Energy-Efficient Load and Store Reuse *

Jun Yang Rajiv Gupta

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

Abstract

Our experience has shown that even programs compiled using optimizing compilers contain very high levels of load and store redundancy. With the increasing importance of energy-efficient processor design, this load-store redundancy can be viewed as an opportunity for reducing on-chip cache activity and hence the energy consumed by the cache. However, taking advantage of this opportunity is a non-trivial task because a load-store reuse mechanism for removing redundancy 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 resuse mechanism. Our experiments with an aggressive redundancy removal mechanism resulted in a net increase in energy used.

In this paper we present the design of a reuse mechanism which has been carefully tuned to achieve two objectives: (i) capture and eliminate a large fraction of redundant load and store instructions; and (ii) eliminate the redundancy using less energy than what is saved in the cache. A number of strategies are used to minimize the activity in the *reuse unit*. For programs with high levels of redundancy we obtain IPC improvements of up to 55% and net cache energy savings of up to 47%. Even more importantly, in programs where little redundancy is found, the net increase in energy comsumed is less than 3%.

Keywords - redundant loads, silent (redundant) stores, load linking, bus switching, cache activity, energy savings.

1 Introduction

Importance of energy-efficient processing is growing due to an increase in the use of mobile and embedded computing systems. One source of energy consumption is the data cache which continues to grow in size and area with each new generation of high performance processors. We have identified an opportunity for reducing cache activity which can lead to reductions in energy consumed by the cache as well as improvements in executed instructions per cycle (IPCs) for a class of programs with significant levels of load/store redundancy. In particular, we have observed that even programs compiled using optimizing compilers contain very high levels of load and store redundancy - for example, in `SPEC95` benchmarks compiled using `gcc` with `-O3` option, we observed that on an average 95.6% of all loads and 42% of all stores are redundant. The high levels of redundancy are a direct consequence of limitations of compile-time analysis techniques used by compilers - specially highly conservative memory disambiguation techniques used by compilers make it necessary to introduce load and store instructions to guarantee program correctness. If this redundancy could be eliminated, cache activity as well as address and data bus switching will is reduced leading to lower energy consumption.

Extensive research has been carried out on load and store redundancy removal techniques that are speculative in nature [3, 4]. However, speculative techniques do not reduce memory activity. This is because such techniques must reference the memory to detect misspeculation. Non-speculative load and store reuse techniques are more appropriate for energy efficient design. For example instruction reuse technique developed by Sodani and Sohi [5] when considered in context of load and store instructions can simultaneously improve IPCs and reducing cache activity. We take this approach in this paper. Even though this 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 mechanism for removing redundancy 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 resuse mechanism. Our experiments with an aggressive redundancy removal mechanism resulted in a net increase in energy used because energy consumed by the reuse mechanism greatly exceeded the savings in cache energy. Also if the program contains little redundancy, while the reuse mechanism will continue to consume energy, little savings in cache energy will result. Therefore we would like to minimize the increase in net energy consumed in such situations.

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 eliminate a large fraction of redundant load and store instructions. The second objective of this work is to design a reuse mechanism which performs redundancy elimination in an energy-efficient manner. In programs with significant levels of redundancy, we would like to achieve net energy savings while for programs with little redundancy we would like to minimize the net increase in energy consumed.

The first objective is achieved by designing reuse hardware which is able to capture redundancy arising from multiple sources. For example, a load can be made redundant by three distinct categories of memory instructions executed earlier. The second objective is achieved in two ways. First we design reuse hardware where the history of previously executed load and store operations 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 redundancy, it backs off from certain types of redundancy detection so that the energy consumed by reuse detection hardware can be kept to a minimum. Our experiments demonstrate that for programs with high levels of redundancy we obtain IPC improvements of up to 55% and net cache energy savings of up to 47%. Even more im-

*Supported by DARPA award no. F29601-00-1-0183 and National Science Foundation grants CCR-0096122, EIA-9806525, and CCR-9996362 to the University of Arizona.

portantly, in programs where little redundancy is found, the net energy loss is less than 3%.

A small subset of the redundant loads and stores is also captured by the store buffer. For example, a load may not need to go to memory because it may find the value it requires in the store buffer. Similarly if multiple pending stores to the same address location are pending in the store buffer, all but the last of these writes need not be sent to memory. However, we must remember that the removal of redundancy in these situations will involve associative searches of the store buffer. These operations are clearly quite energy consuming. In contrast the reuse hardware that we have designed mainly involves indexed operations because we use direct mapped structures instead of associative structures. Therefore our approach will not only capture far greater amounts of redundancy than the store buffer, but it is also energy efficient.

The remainder of the paper is organized as follows. In section 2 we describe the different forms of load and store redundancy and present the result of a study that measured the amounts of redundancy in `Spec95` benchmarks. In section 3 we present the hardware design of our reuse mechanism and develop an energy-efficient algorithm for using this mechanism. In section 4 we present the results of the performance evaluation.

2 Load and Store Redundancy Study

In this section we present results of experimental studies designed to measure the degree of load and store redundancy present in programs. Further we have identified different categories of load and store redundancies and measured the contributions of each of these categories. This study is important not only because it tells us how much redundancy is present, but it also guides in designing hardware which focuses on capturing those forms of redundancies that are present in substantial amounts. Therefore the resulting design we obtain is *energy-efficient*.

Types of load and store redundancies. The classification of redundancy is shown in Figure 1. A load is redundant 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 redundant because of a prior instance of the *same* load instruction, a prior instance of a *different* load instruction, or a prior instance of a *store* instruction. In the examples of these situations in Figures 1(a-c), if the addresses, and values, involved in these pairs of instructions are the same, the second instruction which is a load is redundant.

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: Classification of Load and Store Redundancy.

A store is redundant if the value it writes to a location is already present in that location. The store redundancy can be classified as that arising at previous instance of the *same* store, previous instance of a *different* store, and previous instance of a *load*. Examples of these situations are shown in Figures 1(d-f). The second instruction, which is a store, need not be executed if the history of the prior load/store instruction is available and therefore it can be determined that the store is writing into the address the same value that the address already contains.

An aggressive redundancy removal mechanism. We first implemented an aggressive redundancy elimination algorithm which tries to capture maximum possible redundancy. 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 redundant. This approach allowed us to capture all forms of redundancies described above. The sizes of the history tables used were 256 entries each. The outcome of this experiment is shown in Figure 2. While significant IPC improvements were observed, 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.

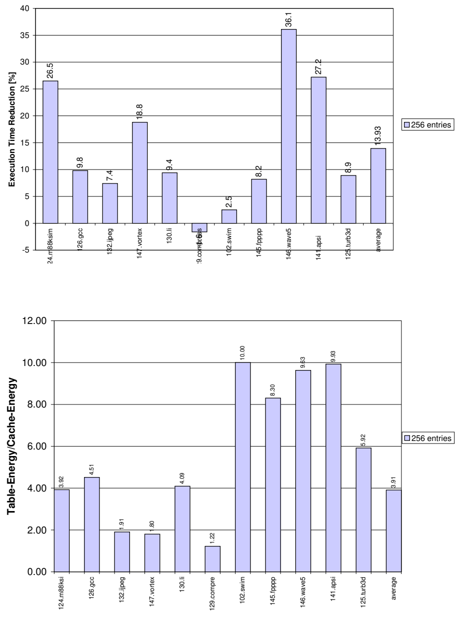


Figure 2: An Aggressive Reuse Scheme: IPC Improvement and Increased Energy Consumption.

In order to develop an energy-efficient design we carried out studies which helped us in the design of such a mechanism. First study measured the amounts of different types of redundancies so that we could focus on those types which will give most benefit. The next study further guides the selection of table sizes and the decisions relating to the complexity of algorithms for capturing *different* load/store redundancy.

Amounts of load and store redundancy of different types. The results of our study of redundancy in `Spec95` benchmarks is shown in Figures 3 and 4. First let us consider the characteristics of load redundancy. A load may be redundant due to all three conditions of Figure 1 being true. Therefore we prioratized our redundancy detection process: we first looked for same load redundancy, then different load redundancy, and finally store to load redundancy. Each load is put only in the first category that it is found to satisfy. As we can see from Figure 3, on an average, nearly 95.6% of total executed loads were detected as being redundant. Moreover the redundant loads in each of the three categories are substantial. Under our definition nearly all loads are redundant. This is because before a value is loaded it is almost certainly computed and stored at some earlier point in the program. Therefore if a load is not same or different load redundant, it most certainly is store to load redundant. Traditionally reuse techniques have focussed on exploiting *same* load redundancy. However, as we can see the other two categories of load redundancy are also very importnat. Therefore we design an implementation that exploits all three types of load redundancy.

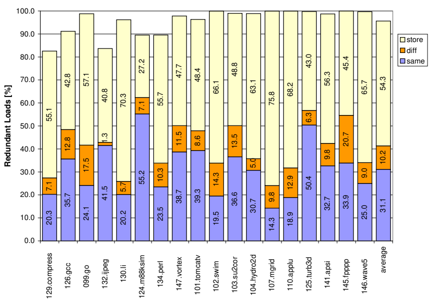


Figure 3: Load Redundancy Characteristics of Spec95 Benchmarks.

Now lets look at the results of store redundancy study in Figure 4. On an average, in these benchmarks, 42% of all executed stores are redundant. As we can see, most of the redundancy is same store redundancy. There is also some redundancy due to different stores. However, the store redundancy due to loads is almost nonexistent. Therefore in our further studies in this section, that will guide our design of reuse hardware, we will only consider two types of redundancy.

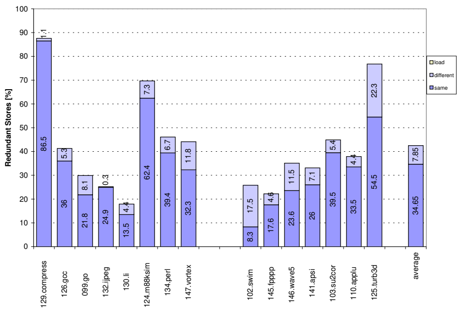


Figure 4: Store Redundancy Characteristics of Spec95 Benchmarks.

History table sizes and linking parameters. The above experiments have already helped in restricting our attention to those categories of redundancies that will provide the most benefit. Next we conducted experiments which aimed at helping us in arriving at an energy efficient design in two ways. First we considered different tables sizes to save load and store history. The energy consumed by these tables increases linearly with their size. Therefore we should use larger tables only if it results in substantially higher levels of redundancy to be removed. Second we intend 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 redundancy 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 redundancy. Also we found that small history table sizes of 32 and 64 entries are quite sufficient. If we increase the table sizes further we capture more redundancy; however, the increase in energy consumed grows much more rapildly that the increase in the amount of redundancy detected. We repeated a similar experiment for store redundancy. We evaluated the benefit of linking a store to different prior stores. Provide links is not very useful here. Only a very small amount of additional redundancy is captured by linking stores with prior stores. Therefore we decided to consider only *same* store redundancy. Also again we found that small history table sizes of 32 and 64 entries for store history are quite sufficient. If we increase the store table sizes further we capture more redundancy; however, the increase in energy consumed grows much more rapildly that the increase in the amount of redundancy detected.

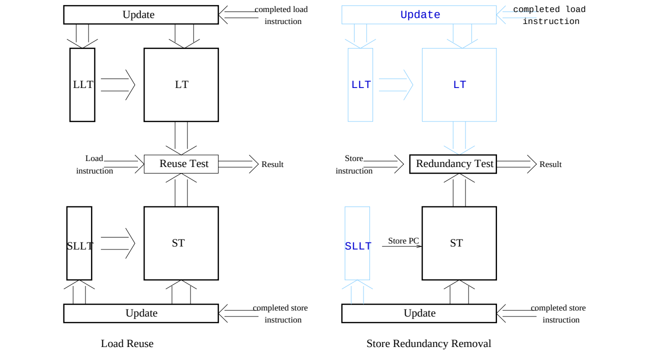
3 Reuse Hardware and Update Algorithms

In this section we present the design of the *Reuse Unit* (RU), the hardware structure for detecting and removing redundant 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 redundancy removal.

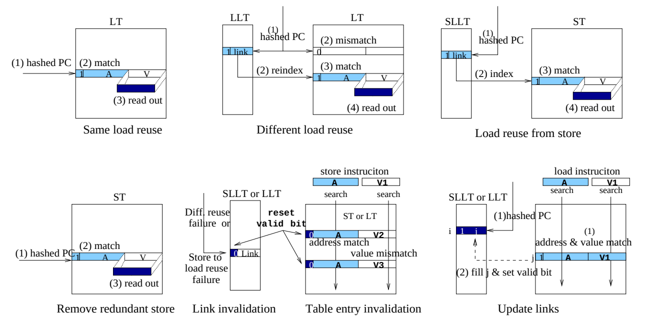
RU Structures. In Figure 5 we first show the high level block diagram of the RU(a). Next we describe the various structures in the RU and then later we describe how it is maintained through different operations(b).

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



(a) Block Diagram.



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

Figure 5: The Reuse Unit (RU).

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 redundancy. 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.

Redundancy Removal. 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.

Removal of redundant stores is carried by simply using the ST history. Removal of a redundant 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 redundancy removal. 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 redundancy all greatly reduce RU activity and were therefore extremely important in obtaining an energy efficient design.

Integration into Superscalar Pipeline. The reuse mechanism is incorporated into a superscalar pipeline as shown in Figure 6. An extra stage, the load store redundancy removal (LSRR) stage, is introduced immediately preceding the data cache access stage. This stage uses the RU to determine if the load or store instruction is redundant and therefore need not be sent to the cache. If the instruction is not found to be redundant because all reuse tests fail, we send it to the cache. Therefore the load/store instructions that are not redundant pay an extra penalty of one cycle in this implementation. The ones that are found to be redundant benefit because they are completed in one cycle.

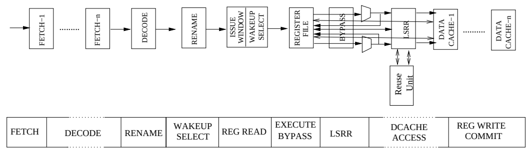


Figure 6: Superscalar Processor with Reuse Hardware.

4 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 6. The baseline processor has one less stage since it does not carry out redundancy detection and removal. Therefore if load/store redundancy elimination 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 [6] 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.

Now let us examine the results obtained. We present the degree of redundancy captured. The reduction cache activity in terms of references that go to the data cache and the amount of address and data bus switching is presented. The energy savings in the cache and the energy used by the reuse hardware is presented and based upon this the net cache energy savings are computed. Finally the IPC improvements are presented.

Redundancy Captured. On an average over all the benchmarks nearly 25% and 31% of all loads from table of 32 and 64 entries were eliminated by our mechanism (see Figure 7). This number is quite substantial and most importantly we can see that all three types of redundancy play an important role in the overall performance. Thus our decision to capture the three types of redundancy was a correct one. On an average nearly 12% of stores were also eliminated (see Figure 7).

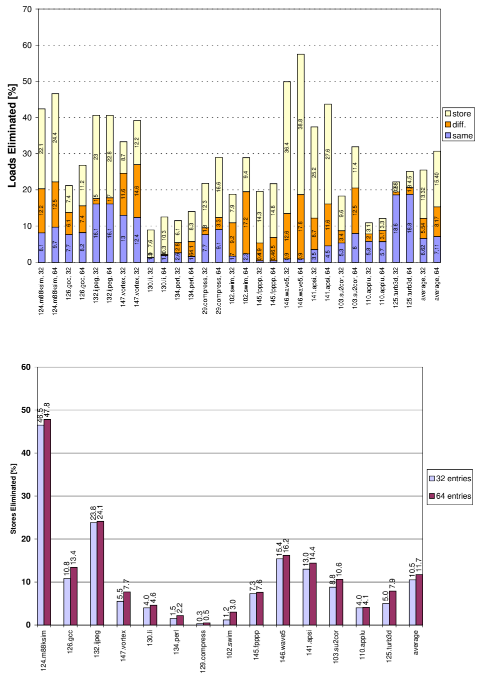


Figure 7: Redundancy Captured.

As expected, the redundancy captured is lower than the redundancy observed in the ideal study that we conducted. This is because the ideal study did not take into account the timing issues. It was assumed that an instruction could always benefit from a prior instruction. However, in a superscalar processor the prior instructions may not have completed execution and thus redundancy may not be captured. In addition, limited table sizes, the delays in establishing links, the limitation on the number of links, etc. all contribute to the loss.

Cache Activity. The successful removal of redundancy reduces cache related activity in two ways: first the read and write accesses to the cache are reduced and second the bus activity of the data and address busses is reduced. We measured the changes in these activities. From the results in Figure 8 we can see that the number of cache accesses are reduced by nearly 15% and 18% when history tables of sizes 32 and 64 entries are used. The address bus switching is reduced by nearly 15% and 19% while the data bus switching activity is reduced by 22% and 28% for the 32 and 64 entry tables respectively. Therefore there is a significant reduction in cache related activity.

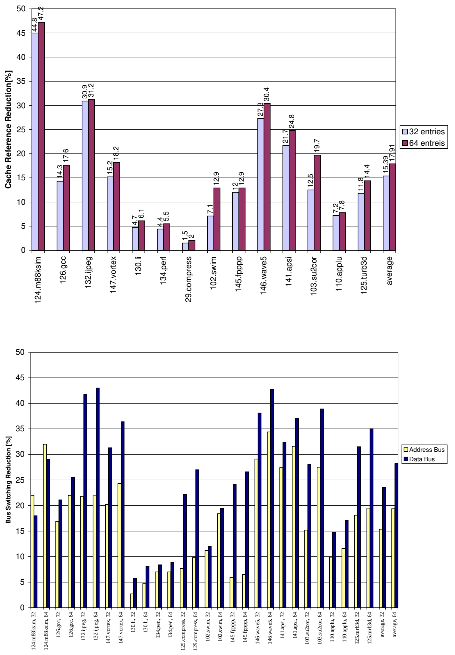


Figure 8: Reduction in Cache Activity.

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 9, 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 9. As we can see the savings are substantial ranging from a few percent to as much as 47%. For benchmarks with low levels of redundancy 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 redundancy is captured and very little increase in energy is observed when little redundancy is found.

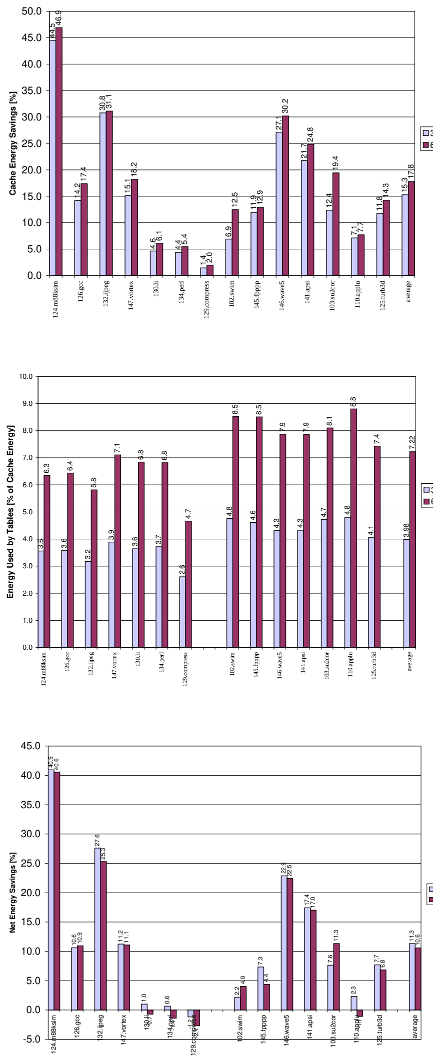


Figure 9: Energy Figures: (a) Cache Savings; (b) Reuse Hardware Consumption; and (c) Net Savings.

IPC Improvements. Next we present the IPC improvements that are observed due to load and store redundancy removal. In Figure 10 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 the savings are due to redundancy removal. The average IPC improvements range between 3% and 15% for different values of data cache access cycles. For programs with high levels of redundancy 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%.

We can compare the IPC results obtained for 2 cycle cache latency by our energy-efficient algorithm with those obtained by the aggressive energy-inefficient algorithm whose results were given in Figure 2. The IPCs we obtain with the energy-efficient algorithm are only slightly lower. However,

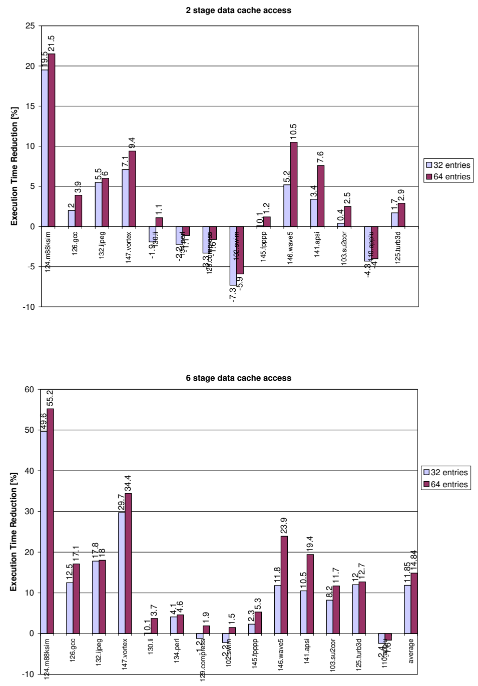


Figure 10: IPC Improvements for Varying Cache Latencies.

this algorithm saves energy while the aggressive algorithm greatly increased the amount of energy used.

In conclusion we have demonstrated that programs contain significant levels of load and store redundancy. By eliminating this redundancy we can achieve significant energy savings. We have presented a design of reuse unit that simultaneously provides execution speedups and energy savings.

References

[1] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," In *The 27th International Conference on Computer Architecture*, pages 83–94, May 2000.

[2] K. Ghose. Reducing power in superscalar processor caches using subbanking, multiple line buffers, and bit line segmentation. In *The ACM/IEEE Symposium on Low Power Electronics and Design*, pages 70–75, 1999.

[3] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *The 31st Annual IEEE-ACM International Symposium on Microarchitecture*, pages 216–225, December 1998.

[4] K. Lepak and M. H. Lipasti. On the value locality of store instructions. In *The 27th Annual International Symposium on Computer Architecture*, pages 182–191, Vancouver, Canada, June 10–14, 2000.

[5] A. Sodani and G. S. Sohi. Dynamic instruction reuse. In *The 24th International Conference on Computer Architecture*, 1997.

[6] N. Vijaykrishnan, M. Kandemir, M.J. Irwin, H.S. Kim, and W. Ye, "Energy-Driven Integrated Hardware-Software Optimizations Using SimplePower," In *The 27th International Conference on Computer Architecture*, pages 95–106, May 2000.