Load Redundancy Removal through Instruction Reuse *

Jun Yang
junyang@cs.arizona.edu
Dept. of Computer Science
The University of Arizona
Tucson, AZ 85721

Rajiv Gupta
gupta@cs.arizona.edu
Dept. of Computer Science
The University of Arizona
Tucson, AZ 85721

Abstract

Instruction reuse techniques have been developed to detect and remove redundancy at runtime. By maintaining the execution history of an instruction, reuse techniques detect if a subsequent execution of an instruction will yield the same result as its previous execution, and if this is the case, the result is made available to dependent instructions without executing the instruction. This approach eliminates **same instruction redundancy***, that is, redundancy across different dynamic instances of the same static instruction. However, the main limitation of existing instruction reuse techniques is that they do not detect or eliminate* **different instruction redundancy***, that is, redundancy across dynamic instances of statically distinct instructions.*

We present instruction reuse techniques for **load redundancy removal** *that eliminate both same and different instruction redundancy. We first present a study that shows that in addition to significant levels of same instruction redundancy (average of 20%), load instructions also contain high levels (average of 35%) of different instruction redundancy arising at other load or store instructions. We also describe studies that characterize the behavior of the redundancy and develop a hardware implementation guided by this characterization. Our experiments show that our techniques yield IPC improvements of up to 11% and reduces off-chip traffic due to cache misses by as much as 32% for* `SPECint95` *benchmarks.*

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

1 Introduction

Studies have shown that data values computed by various instructions can be predicted with high degree of accuracy for a wide range of general purpose programs [12]. The predictability of data values is a direct consequence of high levels of redundancy, including load redundancy, that is present in programs. Numerous research efforts have been aimed at detecting and removing redundancy through both microarchitectural mechanisms [4, 14, 15, 3, 5, 6, 9, 10] as well as compile-time optimizations [1, 2]. It is extremely important to eliminate load redundancy since loads are one of the main sources of performance degradation. The long latency of load instructions and even longer delays resulting from cache misses caused by loads result in the limiting of achievable IPCs (instructions completed per cycle).

There are two existing techniques that exploit the presence of load redundancy in programs for tolerating load latency and achieving improved performance in terms of execution speeds. They are *load (instruction) reuse* [4, 14, 15] and *load value prediction* [3, 5, 6, 9, 10]. *Load reuse* techniques maintain the execution history of a load instruction and use it to detect if a subsequent execution of an instruction will definitely yield the same result as its previous execution. If this is the case, the result is made available to dependent instructions without executing the instruction. Load value prediction techniques also maintain execution history of a load instruction but this history can only be used to predict the possible outcome of future executions of the load instruction. It is not possible to determine with absolute certainty that the predicted value is indeed correct. The predicted value is made available to dependent instructions so that their execution can proceed speculatively. At the same time the execution of the load instruction proceeds so that the correctness of value prediction can be confirmed later. If a misprediction is detected, dependent instructions must be reexecuted.

The key difference between the two techniques described above is that while load value prediction is speculative in nature, load reuse is nonspeculative. Therefore while load reuse eliminates the execution of redundant loads altogether, load value prediction cannot do so. Although load value prediction provides an effective way for tolerating load latency, it requires that the load instruction be executed in order to verify the correctness of the predicted value. The important consequence of this difference is that while both techniques can be used to tolerate long load latencies, only load reuse techniques cut down cache misses resulting from load instructions.

It is important to note that existing instruction reuse techniques only remove a specific class of load redundancies [4, 14]. In particular, load reuse eliminates *same instruction redundancy*, that is, redundancy across different dynamic instances of a specific static instruction. The main limitation of existing instruction reuse techniques is that they do not detect or eliminate *different instruction redundancy*, that is, redundancy across dynamic instances of statically distinct instructions. A load can also be eliminated using outcomes of other load or store instructions. Dependence prediction techniques have been proposed to exploit the presence of different instruction redundancy [7, 8, 13]. However, like value prediction, they are also speculative in nature and are therefore unable to reduce the number of cache misses.

We conducted experiments to measure the load redundancy present in `SPECint95` benchmarks. In these measurements each load was first tested for an opportunity for same load redundancy and only if this was not found, the load was then tested for different instruction redundancy, and finally for different store redundancy. Thus, the same load redundancy figures represent an upper bound on this type of redundancy. Figure 1 shows the results of this study. As we can see the average same load redundancy is 23% while different load and store redundancies are 14% and 29% respectively. Therefore it is quite clear that different instruction (load and store) redundancy must also be exploited since it is greater than the amount of same load redundancy.

The remainder of the paper is organized as follows. In section 2 we describe studies that characterize the behavior of load redundancy in terms of criteria that clearly influence a practical implementation of load reuse algorithms. In section 3 we develop a hardware implementation of load instruction reuse based upon the observations of section 2. This implementation eliminates both same and different instruction load redundancy. In section 4 we present results of experiments that show that our technique yields IPC improvements of up to 11% and reduce off-chip traffic due to cache misses by as much as 32% for some `SPECint95` benchmarks.

PC_1: Load R, addr
. . .
PC_1: Load R, addr
(a) Same Load Reuse.

PC_1: Load R, addr
. . .
PC_2: Load R, addr
(b) Different Load Reuse.

PC_1: Store R, addr
. . .
PC_2: Load R, addr
(c) Different Store Reuse.

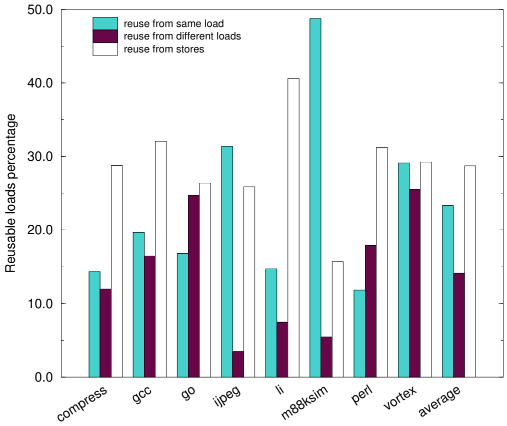

Figure 1. Available load redundancy.

2 Load Reuse Characteristics

Before we can design an effective implementation of a load reuse mechanism, we must understand the characteristics of load reuse, that is, we must understand the relationships among instructions from which reuse arises. Let us first consider the load reuse opportunities arising at a previous execution of the same load or a different load. At a high level the load reuse algorithm for exploiting such redundancy consists of two main steps: the *linking* step and the *exploitation* step. The *linking* step performs detection of *different load reuse* opportunities among instructions that *have been executed* and links a load with other load instructions with which load reuse was detected. The *exploitation* step uses the history of a load's own past execution as well as that of other loads linked to it to detect and exploit same and different load redundancy in load instructions that are *yet to be executed.* If redundancy is detected, the execution of the load instruction is avoided.

Corresponding to the two main steps described above, we design two hardware structures. The *linking structure* which maintains a trace of *recently executed* load instructions and uses this information to link a load with other load instructions that are part of this trace. The *exploitation structure* will contain the information on *past load reuse* and is used to avoid execution of future load instructions with same or different load reuse. We conducted some experimental studies to determine the organization and sizes of the above hardware structures.

2.1 Linking structure for different load reuse

In order to detect the existence of different load reuse, the detection structure is used to maintain information on the recently executed load instructions. When a new load instruction completes its execution, its outcomes (address and value of referenced memory location) are compared against the saved information to see if reuse existed between this load instruction and other recently executed load instructions. Our objective is to limit the amount of information saved while at the same time ensuring that most of the different load reuse opportunities are detected.

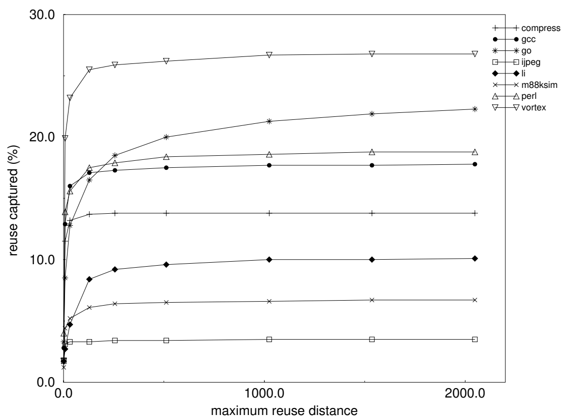

Figure 2. Reuse distances vs captured reuse.

We performed experiments in which the information on recently executed loads was maintained as a FIFO, called the *recent load queue* (RLQ), and the size of the FIFO was varied to determine a suitable size that was necessary to capture different load reuse. Given a sequence of dynamically encountered load instructions, the distance between two loads in the sequence involved in reuse is referred to as the *reuse distance.* The length of the FIFO limits the maximum allowable reuse distance, that is, reuse between loads instances that have a reuse distance greater than the length of the FIFO cannot be detected. Figure 2 shows the amount of reuse captured for different values of maximum allowable *reuse distance.* From the figure we can see that nearly all of the different load reuse opportunities have a reuse distance of less than 256 and therefore a FIFO of length 256 is sufficient.

2.2 Exploitation structure for same and different load reuse

As load instructions are executed, their execution history is stored in a *load table* so that their future executions can be avoided if they are found to be redundant. Histories of all loads are maintained in this load table. When a new load is encountered, its operands (values used in computing the effective address) are compared with the operands of selected loads in the load table. These selected loads correspond to the past execution of the same load for detecting same load redundancy and past executions of other loads that are linked to the current load in the linking step due to different load redundancy. If comparisons of operands yield a match and the value corresponding to the load is still valid, then the previously loaded value can be reused from the load table and the current load instruction need not proceed to memory.

There are two parameters that must be selected: the number of entries in the load table called the *load table size* and the maximum number of other loads to which a load can be linked due to observed different load redundancy called the *reuse set size.* We conducted experiments to determine the suitable load table size and reuse set size. First we conducted experiments that placed upper bounds on these sizes. Our experiments showed that in most benchmarks the number of static loads involved in reuse was less that 3K and almost all the different load reuse was captured by a maximum reuse set size of 16.

First we conducted experiments where we fixed the reuse set size to 16 and varied the load table size from 32 entries to 2K entries. The total load reuse, which includes both the same and different load reuse, captured by different table sizes is given in Table 1. As we can see, the amount of reuse captured increases with table size. However, for most of benchmarks, the increase in the amount of reuse captured is minimal as the size of the table is increased beyond 512 entries. Therefore we pick 512 as an appropriate size for our load tables.

Benchmarks	Table Sizes				
	32	128	512	1K	2K
compress	50.1	53.8	54.3	54.4	54.4
gcc	54.3	62.3	65.8	66.5	66.8
go	39.9	50.8	57.9	59.3	59.9
ijpeg	53.8	59.4	59.6	59.4	59.4
li	41.6	55.1	58.2	59.7	60.4
m88ksim	59.1	65.5	68.8	69.2	69.4
perl	45.7	51.9	54.5	55.3	55.7
vortex	65.0	74.7	77.6	79.2	79.7

Table 1. Load reuse with different table sizes and reuse set size of 16.

In order to determine an appropriate reuse set size, we fixed the load table size to 512 entries and varied the reuse set size from 2 to 16. The amount of load reuse captured by by various reuse set sizes is shown in Table 2. Again we can see that the amount of reuse captured increases with the reuse set size. However, for most of the benchmarks the increases in the reuse are less significant as the reuse set size is increased beyond 4. Therefore we choose 4 to be a reasonable reuse set size.

Benchmarks	Reuse Set Sizes			
	2	4	8	16
compress	54.3	54.3	54.3	54.3
gcc	63.0	64.1	65.5	65.8
go	45.6	50.2	55.0	57.9
ijpeg	59.4	59.6	59.6	59.6
li	56.3	58.2	58.2	58.2
m88ksim	67.8	68.4	68.8	68.8
perl	52.0	53.9	54.5	54.5
vortex	72.3	74.4	76.9	77.6

Table 2. Load reuse with different reuse set sizes and table size of 512.

2.3 Different store reuse

In order to detect different store reuse we must also maintain history of recently executed stores. One approach is to save both information on both recent loads and stores in the same FIFO. However, through separate experiments we observed that this is not a good choice because often it led to the FIFO being filled entirely by stores and therefore different load reuse opportunities could not be detected. Therefore it is important that the recent history of stores should be maintained in separate FIFO, the *recent store queue* (RSQ). We also observed that reuse distance for different store reuse was significantly smaller than that required for different load reuse. Therefore instead of using a linking structure, we simultaneously compare the load with a small number of entries in the RSQ to detect and exploit different store reuse opportunities.

3 The Hardware Design

There are two main components of the resue mechanism: one is responsible for exploiting load reuse opportunities originating at same or different load instructions while the other is responsible for exploiting reuse originating at other store instructions. We first describe these two components and then conclude this section by describing how the load reuse mechanism is integrated into a superscalar pipeline.

3.1 Reuse originating at same/different loads

The hardware for detecting load reuse opportunities originating at previous executions of same or different loads is shown in Figure 3. The contents and use of the three main components of this hardware are described next:

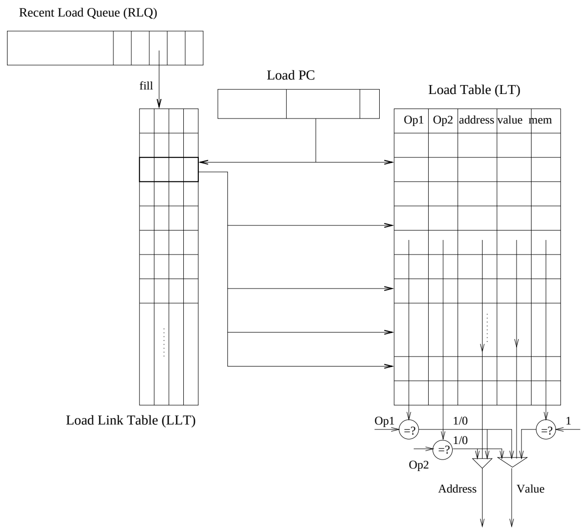

Figure 3. Load table design.

The **Load Table** (LT) is used to store the execution history of each load encountered during execution. There is at most one entry per static load and the table is consulted using an index derived from the PC of the load instruction. The execution history of a load consists of the dynamic operands from which the address is computed, the computed address, and the loaded value. In addition there is *mem* bit which indicates whether the loaded value is valid or whether it is invalid because a store to the same address has been encountered since. The *mem* bit must be invalidated as soon as a store instruction to the same address is encountered. At this point the value that will be stored by the store instruction is in general not known.

When a load instruction is encountered during execution, its LT entry is consulted to look for same load reuse. The operands of the load instruction are matched against those stored in its LT table entry. If the operands match, and the *mem* bit is valid, the load address and value can be directly taken from the LT entry. Thus, there is no need to compute the load address or fetch the loaded value from memory. In other words the execution of the load instruction is now complete. If the operands match but the *mem* bit is invalid, we can reuse the address but not the value. Thus, the address computation can be avoided but the value must be loaded from memory.

The **Load Linking Table** (LLT) is used to link a load with other loads in presence of different load reuse opportunities. The LLT is consulted using an index derived from the PC of the load instruction. An entry corresponding to a load consists of the reuse set of the load. Thus LT indices of up to 4 other loads can be found in this entry. The histories of these loads are obtained from the LT and also compared with the operands of a load encountered during execution. If a suitable match occurs, the execution of the load can be avoided in the same manner as described earlier for same load reuse.

The **Recent Load Queue** (RLQ) is used to detect different load reuse opportunities from past executions and fill in the LLT so that these can be exploited in the future. The RLQ operates as a FIFO which contains histories for a fixed number of most recently executed loads. When a load which could be reused completes its execution, the history of this load is compared with that of other loads in the RLQ. If a match is found, the LT index of the load at which reuse originates is entered into the reuse set of the executed load. Therefore if the same load is encountered in the future, an attempt will be made to see if its execution can be avoided using a different load.

3.2 Reuse originating at different stores

If the current load cannot find reuse from a previous load, either directly through LT or indirectly through LLT, it attempts to reuse address and value from a recently executed store. For this purpose we maintain a *recent store queue* (RSQ) which also functions as a FIFO (see Figure 4). The RSQ holds the history of the most recently seen stores. The difference between RSQ and the standard store queue is that RSQ contains the operands from which address is computed so that the address calculation can be reused in addition to the value. A *mem* bit is maintained so that only the value from the latest store to an address can be reused. As new stores are encountered, previous stores to the same addresses in the RSQ are invalidated.

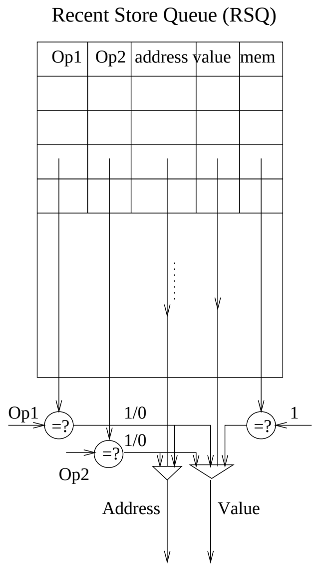

Figure 4. Store queue design.

3.3 Integration of reuse mechanism into superscalar pipeline

The pipeline of the superscalar machine model used in our work is shown in Figure 5. The load operands are compared against LT table entries and the RSQ in the register read stage since the operands are first known at this time. If the load can reuse the address from LT or RSQ, it then skips the execute bypass stage and goes directly to data cache access stage. At the beginning of the data cache access stage we check the *mem* bits to see if the value can be reused from the LT or the RSQ. If the *mem* bit is valid the instruction can skip the data cache access stage and can proceed to wait for committing. In addition, the load value is supplied to waiting instructions through bypass paths. When the load instruction is found not to be reusable, the load instruction will flow through the data cache access stage just as in normal execution. In this case, at the end of the data cache access stage, the LT table will be updated. The RSQ and the LT entries are invalidated by stores that have the same addresses computed at the execution stage. Since a load may finish execution after a store that appears after the load in program order, the load does not update the LT entry which has already been invalidated by such a store.

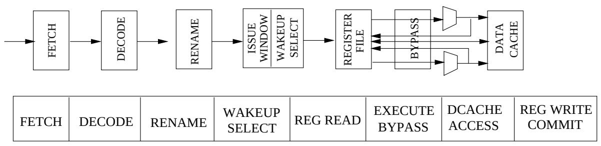

Figure 5. Superscalar machine pipeline.

4 Experimental Results

We conducted experiments to study the impact of our load reuse technique on both the performance in terms of IPCs and the memory traffic caused by cache misses. In these experiments the superscalar configuration shown in Figure 6 was used. The processor has an eight stage pipeline and can issue four instructions per cycle. The MIPS-I instruction set was used and all programs were compiled using the `gcc 2.7.2` compiler with `-O3`, the highest, level of optimization. We assume an on-chip cache and any cache misses result in off-chip memory traffic. The simulators used in this work were generated automatically from machine descriptions written in ADL (architecture description language) using the FAST system developed by us [11].

Execution mechanism	4-issue per cycle, out of order execution, in order commit, central window algorithm, 32 entry reorder buffer, 32 entry load/store queue.
Branch predictor	perfect branch predictor
Instruction cache	16K bytes, directed mapped, 32 byte cache line, 6 cycle miss latency
On-chip data cache	16k bytes, 32 byte cache line, 6 cycle miss latency, 64 bit memory path
Load-reuse mechanism	512 entry load table, reuse set size of 4, 256 entry recent load queue

Figure 6. Superscalar machine configuration.

4.1 Load reuse captured

First we report on the amount of reuse captured by our load reuse implementation for `SPECint95` benchmarks. As shown in Figure 7(a), the average amount

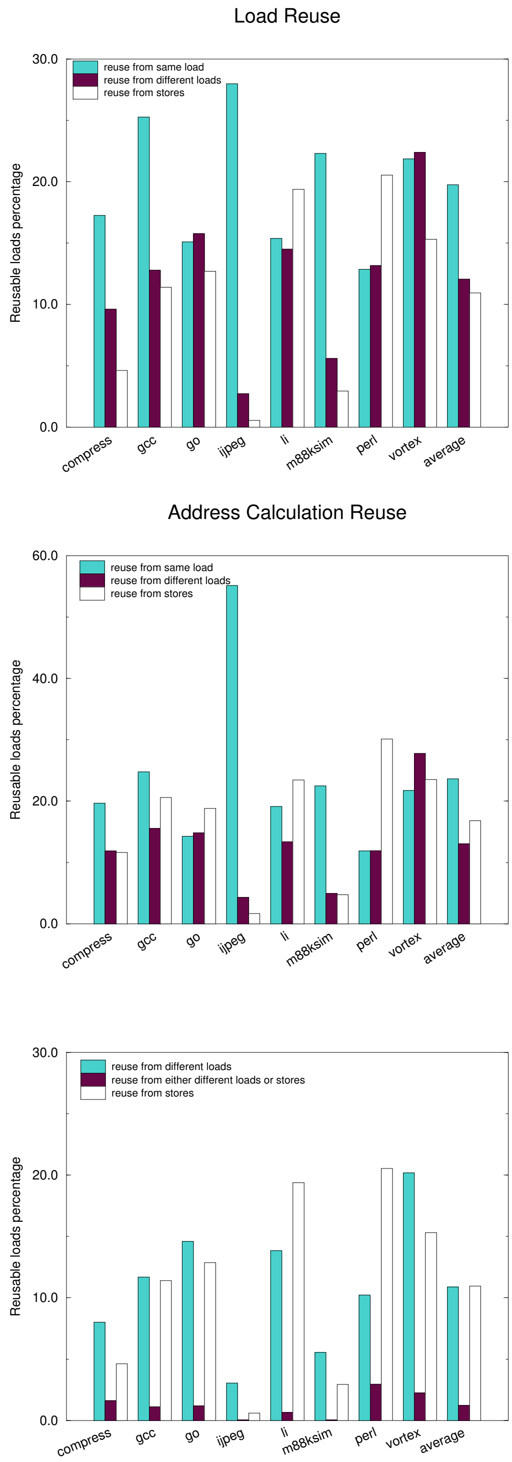

Figure 7. (a) Load reuse; (b) Reuse of address calculations; and (c) Different load reuse from: i. load only, ii. load or store, and iii. stores only.

of same load reuse is approximately 20%. This is nearly equal the same load reuse achieved by a technique proposed in [14]. However, unlike the technique in [14], we also able to achieve an average of additional 12% of different load reuse and 11% of different store reuse. Thus, the total average amount of load reuse that we achieve is nearly 43%. Thus, we have more than doubled the amount of load reuse in comparison to existing techniques.

It should also be noted that address calculations can also be used while performing load reuse. Each time load value is reused, the address calculation is also reused by our technique. The average amount of address calculations reused by the technique in [14] is approximately 21%. Since we also perform different instruction reuse, we are able to use an average of nearly 54% which is also much greater (see Figure 7(b)).

Finally we analyzed the situations of different instruction reuse further to understand the relative importance of different load reuse and different store reuse. We categorized different instruction reuse into three categories: reuse that can only be exploited from another load, reuse that can be exploited from either another store or another load, and reuse that can only be exploited from another store. As the results in Figure 7(c) show, we found that on an average nearly equal amounts of the reuse, approximately 11% each, fell into the first and third categories while the second category accounted for a little over 1% of the reuse. Also recall that none of the opportunities for different instruction reuse could have been captured as same load reuse. Thus, we can conclude that it is important to exploit both different load reuse and different store reuse opportunities since they are substantial and distinct.

4.2 IPC improvements

There are two factors that contribute to IPC improvements from load reuse: the reuse of address calculations and the reuse of the loaded value itself. We took advantage of both these aspects in our implementation. The resulting IPC improvements are shown in Figure 8. The average improvement was 5% and the maximum improvement of 11% was observed for the `m88ksim` benchmark.

4.3 Memory traffic reduction

We measured changes in the overall off-chip memory traffic caused by cache misses. Since load reuse reduces cache accesses, the number of cache misses is reduced. These reductions were found to be significant. As shown in Figure 9, the average reduction in off-chip traffic is 17% and for the `m88ksim` benchmark the reduction is as high as 32%.

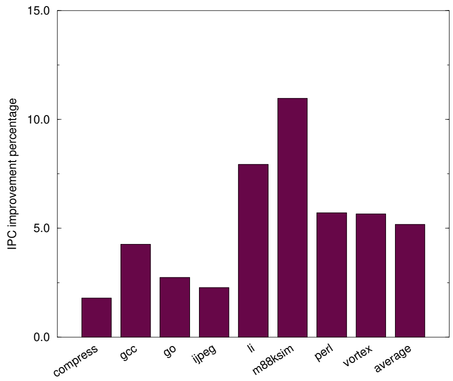

Figure 8. IPC improvements.

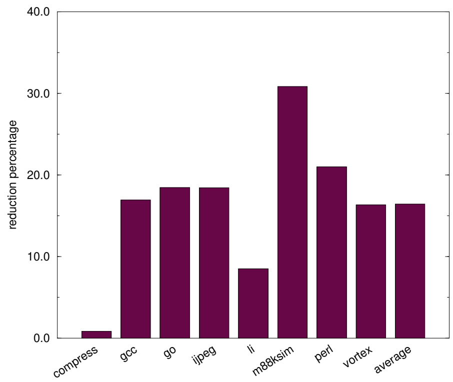

Figure 9. Reduction in off-chip memory traffic due to cache misses.

4.4 Power consumption

In the systems being designed today power consumption is an important issue since often these systems are battery powered. The introduction of hardware structures for load reuse will consume additional power. However, the additional power consumed may be offset by other reductions cause by load reuse. First, the power consumed by external buses is reduced due to lower off-chip memory traffic. Second the switching of the address bus due to the placing of load addresses on the address bus is also reduced. The results in Figure 10 show that by preventing redundant loads from proceeding to the cache, our load reuse techniques reduces the address bus switching activity on an average by nearly 27%. For the `vortex` benchmark the reduction is around 37%. Thus both the reductions in memory traffic and switching activity is substantial.

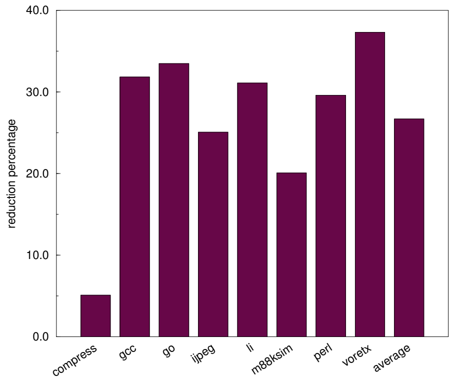

Figure 10. Reduction in address bus switching activity.

5 Conclusion

We have demonstrated that significant levels of load reuse opportunities exist in `SPECint95` benchmarks and these opportunities can be effectively detected and exploited through microarchitectural mechanisms. In particular, this paper makes the following contributions.

- We demonstrated that significant amounts of different instruction reuse is present in programs which goes unexploited by existing instruction reuse techniques. By designing a new implementation for load reuse we are able to capture and exploit both same and different instruction reuse opportunities for load instructions.

- We demonstrated that significant IPC improvements are observed when load reuse is exploited. Moreover substantial reductions in address bus switching and off-chip traffic are also achieved by our load reuse mechanism.

References

[1] R. Bodik, R. Gupta, and M.L. Soffa, "Load-Reuse Analysis: Design and Evaluation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 64-76, Atlanta, Georgia, May 1999.

[2] R. Bodik and R. Gupta, "Array Data-Flow Analysis for Load-Store Optimizations in Superscalar Architectures," *International Journal of Parallel Programming*, special issue on Languages and Compilers for Parallel Computing, Vol. 24, No. 6, pages 481-512, 1996.

[3] F. Gabbay and A. Mendelson, "Speculative execution based on value prediction," *EE Department TR#1080, Technion - Israel Institute of Technology*, Nov. 1996.

[4] S.P. Harbison, "An Architecture Alternative to Optimizing Compilers," *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 57-65, March 1982.

[5] M.H. Lipasti, C.B. Wilkerson and J.P. Shen, "Value Locality and Load Value Prediction," *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138-147, Oct. 1996.

[6] M.H. Lipasti, "Value Locality and Speculative Execution," *Ph.D. Thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University*", May 1997.

[7] A. Moshovos and G. Sohi, "Streamlining Inter-Operation Memory Communication via Data Dependence Prediction," *30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 235-245, Dec. 1997.

[8] A. Moshovos and G. Sohi, "Read-After-Read Memory Dependence Prediction," *32th Annual ACM/IEEE International Symposium on Microarchitecture*, Nov. 1999.

[9] T. Nakra, R. Gupta, and M.L. Soffa, "Value Prediction in VLIW Machines," *ACM/IEEE 26th International Symposium on Computer Architecture*, pages 258-269, Atlanta, Georgia, May 1999.

[10] T. Nakra, R. Gupta, and M.L. Soffa, "Global Context-based Value Prediction," *Fifth IEEE International Symposium on High Performance Computer Architecture*, pages 4-12, Orlando, Florida, January 1999.

[11] S. Onder and R. Gupta, "Automatic Generation of Microarchitecture Simulators," *IEEE International Conference on Computer Languages*, pages 80-89, Chicago, Illinois, May 1998.

[12] Y. Sazeides and J.E. Smith, "The Predictability of Data Values," *30th Annual ACM/IEEE International Symp. on Microarchitecture*, pages 248-258, Dec. 1997.

[13] Y. Sazeides, S. Vassiliadis and J.E. Smith, "The performance potential of data dependence speculation and collapsing ," *29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 238-247, Dec. 1996.

[14] A. Sodani and G. Sohi, "Dynamic Instruction Reuse," *24th Annual International Symposium on Computer Architecture*, pages 194-205, July 1997.

[15] A. Sodani and G. Sohi, "Understanding the Difference Between Value Prediction and Instruction Reuse," *31th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 205-215, Dec. 1998.