

The 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 the same address can be detected with no additional time or energy overhead. Once detected, the data in such an entry can be forwarded to subsequent loads using the same mechanism as existing store–to–load forwarding. The new design proposed in this paper 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.

A load in the *base LSQ* does not use the *"data"* field. The register file reads 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 *"cached"* state their 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. Thus its energy overhead is negligible.

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. A simple approach for dealing with this problem is to invalidate all cached entries any time a coherence action occurs in the L1. The inclusion principle may need to be extended to the CLSQ.

3. EXPERIMENTAL SETUP

The CLSQ energy savings were evaluated by implementing the proposed changes in the Wattch-1.02 [3] simulator. Derived from SimpleScalar [4], Wattch provides detailed energy consumption figures for each major element of an out–of–order processor. The Wattch energy consumption model used in the experiments takes into account leakage energy.

Of the architectures supported by Wattch, the 64–bit Alpha architecture was modeled due to the availability of high quality compilers for this platform. The SPEC95 benchmark suite was used, the benchmarks were compiled with the -O4 flag using the Compaq compiler targeted for the Alpha 21264 processor. The benchmarks were executed to completion using the training input sets. The CMOS process parameters for the simulated architecture were a 1.5GHz clock and a .10μm feature size.

The processor modeled uses a memory and cache organization based on the Alpha 21264 [8]: 64KB dual ported data and instruction L1 caches with 64 byte lines and 3 cycles latency, 512KB 2-way set associative, 12 cycle unified L2 cache, and 80 cycle main memory. The reorder buffer size is 64. The machine is 4–issue, it has 4 integer units, 4 floating point units and one multiplication/division unit. These parameters are kept constant for all the experiments, only the size of the LSQ is varied.

The Alpha *"Universal NOP"* is encoded as a load having as destination the register hardwired to zero. The SimpleScalar/Wattch simulator does not recognize this instruction as a NOP, treating it as a load. This has an effect of increasing cache traffic from 2 to 33% for SpecINT benchmarks. SimpleScalar also contains an error in dealing with the number of cache read/write ports: it is possible to have twice as many cache accesses in a cycle than the number of

cache ports. These issues were corrected in the simulator used for this paper.

4. EXPERIMENTAL RESULTS

The CLSQ allows for a reduction in the data cache energy consumption by avoiding cache accesses. The number of cache accesses that can be eliminated is directly dependent on the number of entries that are available to be put in *"cached"* state. The CLSQ can have entries in the *"cached"* state only when it is not full. Figure 3 presents the percentage of time when the LSQ was full for a system using the *base LSQ*. In general, for the integer benchmarks the LSQ is less full than for the floating point benchmarks. It can be observed that even a modest 8–entry LSQ is full on average only 46% of the time for the integer benchmarks, and 81% of the time for the floating point benchmarks. Increasing the LSQ size dramatically decreases the amount of time the LSQ is full, a 32–entry LSQ is only full 6% and 25% of the time for the integer and floating point benchmarks, respectively. A 64–entry LSQ is almost never full because the configuration used has a 64–entry RUU.

The above results show the availability of a significant number of unused entries in the *base LSQ*. Those unused entries can be put in the *"cached"* state, allowing for potential hits in the CLSQ, thus avoiding data cache accesses that are more expensive in terms of both time and energy consumption.

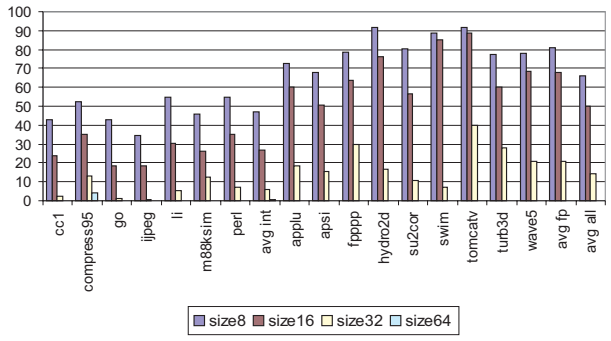

Figure 3: Percentage of time when the baseline LSQ was full

Figure 4 shows the percentage of the total executed load instructions that hit only on *"cached"* entries in the CLSQ. It does not include LSQ hits due to store–to–load forwarding. The percentage for integer benchmarks, ranging on average from 21.5% for an 8–entry LSQ to 45.5% for a 64–entry LSQ, is higher than the percentage for the floating point benchmarks that ranges from 4.38% to 27%. The significant number of hits in the CLSQ shows that the proposed design is an effective mean of avoiding cache accesses and reducing the data cache energy consumption.

It is interesting to compare the number of hits in the CLSQ with the hits in the *base LSQ* due to store–to–load forwarding. Figure 5 shows the percentage of load instructions that hit in the *base LSQ* due to store–to–load forwarding. It can be observed that the percentage of load instructions that hit in CLSQ is significantly higher than the percentage of load instructions that get store–to–load forwarding. This can be explained by the increased number

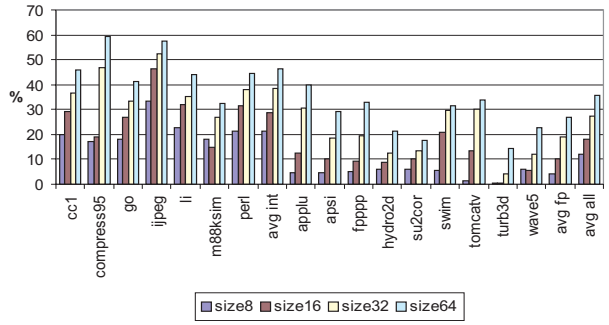

Figure 4: Percentage load instructions that hit in the CLSQ

of entries that can be hit on in the CLSQ, and by a longer lifetime in the LSQ of the entries in *"cached"* state. An example of a case where store–to–load forwarding does not work as well as it could is passing procedure parameters on the stack. Writing parameters to the stack is likely to be a cache hit, so the write operation can complete quickly. If enough instructions are executed between writing the procedure parameters in the caller and reading them in the callee, the write instruction has time to complete and be retired, so store–to–load forwarding cannot take place. For space reasons the LSQ hits due to store–to–load forwarding for the CLSQ are not shown here, but the results are within 1% of the ones for the *base LSQ*.

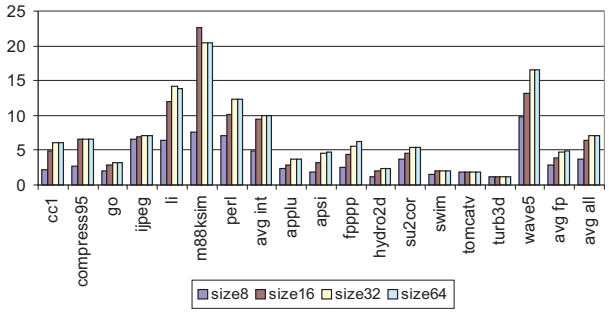

Figure 5: Percentage load instructions that hit in the *base LSQ* (i.e. store–to–load forwarding)

One of the main ideas implemented by the CLSQ is allowing hits on the data read by load instructions. Figure 6 can be used to evaluate the effect of reusing data from retired load instructions. The figure shows the percentage of load instructions that hit in the CLSQ on *"cached"* entries that have been generated by retired load instructions, the rest of the hits are generated by retired store instructions. The percentage is over 50% in most cases. This result justifies adding the capability to hit on data from retired load instructions to the CLSQ. All the CLSQ hits, on data from either retired load or store instructions are new, they are not the same as the LSQ hits due to store–to–load forwarding.

An important effect of load instructions hitting in the CLSQ is that the number of L1 cache accesses is reduced, as shown in Figure 7. For the integer benchmarks the reduc-

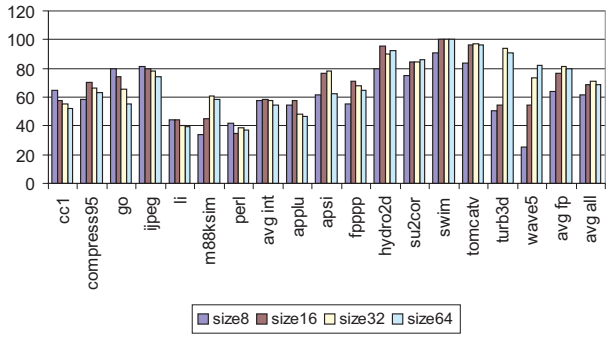

Figure 6: Percentage load instructions that hit in the CLSQ on data from previous load instructions

tion varies on average from 17.25% to 44.30%, and for the floating point benchmarks from 3.26% to 20.53%.

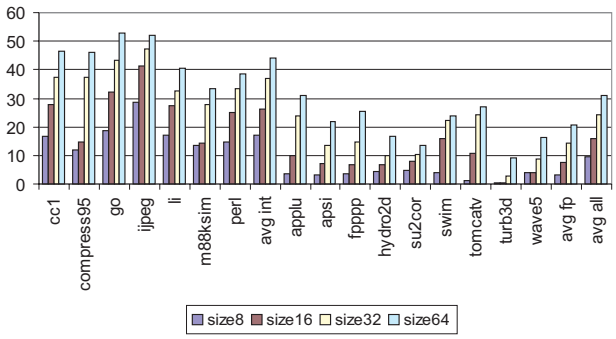

Figure 7: Percentage reduction in the number of data L1 cache accesses when using a CLSQ

Another cause for the reduction in the number of loads that access the data cache is the reduction in the number of miss–speculated loads. Load instructions complete faster when hitting in the CLSQ. Branches that depend on those loads are therefore resolved faster, allowing a decrease in the number of speculative load instructions that are issued, hence reducing the number of load instructions that are miss–speculated. Figure 8 shows the reduction in the total number of instructions that are issued in a system using CLSQ as compared to a system using *base LSQ*. This reduction in the number of miss-speculated load instructions is another contributing factor to the reduction of the data cache energy consumption. For the floating point benchmarks the reduction is under 0.6%, and it is not shown in the graph.

The CLSQ allows for avoiding a significant number L1 cache accesses both directly, by hitting in the CLSQ entries, and indirectly by allowing for a decrease in the number of miss-speculated load instructions. The direct consequence of avoiding a large number of L1 cache accesses is the reduction in energy spent on cache access during the execution of a program. The energy savings for the L1 data cache are shown in Figure 9. The energy savings vary between 5% and 45%. The integer benchmarks show higher energy savings than the floating point ones. This directly correlates with the fact that the integer benchmarks have a higher hit rate

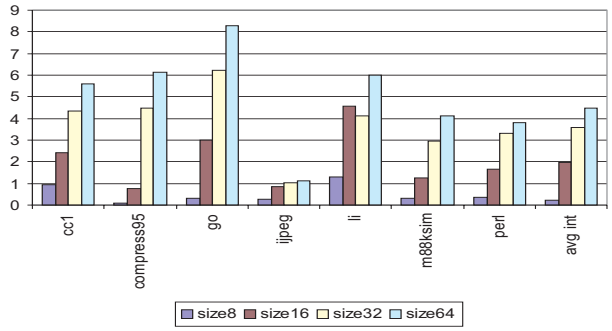

Figure 8: Percentage reduction in the number of load instructions issued with a CLSQ

in the CLSQ.

The above results show that the CLSQ, a design that is centered on the idea of better using existing CPU resources has a significant impact on the data cache energy consumption. The influence of the CLSQ on the total processor en-

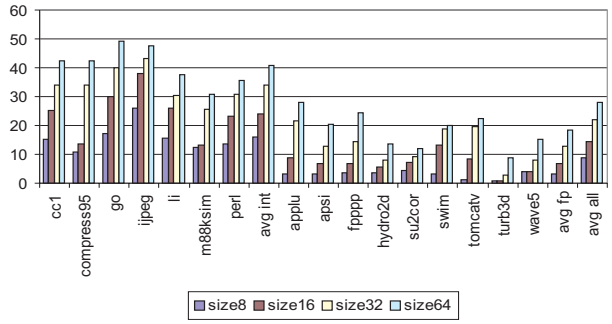

Figure 9: Percentage dcache energy consumption reduction with a CLSQ

ergy consumption is shown in Figure 10. The total processor energy consumption savings vary between 1% and 8%. The main factor contributing to the decrease of the energy consumption for the whole processor is the reduction of energy consumption by the data cache.

The load instructions that hit in the LSQ complete in 1 cycle, faster than instructions that have to access the cache. The CLSQ thus allows for an increase in execution speed. Hence the CLSQ has a positive impact on both factors of the energy–delay product: it reduces the energy consumption, and it increases the execution speed. Figure 11 shows the improvement of the energy–delay product when using a CLSQ. For the integer benchmarks the energy–delay product improvement ranges on average from 5.79% to 15.75% for the integer benchmarks and from 0.83% to 3.44% for the floating point ones.

5. CONCLUSION

This paper presents a novel technique for reducing the data cache energy consumption and the energy–delay product. The technique is applied in the design of *CLSQ*, a LSQ implementation that allows data from Load/Store instructions to be retained in the deallocated LSQ entries after the

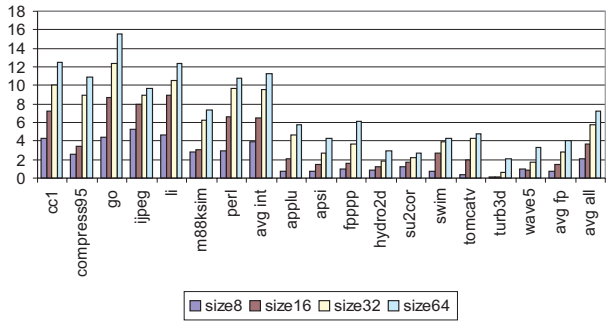

Figure 10: Percentage processor energy consumption reduction with a CLSQ

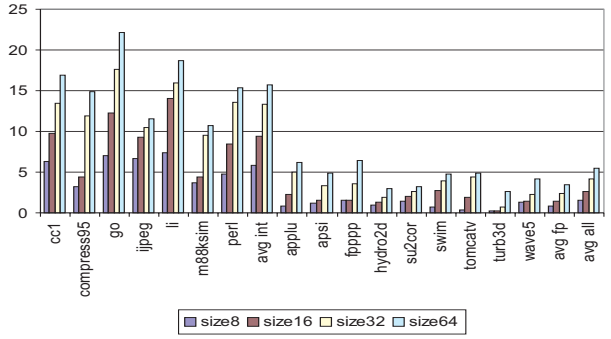

Figure 11: Percentage energy–delay product improvement when using a CLSQ

corresponding instructions have been retired. The paper shows that the number of load instructions that hit in the CLSQ is significant. Hitting in the CLSQ avoids a cache access, is faster and consumes less energy than a cache access, hence it reduces the total energy spent for a load instruction and it improves the energy–delay product.

The proposed modifications to the LSQ are small and do not have a negative impact on performance. The design makes better use of resources already existent in a modern CPU. The expected increase in cache latency for future processor will increase the speed–up obtained by using the CLSQ. The ideas proposed in this paper can be applied to other types of Load/Store Unit organization: units that have separate queues for loads and stores, and units that store full cache lines in each entry.

Future extensions of this work will include studying the impact on data cache energy consumption of allowing hits in the LSQ on pending load instructions and the effect of squashing silent stores that can be detected at the LSQ level.

6. REFERENCES

[1] T. M. Austin and G. S. Sohi. Zero-cycle loads: Microarchitecture support for reducing load late ncy. pages 82–92.

[2] 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.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, pages 83–94, 2000.

[4] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, 1997.

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

[6] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, (Q1):13, Feb. 2001.

[7] K. Inoue, T. Ishihara, and K. Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 273–275, 1999.

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

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

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

[11] A. Moshovos and G. S. Sohi. Read-after-read memory dependence prediction. 1999.

[12] D. Nicolaescu, A. Veidenbaum, and A. Nicolau. Reducing power consumption for high-associativity data caches in embedded processors. In *DATE2003 Proceedings*, 2003.

[13] W. Tang, A. Veidenbaum, A. Nicolau, and R. Gupta. Simultaneous way-footprint prediction and branch prediction for energy savings in set-associative instruction caches. In *IEEE Workshop on Power Management for Real-Time and Embedded Systems*, 2001.

[14] J. Yang and R. Gupta. Energy-efficient load and store reuse. In *ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 72–75, 2001.