Energy-Efficient Load and Store Reuse[*]

Jun Yang Rajiv Gupta
Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

ABSTRACT

A load and store reuse mechanism can be used for filtering memory references to reduce memory activity including on-chip cache activity. The challenging aspect of this task is to ensure that energy savings achieved in memory are not offset by energy used by the reuse hardware. In this paper we present the design of a reuse mechanism which has been carefully tuned to achieve net energy savings. In contrast to traditional filter cache designs which trade-off energy reductions with higher execution times, our approach reduces both energy and execution time.

1. INTRODUCTION

One source of energy consumption is the data cache which continues to grow in size and area. One approach that has been suggested is to introduce a filter cache between the CPU and the L1 cache [2]. A filter cache is small in size and therefore energy efficient. It leads to reduction in energy consumed by the larger L1 cache at the cost of increased execution times. In this paper we propose the use of a load and store reuse unit as a filtering mechanism. This approach generally leads to reductions in both the energy consumed and the execution time.

Extensive research has been carried out on load and store reuse that are speculative in nature which are typically not energy efficient. In this paper we use non-speculative reuse techniques. The reuse opportunities can arise in multiple ways (see Figure 1). A load can be avoided using reuse if value that it loads from an address was loaded or stored from the same address by a prior memory instruction and the contents of that location have not changed since. An instance of a load instruction can be reused because of a prior instance of the *same* or *different* load instruction, or a prior instance of a *store* instruction. A store is avoidable if the value it writes to a location is already present in that location. The store reuse can be classified as that arising at previous instance of the *same* or *different* store, and previous instance of a *load*. The existing non-speculative techniques for reuse only target same load reuse opportunities [3].

[*]Supported by DARPA PAC/C Award F29601-00-1-0183 and NSF grants CCR-0105355, CCR-0096122, EIA-9806525, and EIA-0080123 to the University of Arizona.

PC_1: Load R1, addr
...
PC_1: Load R1, addr
(a) *Same* Load.

PC_1: Store R1, addr
...
PC_1: Store R1, addr
(d) *Same* Store.

PC_1: Load R1, addr
...
PC_2: Load R2, addr
(b) *Different* Load.

PC_1: Store R1, addr
...
PC_2: Store R2, addr
(e) *Different* Store.

PC_1: Store R1, addr
...
PC_2: Load R2, addr
(c) *Store* Load.

PC_1: Load R1, addr
...
PC_2: Store R2, addr
(f) *Load* Store.

Figure 1: Opportunities for Load and Store Reuse.

Even though a non-speculative approach will reduce cache activity, achieving an overall reduction in energy consumed using reuse is still a challenging task. This is because a load-store reuse filter will itself consume energy. To get an overall reduction in energy consumed we must save more energy in the cache than is consumed by the reuse mechanism. In fact we implemented an aggressive reuse algorithm which tries to capture maximum possible reuse. We used separate table structures to save the histories of past loads and stores. We used associative searches on the address fields of these tables to determine if a load or store was avoidable. This approach allowed us to capture all forms of reuse described above. The sizes of the history tables used were 256 entries each. The outcome of this experiment is shown in Figure 2. Instead of saving energy we ended up using more energy. This is because the reuse mechanism used much more energy than what is consumed due to the cache activity.

In this paper we present the design of a reuse mechanism which has been carefully tuned to achieve two objectives. Our first objective is to develop a reuse mechanism that is able to capture and filter out a large fraction of reusable loads and stores. This objective is achieved by designing reuse hardware which is able to capture reuse arising from multiple sources. The second objective of this work is to design a reuse mechanism which performs the filtering task in an energy-efficient manner. In programs with significant levels of reuse, we would like to achieve net energy savings while for programs with little reuse we would like to minimize the net increase in energy consumed. This objective is achieved in two ways. First we design reuse hardware where the history of previously executed load and store operations

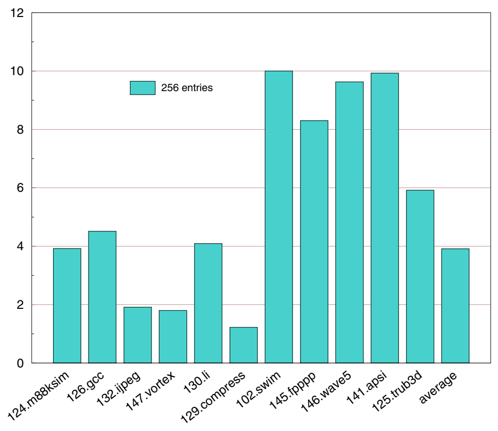


Figure 2: An Aggressive Reuse Scheme.

is accessed through indexed tables - associative lookups are kept to a minimum. Second our algorithm is not too aggressive, and in fact if it fails to detect reuse, it backs off from certain types of reuse detection so that the energy consumed by reuse detection hardware can be kept to a minimum.

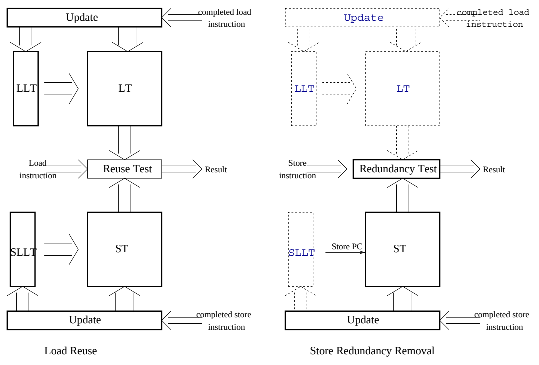
The remainder of the paper is organized as follows. In section 2 we present the hardware design of our reuse mechanism and develop an energy-efficient algorithm for using this mechanism. In section 3 we present the results of the performance evaluation.

2. REUSE HARDWARE DESIGN

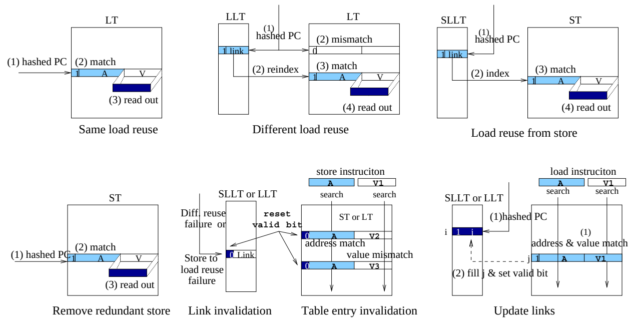
Reuse studies. To develop an energy-efficient design we carried out studies which guided the design of such a mechanism. First study measured the amounts of different types of reuse opportunities. We concluded that while all types of load reuse should be exploited, it is useful to prioritize the reuse detection process: we first looked for same load reuse, then different load reuse, and finally store to load reuse. Each load is put only in the first category that it is found to satisfy. Most of the reuse for stores is due to same or different stores. Therefore we only considered these two types of reuse for avoiding stores.

The second study guided the selection of table sizes and the decisions relating to the complexity of algorithms for capturing *different* load/store reuse. We considered different tables sizes to save load and store history since the energy consumed by these tables increases with their size. We found that separate load and store history tables both of sizes of 32 or 64 entries are quite sufficient. If we increase the table sizes further the energy consumed grows much more rapidly that the increase in the amount of reuse. We also intended to design reuse hardware which minimizes high energy consuming associative searches and mainly uses indexed accesses whenever possible. For this purpose, other than the same load/store reuse cases, we explicitly link the two instructions involved in the reuse. An instruction may be able to reuse results from many different instructions. Thus it is unclear how many links need to be allowed. Again creating more links will cause increased energy consumption both in their creation and use. We found through our experiments that linking a load with one other load is sufficient - linking it with more loads finds only small amounts of additional reuse. We also evaluated the benefit of linking a store to different prior stores and found that providing such links is not very useful.

Reuse unit. We now present the design of the *Reuse Unit* (RU), the hardware structure for detecting and filtering load and store instructions. The various pieces of the reuse unit are not always used by every memory instruction. We selectively prevent their use to reduce activity and save energy if we feel that their use will not result in additional load and store filtering.



(a) Block Diagram.



(b) RU Operations; A stands for address, V stands for value

Figure 3: The Reuse Unit (RU).

In Figure 3 we first show the high level block diagram of the RU(a). Next we describe the various structures in the RU and then later we describe how it is maintained through different operations(b).

LOAD TABLE (LT): LT stores histories of loads that have already been executed. It is designed as a small direct map array so that we can achieve low access times and energy. Each entry contains the effective address, data value at that address, and a mem-valid bit. The mem-valid flags the validity of current contents of the entry. The entry is invalid if nothing has been stored in it or the value stored in the entry is stale.

LOAD LINK TABLE (LLT): This table is needed to capture *different* load reuse opportunities. It is also a direct mapped structure which provides a link between a load and another load whose results can be reused. The link is in form of LT index corresponding to the load at which reuse originates. The LLT contains the same number of entries as the LT and there is one-to-one correspondence between the LT and LLT entries. An additional bit in each LLT entry

is used to indicate the *usability* of the link stored in that entry. A link marked unusable is not used to index the LT. This bit helps in reducing energy consumed by eliminating unnecessary look ups of the LT.
STORE TABLE (ST): ST is the counterpart of LT - it stores the history of past stores that have been executed. It serves two purposes. First, it provides a means for detecting store to load reuse. And second, it helps in the detection of same store reuse. It is also a small direct mapped array. The ST contains fields for memory address, its value, and the mem-valid bit.
STORE to LOAD LINK TABLE (SLLT): This table aids in detecting store to load value reuse. It basically links each load in LT with an entry in ST. Like the LLT, there is also a single bit corresponding to every link which indicates whether the link is usable or not.

A load's PC is hashed into a LT entry in order to attempt same load reuse. If this entry is valid, address stored in the table and the load's effective address are compared for reusability. The load can use the value stored in the entry if the address comparison succeeds. The load's PC is hashed into a LLT entry for different load reuse. If the link in LLT is marked as usable, the corresponding entry pointed to by the link will be tested in the same manner as the same load reuse test. If the load passes this check we have succeeded in different load reuse. Otherwise, the load's PC is hashed into SLLT to attempt store to load reuse.

Filtering of stores is carried by simply using the ST history. Filtering of a store means it is not sent to or deleted from the store queue and therefore it never reaches the cache.

RU updates. On completion of each load/store instruction, the LT/ST table is updated by setting the address and value fields and also setting the mem-valid bit. LLT links are set up only if they are needed, that is, only if same load reuse has failed we create these links to attempt to find different load reuse. Similarly the SLLT link for a load is set up only if we fail to find different load reuse. The LLT and SLLT links are set up following an associative search of the address fields in LT and ST. A link is set to be usable when it is first created. It remains set if it is used successfully, that is, its use results in filtering. A link is marked unusable when the reuse test using that link fails.

The above strategies of creating a LLT link only if same load reuse fails, creating a SLLT link only if different reuse fails, and marking links as unusable if their use fails to detect reuse all greatly reduce RU activity and were therefore extremely important in obtaining an energy efficient design.

Integration into a superscalar. The reuse mechanism is incorporated into a superscalar pipeline as shown in Figure 4. An extra stage, the load store reuse (LSR) stage, is introduced immediately preceding the data cache access stage. This stage uses the RU to determine if the load or store instruction is reusable and therefore need not be sent to the cache. If the instruction is not found to be reusable because all reuse tests fail, we send it to the cache. Therefore the load/store instructions that are not reusable pay an extra penalty of one cycle in this implementation. The ones that are found to be reusable benefit because they are completed in one cycle.

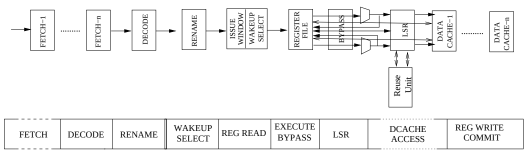


Figure 4: Superscalar Processor with Reuse.

3. PERFORMANCE EVALUATION

We have implemented the techniques described in this paper and now we describe the experimental setup used in this work. The experiments are based upon an 8-issue superscalar processor with the pipeline structure shown in Figure 4. The baseline processor has one less stage since it does not carry out load and store filtering. Therefore if load/store reuse always fails, our processor will take one more cycle than the baseline processor for each data cache access. The data cache is a 32 Kb direct-mapped cache with 32 byte line size and 6 cycle miss penalty. There are 2 read and 2 write ports to the data cache.

The table sizes used in these studies are 32 and 64 for both load and store history tables. The cache energy models were obtained from the SimplePower [4] simulator and the energy models for the reuse hardware were implemented using the models for array and CAM structures used in Wattch [1], both using 0.8 micron technology.

We ran experiments where the number of cycles for data cache access was varied – 2 and 6 cycle accesses were considered. This was motivated by the observation that future generation processors are devoting increasing number of stages to data cache access due to reduced cycle times which are too small to perform cache accesses in a single cycle. All data except the IPC improvements are presented assuming that data cache access takes 2 stages. This is because all other data shows very small changes when the number of stages is changed.

Energy savings. We measured the energy consumed by the cache with and without reuse hardware as well as the energy consumed by the reuse hardware. In Figure 5, the first graph gives us the reduction in energy consumed by the cache. As we see the reductions are substantial and on the average 15% and 18% reductions were observed for table sizes of 32 and 64 respectively. The second graph gives the energy consumed by the reuse hardware as a percentage of energy used by the cache. As we can see, our reuse hardware design is quite energy efficient as the 32 (64) entry table consumes energy that is only 4% (7%) of the energy consumed by the cache.

We define the net savings in the cache energy as the difference between the cache energy savings and the energy consumed by reuse hardware. The net energy savings are plotted in the third graph of Figure 5. As we can see the savings are substantial ranging from a few percent to as much as 47%. For benchmarks with low levels of reuse with table size of 64 there is a net loss in energy. However, the loss is less than 3% in these cases. Thus we have achieved our objective of designing reuse hardware which provides substantial energy savings when high levels of reuse is captured and very little increase in energy is observed when little reuse is found.

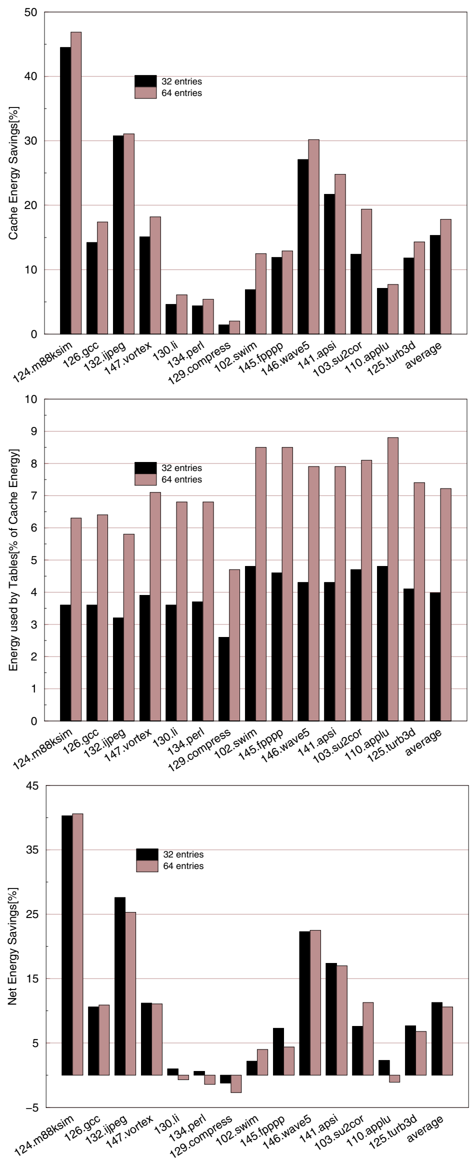


Figure 5: Energy Figures: (a) Cache Savings; (b) Reuse Hardware Consumption; and (c) Net Savings.

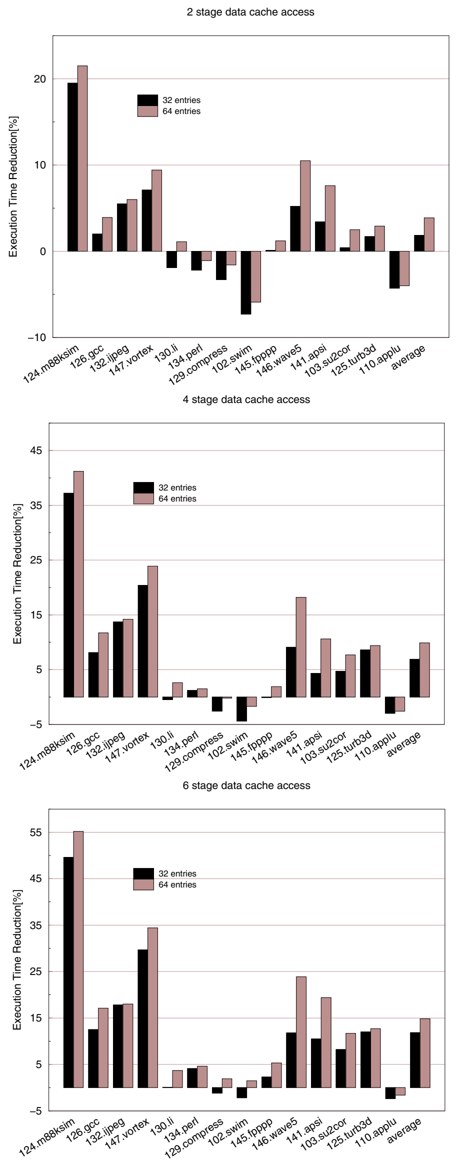


Figure 6: IPC Improvements for Varying Cache Latencies.

IPC improvements. Next we present the IPC improvements that are observed due to load and store filtering. In Figure 6 we present these improvements assuming that data cache access takes 2 and 6 cycles respectively. As we can see, the longer the cache latency, the greater are the savings due to load and store filtering. The average IPC improvements range between 3% and 15% for different values of data cache access cycles. For programs with high levels of reuse the IPC improvements are substantial - even as high as 55%. On the other hand for programs with very low levels of reuse, reductions in IPC were observed because the baseline processor has one less stage and therefore completes cache accesses one cycle earlier. The reductions in IPC are mostly less than 5%.

In conclusion we have demonstrated that programs contain significant levels of load and store reuse opportunities. By filtering loads and stores we can achieve significant energy savings and at the same time generally achieve modest speedups. Therefore our reuse unit is a superior load and store filtering mechanism that traditional filter caches.

4. REFERENCES

[1] D. Brooks, V. Tiwari, and M. Martonosi, “Wattch: A Framework for Architectural-Level Power Analysis and Optimizations,” in *ISCA-27*, pages 83–94, May 2000.

[2] J. Kin, M. Gupta, and W.H. Mangione-Smith. “Filter Cache: An Energy Efficient Memory Structure,” In *MICRO-30*, pages 184–193, December 1997.

[3] A. Sodani and G. S. Sohi, “Dynamic Instruction Eeuse,” in *ISCA-24*, 1997.

[4] N. Vijaykrishnan, M. Kandemir, M.J. Irwin, H.S. Kim, and W. Ye, “Energy-Driven Integrated Hardware-Software Optimizations Using SimplePower,” in *ISCA-27*, pages 95–106, May 2000.