Load and Store Reuse Using Register File Contents

Soner Önder
Department of Computer Science
Michigan Technological University
Houghton, MI 49931-1295

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

ABSTRACT

The detection of opportunities for value reuse optimizations in memory operations require both the addresses and values associated with these operations to be available. Although the values are typically available in the physical register file, their presence cannot be exploited because no correspondence between the values and addresses is maintained. In this paper we propose the explicit management of the physical register file contents as a level in the memory hierarchy by supporting the *Value Address Association Structure* (VAAS). The entries in VAAS have a one-to-one correspondence with entries in the physical register file. For each value in the register file that is involved in a load or store operation, the associated information, including the memory address, are stored in the corresponding VAAS entry. Several optimization tasks can be performed using the combination of physical registers and VAAS.

Specifically VAAS enables unified implementation of the following optimization tasks: (i) *Store-to-load forwarding* is performed without explicitly saving the stored values; (ii) *Load-to-load* forwarding is performed without saving loaded values in a reuse buffer; (iii) *Silent stores* are eliminated without saving or loading the prior value stored to the same addresses; (iv) *Switching of bits in L1 cache* is minimized without saving additional history; and (v) *False memory access order violations* are avoided without holding speculatively loaded values in the speculated loads table.

Our experiments demonstrate that our implementation of non-speculative optimizations is highly effective as it eliminates memory references due to 60% (58%) of loads in `SPECint95` (`SPECfp95`) and 25% (22.6%) of stores in `SPECint95` (`SPECfp95`). On an average over 45% of cache references are eliminated due to non-speculative reuse. On an average the L1 switching activity was reduced by 7.75%.

1. INTRODUCTION

Since frequent memory operations (loads and stores) represent a critical limiting factor to achieving high performance on modern superscalar processors, a great deal of research has focussed on techniques for effectively handling memory operations. These techniques include aggressive load speculation for tolerating load latency [2, 17] as well as a variety of speculative and non-speculative load and store reuse removal techniques [5, 8, 9, 12, 13, 14, 15, 18, 21]. Together they provide an impressive array of techniques for dealing with memory operations. In order to take full advantage of these techniques one would like to incorporate these techniques in a realistic superscalar design. However, the complexity of such an endeavor is very high because each of these techniques uses its own specialized hardware structure to maintain the state needed for carrying out the respective optimization task.

Load reuse techniques, such as load value prediction, have another dimension of complexity to them – they are speculative in nature [5, 9]. With an increasing emphasis on power-aware processor design for the mobile computing environment, it is important to make judicious use of speculation [10]. While branch prediction is essential for achieving high performance, we believe that it may not be necessary to employ speculative techniques for load reuse. Instead non-speculative load and store reuse techniques can be applied to aggressively perform load and store reuse. A case for employing non-speculative value reuse techniques instead of value prediction techniques was made in [18]. The techniques we describe in this paper are useful in both the presence and absence of load speculation.

In this paper we present a simple unified solution for several optimization tasks associated with memory operations which uses a single simple hardware structure for carrying out multiple tasks. The key idea of our approach is to manage the contents of the physical register file as a level in the memory hierarchy. This effectively means that we must maintain address associations for values that are contained in the physical registers. Nearly all of the existing techniques for various optimizations of memory operations maintain such an association in some dedicated hardware structure. Moreover the values are stored in this structure even if they are present in some physical register and typically each optimization uses a structure that is specially designed for it. We maintain a single dedicated structure, called the *Value-Address Association Structure* (VAAS), for maintaining address associations of loaded and stored values present in the physical register file. The VAAS acts as a shared structure for implementing multiple optimizations. Moreover it exploits data values present in the physical registers to carry out these optimizations. These optimizations are applicable in context of uniprocessors and are not always applicable to multiprocessor systems. As we demonstrate in this paper, the VAAS can be integrated in an out-of-order superscalar both in absence and presence of load speculation.

The remainder of the paper is organized as follows. Before we discuss our techniques in detail, we provide an overview of non-speculative optimizations performed using VAAS in section 2. In section 3 we describe the implementation of VAAS. We also give the runtime algorithm for maintaining VAAS such that at any given time all existing address associations are valid, that is, the value in the physical register file represents the true current value corresponding to the associated address. We also discuss timing issues related to the implementation of all of the optimizations in context of a superscalar processor design which does not support load speculation. In section 4 we present results of experiments performed to evaluate the impact of non-speculative optimizations for `SPEC95` benchmarks. In section 5 we discuss the use of VAAS in the presence of load speculation. We discuss how the previously described non-speculative optimizations can now be speculatively performed and also the new optimization of avoiding memory access order violations can be performed. Additional related work is discussed in section 6 and the concluding remarks are given in section 7.

2. VALUE REUSE BASED NON-SPECULATIVE OPTIMIZATIONS

Let us briefly examine the optimizations that exploit the VAAS in the absence of load speculation. We highlight the differences between existing implementations [8, 18] of the optimizations and VAAS based implementations.

2.1 Store-to-load Forwarding for Load Avoidance

This optimization is performed by forwarding values from prior store operations, to a (redundant) load. In existing superscalar designs the values from prior stores are forwarded to pending loads through the forwarding buffer which contains address-value pairs corresponding to stores which have not yet retired. However, once a store retires, the forwarding buffer entry is deallocated and the value stored in memory cannot be forwarded to future loads from the forwarding buffer. In contrast, our solution relies on the physical register file for values and forwarding of values from retired stores to a redundant load is performed as long as the value is present in a physical register. The information typically held in the forwarding buffer is contained collectively in the physical register file and the VAAS. Therefore no dedicated forwarding buffer is required for store-to-load forwarding. On an average our technique is able to avoid memory reads corresponding to 27% of loads through store-to-load forwarding in `SPEC95` benchmarks.

2.2 Load-to-load Forwarding for Load Avoidance

This optimization is performed by forwarding values from prior load operations to a redundant load. This optimization is named as the *load reuse* optimization. In order to hold values for potential reuse a dedicated structure called the *load reuse buffer* is maintained. The technique in [18] only allows reuse between different instances of the same load instruction. This is because the reuse buffer is indexed using an instruction's PC value. An additional *load linking table* is supported in [21] to link loads with different PC values and thus achieve reuse across instances of loads with different PC values. VAAS based approach has two advantages. First our implementation does not maintain a reuse buffer to save values but simply obtains them from the register file. Second PCs do not play any role in detecting or exploiting reuse opportunities. Therefore we do not need to distinguish between load reuse opportunities arising from dynamic instances of same static load instruction and dynamic instances of statically distinct load instructions. On an average our technique is able to avoid memory reads corresponding to 32% of loads through load-to-load forwarding in `SPEC95` benchmarks.

2.3 Removal of Silent Stores

This is another important optimization that can greatly reduce the number of stores that write to memory. If the value to be written by a store is the same as the value already present at that address, then the store is redundant. One approach to carry out this optimization is to maintain a structure to hold the history (values and corresponding addresses) of recently executed loads and stores. Another approach recently proposed by Lepak and Lipasti [8] loads the value from a memory location and if it is different from the value to be stored, the store is allowed to write to memory. Therefore for every store that is encountered one load must also be performed. Our implementation eliminates silent stores whose redundancy can be established from the values in the physical register file and their associated addresses in VAAS. The value may be present in a physical register due to a prior load or a store to the same address. In contrast to [8], we do not require execution of any additional load operations. Our technique is very effective because on an average we are able to eliminate memory writes associated with nearly 25% (22.6%) of the stores in `SPECint95` (`SPECfp95`) benchmarks.

2.4 Power Optimization for L1 Cache

This goal is achievable using VAAS because it allows support of a mechanism for lowering the switching activity caused by writes to the L1 cache. Reduction in switching is achieved by providing the option of storing data values either in their original form or in their bitwise complimented form. We choose to store the form which has a lower Hamming distance from the old value present in memory. The optimization can only be performed in those cases where the old value can be found in one of the physical registers. The cache is modified by providing an additional bit per word which indicates the form in which the value is stored (complimented or original). Note that the removal of redundant stores does

PC1:	St R1, Addr1
	......
PC2:	St R2, Addr2
	......
PC3:	Ld R3, Addr3
	......
PC4:	Ld R3, Addr4

Address-Only Optimizations	Enabling Condition	Existing Technique
Ld R3, Addr3 is redundant due to St R2, Addr2.	Addr2 = Addr3	*Forwarding buffer* supplies values to loads from un-retired stores.
Ld R3, Addr4 is redundant due to Ld R3, Addr3. PC3 = PC4 $\Rightarrow$ self reuse. PC3 $\neq$ PC4 $\Rightarrow$ different load reuse.	Addr3 = Addr4	*Load reuse buffer* supplies values to redundant loads [18]. Different load reuse requires an additional *load linking table* [21].

Value-Address Optimizations	Enabling Condition	Existing Technique
St R2, Addr2 is value redundant due to St R1, Addr1.	Addr1 = Addr2 *AND* Value(R1) = Value(R2)	Value at Addr2 is *loaded* from memory for verification in [8].
Reducing switching of bits in the L1 cache by storing compliment of the value.	Addr1 = Addr2 AND Value(R1) $\neq$ Value(R2)	None.

Figure 1: Illustration of optimizations enabled by VAAS.

not eliminate any switching activity because by definition they do not cause any switching. This is an important optimization because on-chip cache memories already consume nearly 30% of total power consumed by a processor. This number is expected to grow further as larger caches are being put on future processors. Moreover the amount of power consumed by the L1 cache is over three times greater than an on-chip L2 cache due to the high degree to switching activity in the L1 cache. Our experiments show that on an average the degree of switching activity is reduced by 6.59% (8.67%) for `SPECint95` (`SPECfp95`) benchmarks.

The various optimizations described above are illustrated in Figure 1. Along with an illustration of each optimization, the precise conditions under which the optimizations are applicable are given and existing techniques are also listed. The optimizations are separated into two categories (described in two separate tables in Figure 1). The applicability of optimizations in the first category is determined simply by checking addresses and the values are used only if the optimization is found to be applicable. In contrast, both addresses and values are needed to determine the applicability of the optimizations in the second category. If the optimization is found to be applicable, values are not explicitly used because these optimizations simply eliminate operations that otherwise would have been performed.

3. VALUE-ADDRESS ASSOCIATION

From the conditions for optimization opportunities described in Figure 1, it is clear that address-value associations are needed to detect such opportunities. In fact existing techniques for some of such optimizations explicitly maintain such associations. For example, the load reuse buffer maintains address-value associations encountered during recently executed load instructions and the forwarding buffer maintains address-value associations encountered by store instructions that are yet to be committed.

3.1 VAAS Structure and its Use

The structure of VAAS that we maintain for remembering valid address-value pairs is shown in Figure 2. It consists of an *address CAM* and some additional fields of information. There is a one-to-one correspondence between the entries of the physical register file and the address CAM. For each physical register that contains a loaded or stored value, the corresponding address is contained in the corresponding address entry of the CAM. The valid bit indicates whether the CAM entry contains a valid address. To perform the switching optimization task an additional item is also needed. In particular, we also remember whether the value stored in memory in its original form or complimented form using the C bit.

It is quite easy to see how the VAAS is used to perform optimization of *load* operations. When a load is encountered we search for the memory address involved in the address CAM. If a hit occurs (i.e., address match occurs with an entry whose valid bit is set), we do not need to load the value from memory but instead directly obtain the value from the corresponding physical register. In this way we implement *store-to-load* as well as both types of *load-to-load* forwarding. In fact it is important to note that our implementation is greatly simplified because we do not distinguish between the types of sources for the values. Existing techniques explicitly categorize and separately handle the three different types of sources: *forwarding buffer* for *store-to-load* forwarding, *load reuse buffer* for *load-to-load* forwarding between instances of the same static load [18] and *load linking*

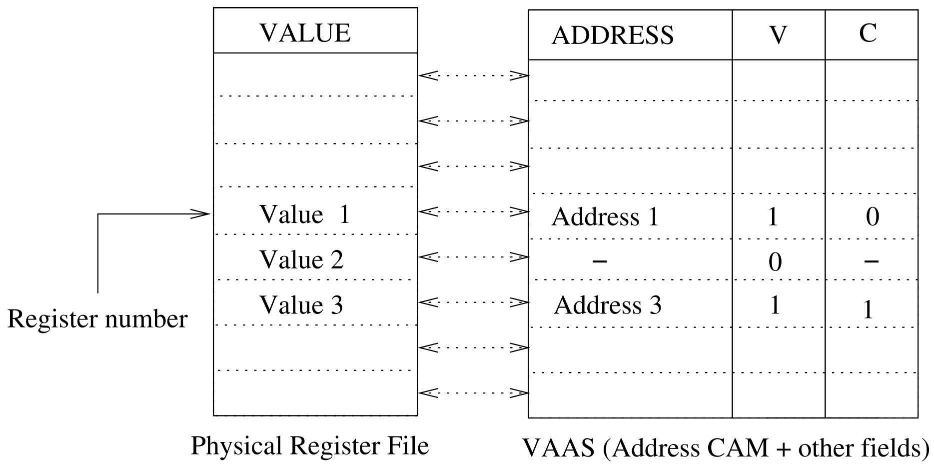

Figure 2: The VAAS structure.

table together with the *load reuse buffer* for *load-to-load* forwarding between load instances of statically distinct loads [21].

Now let us consider the optimizations involving *store* operations. A *redundant store* is detected as follows. If the address to which store is being performed hits in the address CAM, the value at the corresponding entry in the physical register file is read and compared with the value to be stored. If the two are the same, the store commits without writing the value to memory. Note that the entry at which the hit occurs may have been created by an earlier load from the same address or an earlier store to that address. Much like load reuse detection, silent stores are detected without distinguishing between the different types of sources (earlier load or store operations) due to which they arise.

3.2 Maintaining VAAS Contents

In the above discussion we showed how the contents of VAAS are used to perform optimizations. Now we discuss how the contents of VAAS are maintained. If a memory location has not been updated, and the address is found in the address CAM, then the most recent value corresponding to the memory location is either already available or will become available in the corresponding physical register. However, if the memory location contains the latest value, our technique must guarantee that if the address is also present in the address CAM, then corresponding physical register contains the same value as the memory location. In other words we ensure that the contents of valid entries in the VAAS are consistent with the contents of the memory in the above situation. In this sense the contents of physical register file and VAAS are managed like a write-through cache where the cache contents are always consistent with memory contents. The above consistency is easily achieved by writing the values due to stores to memory when the store commits (i.e., writing due to stores is never delayed). By delaying stores it is possible to eliminate some dead stores – the value corresponding to the latest store to an address can be written while the values from prior stores to the same address can be simply discarded. However, in this work we only eliminate value redundant stores and not dead stores.

The conditions under which valid address entries are created in the CAM and later invalidated are discussed next.

Creation of valid entries.
A valid entry is created when a memory operation is encountered:

By loads. Once the address from which the value is to be loaded is known, a valid entry is created in VAAS by filling in the ADDRESS field and setting the valid bit to 1 to indicate that a valid address is present. When a load reaches the write-back stage, the loaded value is available and is thus written into the assigned physical register.

By stores. A store is handled in a similar fashion as a load. When the address to which a value is to be stored is known, the ADDRESS and other fields of the entry corresponding to the physical register from which the value is to be stored are set up creating a valid entry. The value in the physical register may become available later in time and is stored in the physical register as soon as it is known.

Invalidation of entries.
Again there are two conditions under which an entry is invalidated.

By non-memory instructions. It is possible that the physical register assigned to an instruction, which is not a load or a store instruction, corresponds to a valid entry. When such a non-memory instruction reaches the write-back stage, the address entry is invalidated since the register no longer contains a loaded or stored value. This invalidation condition can be further restricted through value reuse. If the register contains the same value that is being written into it, the entry need not be invalidated. However, a register read and a compare would have to be performed in order to carry out this optimization.

Upon branch misprediction. Upon branch misprediction, the pipeline is drained and resources allocated to misspeculated instructions are deallocated. When the physical register allocated to a misspeculated instruction is freed and added to the free list, the valid bit is set to 0 to indicate that the physical register is no longer associated with the address in the address field.

3.3 Lifetimes of Loaded/Stored Values

The longer we can keep loaded or stored values in registers, the greater is the degree to which future memory operations can be optimized. Our technique promotes optimization in a number of ways. First one should note that according to the rules for creating and destroying valid entries, it is possible for multiple valid entries corresponding to the same address to be created (e.g., when multiple loads to the same address are encountered). Thus, even if one of the entries containing the value is invalidated, the value may continue to be available through other entries.

Second the values stay around longer in the physical registers than they stay in a conventional *forwarding buffer*. This is because as soon as a store or load is committed, the value is no longer available from these structures. However, the value continues to be available from the physical register after the commit. On the other hand, a value may stay longer in the *reuse buffer* than in a physical register. Therefore we incorporate the following policies for extending the lifetimes values in physical registers.

Tracking contents of freed registers. When an instruction retires normally, that is, no misspeculation has occurred, although the physical register associated with the instruction is freed and added to the free list, the valid bit is not unset. Thus, the value corresponding to the address is still available for reuse.

Tracking contents of reassigned registers. The invalidation of a valid entry in VAAS is delayed as long as possible using an observation made by Monreal et al. [11]. When a free register is assigned to an instruction, its contents are not invalidated immediately. Only when the instruction reaches the write-back stage, the entry is invalidated.

Register reassignment policy. Finally a policy for reassigning freed registers to new instructions has been designed to extend the lifetimes of useful loaded or stored values in deallocated registers. A partitioned free list for physical register entries is maintained. One partition tracks free registers which contain valid values that have been loaded or stored at memory addresses while the other partition contains free registers that were most recently assigned to instructions other than memory operations and therefore they do not contain reusable values. Assignments are made from the latter category first. This strategy naturally tends to increases the likelihood that useful values will continue to survive in physical registers for longer periods of time.

3.4 Integrating VAAS into a Superscalar

The design of the VAAS structure presented in this section is suitable for integration in any superscalar design. However, the actions that must be taken when loads and stores are encountered vary depending upon whether or not load speculation is being carried out. In this section we show how to integrate VAAS into an out-of-order superscalar processor where load speculation is not being performed. Thus, loads can only proceed out-of-order with respect to preceding stores once addresses of preceding stores and the load have been generated and no dependence has been detected. If a dependence is detected then load reuse through forwarding is performed using VAAS.

VAAS is integrated into the superscalar pipeline by inserting a new stage called the *memory disambiguation and reuse detection* (MDRD) stage right before the regular data cache access stage (see Figure 3). Memory instructions are allowed to proceed to this stage from the issue window upon completing their address computations *in program order*. Both load and store addresses are entered into the VAAS structure when instructions move into the MDRD stage regardless of the availability of data. Since the instructions do not need to wait for their data operands, if they are not yet available, MDRD is organized as a small reservation station where instructions can wait for their data values to become available, if necessary. The width of MDRD stage is equal to the number of instructions at the head of the reservation station. In order to ensure that all of these instructions can be simultaneously checked for independence or dependence, the number CAM ports provided is equal to the width of the MDRD stage.

Now let us consider the operation of MDRD in greater detail and show how the load and store operations use and update the VAAS contents. Before we describe these details, we should mention that our optimizations are only applied in context of full-word memory operations. Non full-word stores (such as byte accesses) invalidate the entries found in the VAAS upon an address hit and non full-word loads proceed to the DCACHE stage because we do not exploit value reuse at byte level.

Handling Loads.
The execution of a load proceeds according to where the value associated with load address currently resides. The value may be present *only in memory*, *only in VAAS*, or *both in memory and VAAS*. When a load instruction is encountered MDRD accesses VAAS through a CAM port. In the first case a miss occurs while in the latter two cases a VAAS hit occurs. Moreover when a hit occurs, the value associated with the address may already be present in the destination register, and hence VAAS, or the write to the register may be pending. Next we describe how each of the above situations is handled in greater detail.

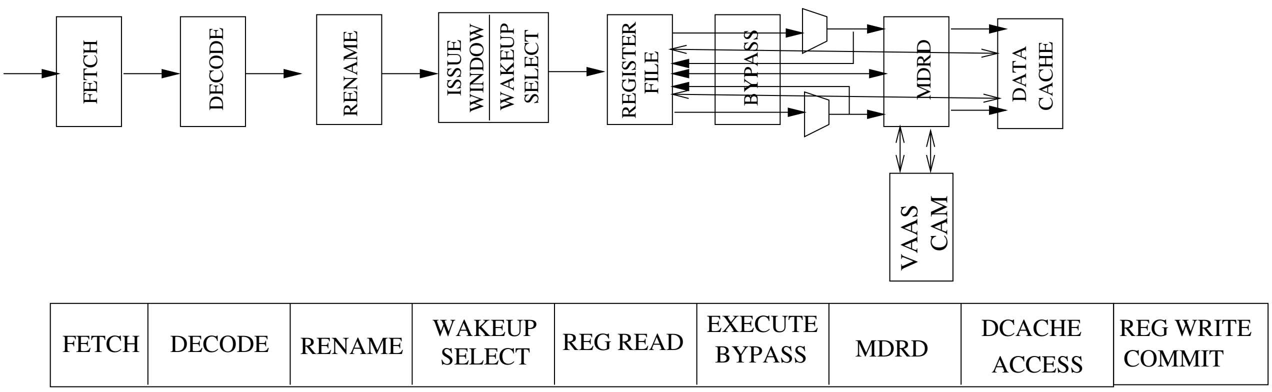

Figure 3: Superscalar pipeline incorporating VAAS.

VAAS miss. If a miss occurs the calculated address is entered into the VAAS and the load proceeds to the regular DCACHE access stage. Once the value arrives and is placed in the register it is also available for use by subsequent memory instructions involving the same address.

VAAS hit and value present. If a VAAS hit occurs and the data value is already available in the corresponding register, the value is copied into the destination register for the load and the load commits. This copying of value implements *load-to-load forwarding* if the creator of the entry was a load and it implements *store-to-load forwarding* if the creator of the entry was a store.

VAAS hit and value absent. In case of a VAAS hit it is also possible that the value is not yet available in the corresponding register because the producer of value has not written it to the register yet. In this case the load instruction is made dependent on the appropriate producer of the data so that as soon as the value becomes available it is directly forwarded to the waiting load instruction. The register from which the load instruction in question is to receive its value may be written to by: (a) an *earlier load* instruction; (b) an *earlier store* instruction; or (c) an instruction which performs an *ALU operation*. This is how each of these cases are handled:

- *Earlier load.* If the producer is an earlier load instruction then by making the current load instruction dependent upon the producer load instruction, we achieve *load-to-load* forwarding. It should be noted that if multiple loads to the same address are encountered before the producer load can make the value available, then all of these loads will become dependent upon the same producer load and therefore all of them will proceed simultaneously when the value becomes available.
- *Earlier store.* In case of a VAAS hit where the entry was created by an earlier store instruction, the load instruction is made dependent upon the producer of the value to be stored. The producer may be an earlier load or an ALU operation. In either case the direct linking of the load to this earlier instruction goes beyond *store-to-load* forwarding and actually achieves complete *bypassing* of both the memory and the store instruction. Again if multiple load instructions loading from the same address are encountered before the value is available, they will all be made dependent on the producer of the data value which can be another load instruction or the producer of a store instruction.
- *ALU operation.* If the producer is an ALU operation, then a hit indicates that there is an intervening store operation that created an entry in the VAAS where the hit occurred. The load instruction is made dependent upon the ALU instruction as discussed above. Making the load dependent upon the producer causes *store bypassing* to occur.

In case of data reuse through loads, the performance is improved because more memory operations will proceed in parallel than the number of available cache/memory ports. In case of data reuse through stores, the effective critical path is reduced which can greatly improve the performance. The scheme performs poorly only when the addresses of store instructions become available at the same time or later than their data values. In this case, the memory operations will be serialized and all the benefits of reuse will be lost (this happened in two of our benchmarks – `099.go` and `130.li`).

Handling Stores.
When a store instruction is encountered again first MDRD accesses the VAAS. In case of a miss, we enter the store address into the VAAS and proceed to the regular DCACHE access stage. In case of a hit, if both the value to be stored by this store is available, and the register at the hit location has its value, the two values are compared. If the values are equal, the store is *silent* and therefore it is squashed after updating the address corresponding to the store's data value register. If the values are different, or if the store value is not available, all matching entries are invalidated. In other words, when the data value is not available there can be only one entry with a given address. When there are multiple entries with the same address all the values are equal and all of them are available. Therefore, VAAS keeps the *in-order dependency status of load and store instructions* and as such acts both as a disambiguator and a reuse detector.

3.5 Hardware Complexity of VAAS

Since the physical register file is always present, the hardware cost of our technique is the address CAM which has the same number of entries as the register file. While the register file is an indexed structure, address CAM is more complex due to its ability to simultaneously search all entries for an address. Depending upon the number of instructions which are allowed simultaneous access to VAAS, appropriate number of read and write ports should also be present. However, we must remember that we do not require a separate forwarding buffer since the values are now supplied by the VAAS. Forwarding buffer requires an organization similar to VAAS because in both cases a parallel search for the address in question is needed and if multiple loads can be issued in each cycle, multiple read ports to the forwarding buffer should also be provided. A forwarding buffer may have fewer entries than the address CAM in VAAS, but then each entry is larger since it must contain both the value and the address. Therefore, the hardware costs of the address CAM is expected to be similar to that of a forwarding buffer. While we need additional ports to read values from the physical register file, we need fewer ports to memory since the values found in physical registers will not have to be fetched from memory. We also avoid the need for other multiple hardware structures needed to implement existing algorithms for the various optimizations discussed earlier.

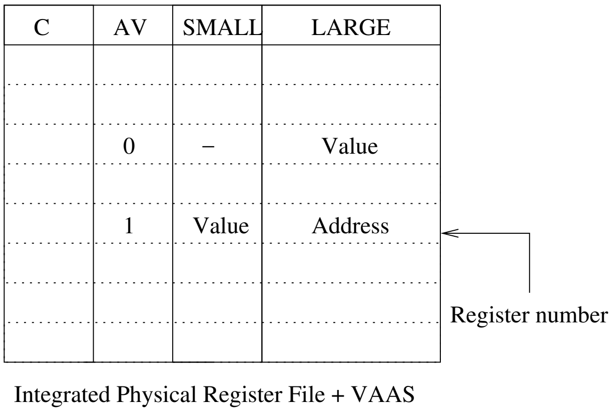

Figure 4: Restricting optimizations to small values.

Very often the integer data values used by programs are small values which can be stored in fewer than 32 bits. This is not only true for multimedia applications, but it also holds true for more general purpose applications such as the `SPECint95` benchmarks [22]. If we restrict our implementation so that it only exploits optimization opportunities that involve small data values, we can lower the cost of the VAAS as follows. We can use the structure shown in Figure 4 where instead of having two large fields of 32 bits for the value and address we provide two fields, SMALL (less than 32 bits) and LARGE (32 bits). If we have an address-value pair where the value is small, we can save the value in the SMALL field and the address in the LARGE field. If the value is large, it is stored in the LARGE field and the address is not stored thus sacrificing optimization opportunities. The AV field indicates whether an entry contains an address-value pair or simply a value. The search for an address hit is carried out over the LARGE field and if a hit occurs, the AV field is used to determine if this is a valid hit.

4. EXPERIMENTAL RESULTS

In this section we report on the experimental results based upon the implementations of the VAAS based superscalar design presented in the preceding section. All optimizations, except store bypassing, were implemented in absence of load speculation. Our simulated architecture is based upon the MIPS-I instruction set. The simulators we use have been generated using the FAST system [16]. The benchmarks we used are from the `SPEC95` suite which were executed on the test inputs. They were compiled using `gcc 2.7.2` compiler with the `-O3` level of optimization. Our simulated architecture uses the following configuration:

Number of rename registers	128
Instruction window size	64
Reorder buffer size	64
Issue width	8 instructions
Retire width	16 instructions
Functional units	8 symmetric units
Memory ports	2 read/write
VAAS ports	4 read/write
Instruction cache	ideal
Data cache	ideal

Silent stores. In Figures 5 and 6 we show the degree of value reuse opportunities in stores found in `SPEC95` benchmarks as well as the value reuse opportunities captured and exploited by VAAS. We found that on an average of 53.9% and 41.7% of total stores to be value redundant in `SPECint95` and `SPECfp95` benchmarks. On an average VAAS captured and avoided writes associated with 25% of total stores in `SPECint95` and 22.6% of stores in `SPECfp95` benchmarks. We believe that VAAS based implementation is very effective as with limited hardware cost it is able to eliminate substantial number of writes to memory. This optimization is achieved without increasing the number of memory reads. In contrast the technique proposed by Lepak and Lipasti [8] requires a load for every store (redundant or not) to detect value redundant stores; thus, greatly increasing the number of memory reads.

Loads avoided. Recall that the memory read due to a load can be avoided by forwarding the value it requires from a prior load or due to an earlier store. The reuse opportunities captured by VAAS in each of these categories is shown separately in Figures 7 and 8. On an average VAAS found slightly more than 60% of loads in `SPECint95` and 58% of loads in `SPECfp95` to be redundant. This number is much greater than the number reported in [18] because Sodani and Sohi only captured self-reuse of loads. Yang and Gupta [21] consider all types of redundancies and on an average found 43% of loads to be redundant in `SPECint95` benchmarks. In contrast, in this paper we are able to capture a greater amount of load reuse (60% vs 43% for SPECint95) which we attribute to the conceptual simplicity of the VAAS structure.

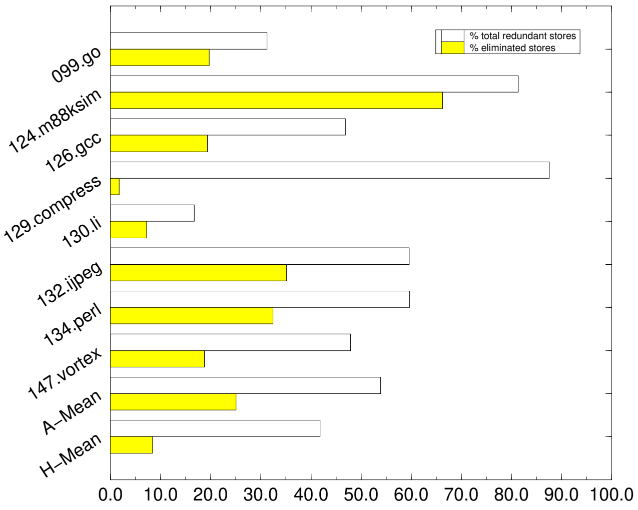

Figure 5: SPECint95 - Silent stores: total present vs detected by VAAS.

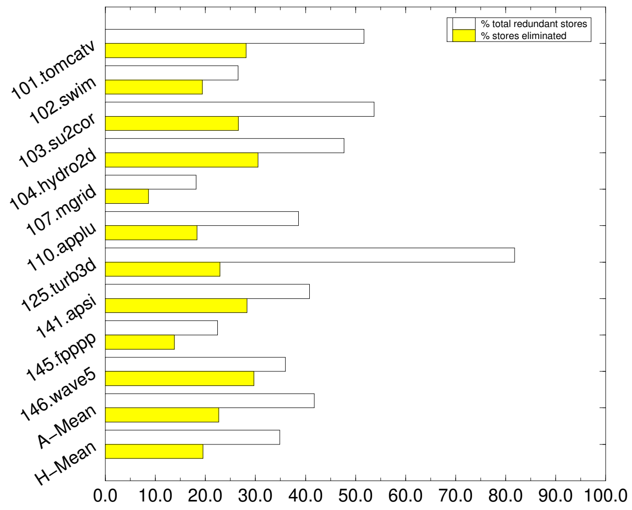

Figure 6: SPECfp95 - Silent stores: total present vs detected by VAAS.

Potential of exploiting small values. We also categorized the values involved in redundant loads and stores according to their size. The results of this experiment are shown in Figures 9 and 10. This experiment yielded interesting results. Nearly 49% of the values involved were very small requiring 1 byte and over 45% were very large requiring all 4 bytes. Only few percent of the values needed 2 or 3 bytes. We believe this is indicative of the fact that programs make very frequent use of small constants (e.g., 0, 1, -1 etc.) and frequently compute addresses (e.g., due to arrays and pointers) [22]. The values in the former category require 1 byte and those in the latter category typically require all 4 bytes. Therefore, if we would like to reduce the combined size of the VAAS and the physical register file as suggested in section 3, we should have a LARGE field of size 4 bytes and SMALL field of size 1 byte. This approach will reduce the degree of reuse captured to nearly half.

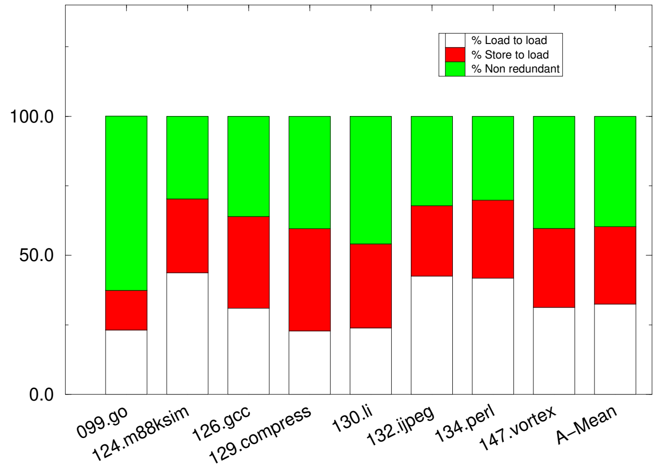

Figure 7: SPECint95 - Load-to-load forwarding: store-to-load forwarding: non-redundant loads

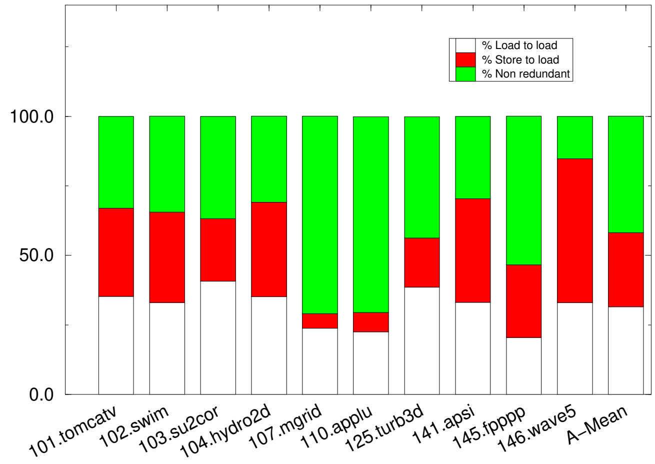

Figure 8: SPECfp95 - Load-to-load forwarding: store-to-load forwarding: non-redundant loads

Memory reference reduction. Given the high degree of load and store reuse detected by VAAS in the SPEC95 benchmark suite, it is not surprising that the reduction in total memory reference is quite substantial. As shown in Figures 11 and 12, on an average the reduction in memory reference is 45% (47%) for `SPECint95` (`SPECfp95`) benchmarks. The reductions attributed to load reuse are shown separately from the reductions resulting due to store reuse. It is quite clear that both load and store reuse play an important role in reducing memory references.

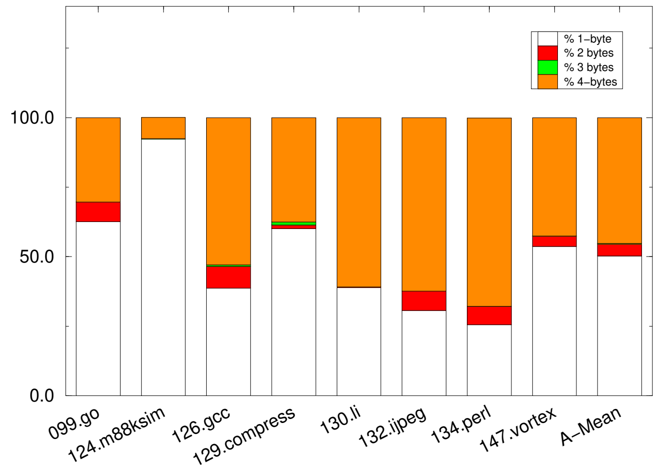

Figure 9: SPECint95 - Reuse involving values of size: 1-byte; 2-bytes; 3-bytes; and 4-bytes.

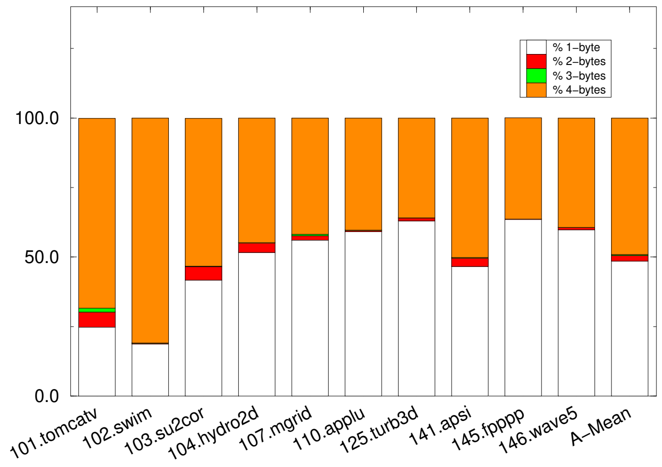

Figure 10: SPECfp95 - Reuse involving values of size: 1-byte; 2-bytes; 3-bytes; and 4-bytes.

Instructions completed per cycle. Finally we compared the performance of a superscalar pipeline that incorporates VAAS with a baseline superscalar machine that does not contain VAAS. In this experiment we choose a baseline machine which has a 1 cycle memory access delay with a perfect data cache. In contrast the VAAS based superscalar machine takes 2 cycles for each memory access. In other words (a) our machine always take 2 cycles to access memory while the baseline superscalar always takes 1 cycle; and (b) even though we reduce memory traffic by over 45%, we assume that we do not have fewer data cache misses than the baseline superscalar.

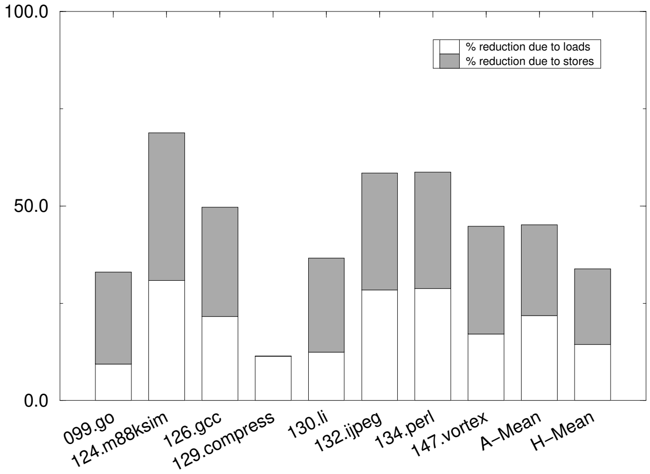

Figure 11: SPECint95 - Memory reference reduction due to redundant loads and stores.

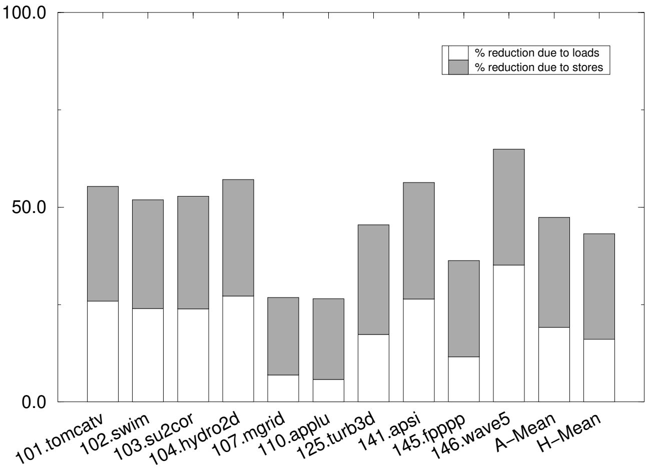

Figure 12: SPECfp95 - Memory reference reduction due to redundant loads and stores.

Under these conditions, as we can see from Figures 13 and 14, we achieve substantial speedups for most benchmarks. In case of `SPECfp95` the speedups range from 5% to 33%. In case of `SPECint95` the speedups for 6 benchmarks range from 2% to 16%. The speedups are due to increased throughput of memory operations because in the VAAS based design the memory operations are split nearly equally among the physical register file and the memory and therefore greater number of them can be handled per cycle. Moreover under some conditions the forwarding of values to a load (from another store or load) by VAAS also speedups the execution of load operations.

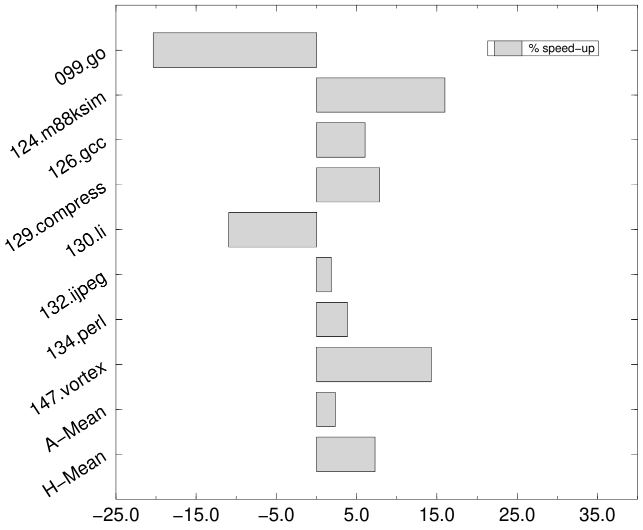

Figure 13: SPECint95 - IPC values.

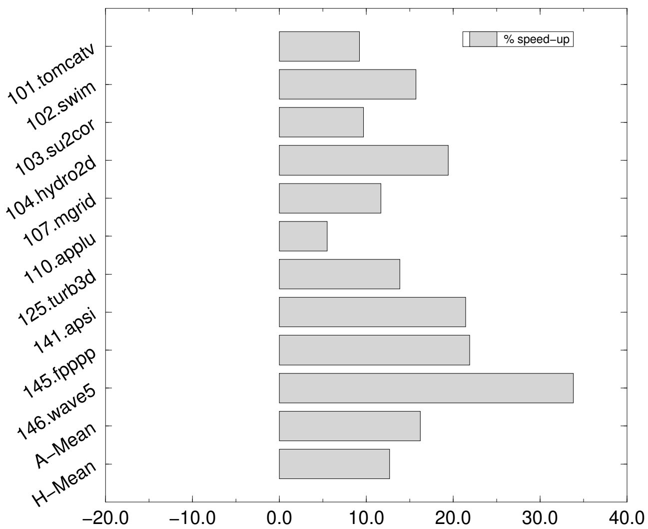

Figure 14: SPECfp95 - IPC values.

There are two integer benchmarks, `099.go` and `130.li`, which show slow downs under the above conditions. Our assessment of this behavior is that the serialization of the address computations in these benchmarks causes the throughput of memory operations to be low and load/store-to-load forwarding to be ineffective. On the other hand we continue to pay the higher price of 2 cycle delay for memory accesses.

Reduction of switching in L1 cache. Finally we measured the reductions in switching activity that can be achieved by storing values in original or complemented form. Since we assumed an ideal data cache, this measurement represents the upperbound on switching reduction. The results are given in Figure 15. On an average we observed 6.59% reduction in switching for `SPECint95` and 8.67% reduction for `SPECfp95` benchmarks. These reductions were derived by applying our optimization to values written by 24.69% of stores in `SPECint95` and 32.04% of stores in `SPECfp95` benchmarks.

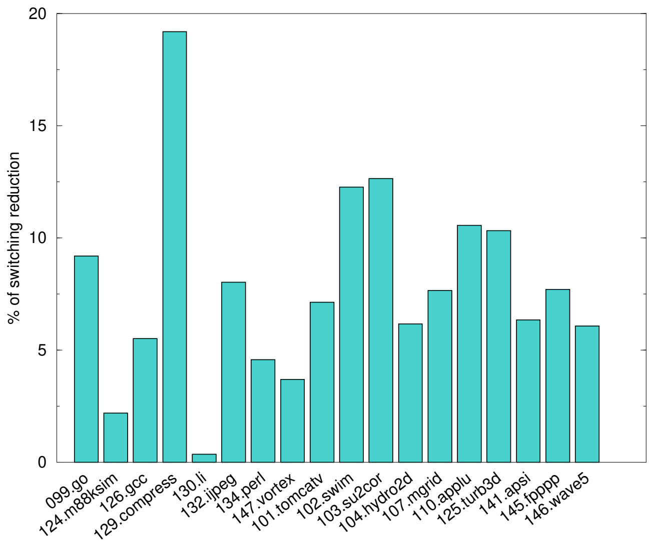

Figure 15: SPEC95 - reduction in switching.

5. LOAD SPECULATION AND MEMORY ACCESS ORDER VIOLATION

In this section we consider an out-of-order processor which also incorporates load speculation. We demonstrate that our approach achieves the following two goals. First we demonstrate that we can efficiently exploit value redundancy to cut down on false memory access order violations. Second we show that all of the optimizations discussed in absence of load speculation can also be performed in presence of load speculation.

We assume that the processor incorporates a mechanism that is responsible for making load speculation decisions and detecting memory access order violations. From the perspective of MDRD, when load speculation is performed, a load is allowed to proceed ahead and access VAAS before its potential conflict with prior stores is resolved.

Suppressing memory access order violations. A certain subset of memory access order violations which would be detected and reported by a mechanism that is simply based upon comparing a load address with addresses of stores above which the load has been speculated can be safely suppressed. In particular, value redundancy can be used to suppress the reporting of these memory access order

PC2:	St R2, Addr2
	......
PC3:	Ld R3, Addr3

Value-Address Optimization	Enabling Condition	Existing Technique
Speculating Ld R3, Addr3 wrt St R2, Addr2 does not cause a memory order violation.	Addr2 $\neq$ Addr3 *OR* (Addr2 = Addr3 AND Value(R2) = Value(R3))	In [17] we achieve this by using an *expanded speculative loads table* and *forwarding buffer.*

Figure 16: Optimization of memory access order violation detection.

violations. The example in Figure 16 illustrates this optimization. Let us assume that the load at PC3 is speculated above the store at PC2. If the addresses of the two are the same, an address based scheme will report a misspeculation. However, if the loaded value is the same as the stored value, then this misspeculation can be safely suppressed.

In [17] we had presented a solution for incorporating the above optimization in a store set [2] based memory disambiguator. In this solution a *speculated loads table* is maintained which contains an entry for each speculated load. The entry identifies the load, contains the load address, speculatively loaded value, and a sequence number which captures the original ordering of the memory operations. The sequence numbers enable identification of the loads in the speculated loads table which have been speculated above a given store. If address of the store matches one of these loads we have a potential for a load misspeculation. If the loaded value matches the value being stored then the misspeculation is suppressed; otherwise it is reported. The value field added to the *speculated loads table* enables the optimization.

In a VAAS based superscalar we can perform the above optimization without a separate speculated loads table. For this purpose we need to extend each VAAS entry by adding two fields. The first field is the AGE field which is used to save the sequence numbers. In addition, we need to know whether a value in a register was placed by a load instruction or a store instruction. Therefore we maintain the L/S field. The savings in a VAAS based implementation result because the *loaded value* field is no longer needed as we can now rely upon the physical register file to supply the value. The modified VAAS structure is shown in Figure 17. When a store address is known, the loads in VAAS with higher sequence numbers are identified. Their addresses and values are compared with the store's address and value to identify misspeculated loads.

Other optimizations. Now let us consider the optimizations presented in section 2 to see how they are carried out in presence of load speculation.

Load-to-load forwarding. The load-to-load forwarding proceeds in much the same way as before. However, there is one important difference. The load that supplies the value may now be a speculated load in which case the value that it supplies to a later load may be incorrect. When misspeculation is detected recovery must be accomplished. One approach for this recovery is to *restart the execution* from the misspeculated load. In this case the later load which was the forwarded an incorrect value is discarded too and executed again and therefore the recovery functions correctly. However, if the recovery process is optimized by only executing those instructions which were directly or indirectly affected by the misspeculated value, a more complex mechanism is needed to identify the subset of instructions that must be executed again. Typically this is achieved by linking all the instructions through which a speculated value is propagated. Since now load-to-load forwarding may be performed speculatively, the involved loads must also be linked together to correctly carry out the recovery process.

Store-to-load forwarding. As in the case of load-to-load forwarding, we can perform store-to-load forwarding without checking to see if the load which is the target of forwarded value is being executed speculatively or non-speculatively. Consider the example shown below in which there are two stores followed by a load. Let us assume that the load is speculated above the second store. If Addr0 and Addr1 are equal, store-to-load forwarding will be carried from the first store to the load. If later it is found that Addr2 is not the same as Addr0/Addr1, then load speculation is successful and therefore forwarding is also successful. On the other hand if Addr2 is the same as Addr0/Addr1, then a potential for misspeculation exists. The forwarded value is compared with the value stored by the second store to determine whether the misspeculation should be suppressed or reported. Either case is routinely handled by the memory access order violation detection mechanism. Therefore store-to-load forwarding does not require any special consideration in presence of load speculation.

PC0:	St R0, Addr0
	......
PC1:	St R1, Addr1
	......
PC2:	Ld R2, Addr2

Silent stores. Load speculation has an interesting implication for detection of silent stores. All silent stores that are suppressed in absence of load speculation continue to be detected and suppressed in presence of load speculation. In addition, some additional silent stores may be detected in presence of load speculation. Consider the example given below where load follows a store.

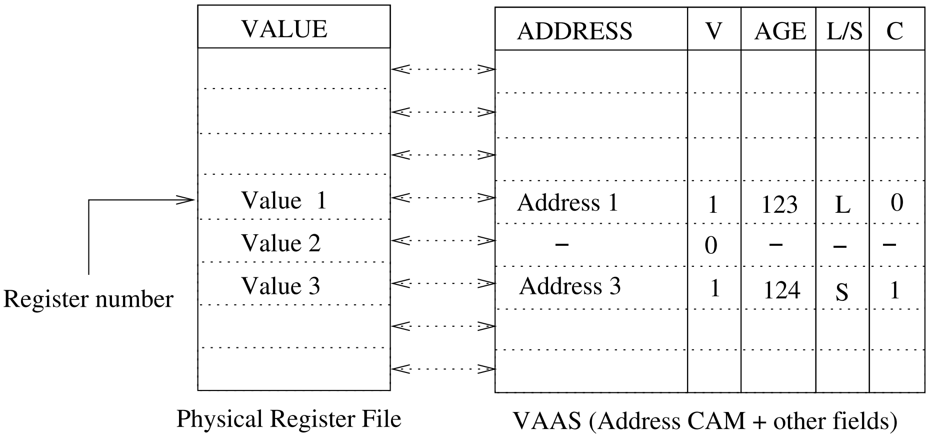

Figure 17: Modified VAAS structure.

```
PC1:   St R1, Addr1
       ......
PC2:   Ld R2, Addr2
```

Let us assume that the load is speculated above the store. If Addr1 and Addr2 are not the same, the memory access order violation will conclude that load speculation was successful. On the other hand if Addr1 and Addr2 are the same two cases arise. If the speculatively loaded value is different from the stored value, misspeculation occurs. On the other hand if the values are the same, no misspeculation is reported. Moreover in this situation it is also the case that the store must be a silent store. Therefore speculative execution of the load helps us ascertain that the store is a silent store. However, such a determination may not have been made if load speculation had not been performed.

Experimental evaluation. We have performed experiments to determine the extent to which value reuse can reduce the number of mispredictions resulting from load speculation. The results are shown in Figures 18 and 19. As we can see, for an 8-issue processor, the reductions in mispredictions are substantial (12-24%). As the issue width is increased to 16-issue, the degree of speculation performed by the processor increases and therefore the reductions in mispredictions achieved by exploiting value redundancy are even greater.

Comparison with speculative value reuse techniques. There are two pieces of research which share an important characteristic with our work. The first is the work by Jourdan et al. [6] which exploits value reuse opportunities detected through examination of register file contents in carrying out value forwarding which is predictive in nature. The second is the work by Tullsen and Seng [19] on storageless load value prediction. Both of the works exploit value reuse opportunities using register file contents. Both of the above techniques for reuse are speculative techniques which do not generate memory traffic to access values needed to verify that speculation was correctly performed. However, we present a simple unified solution for many optimization tasks which require similar types of information.

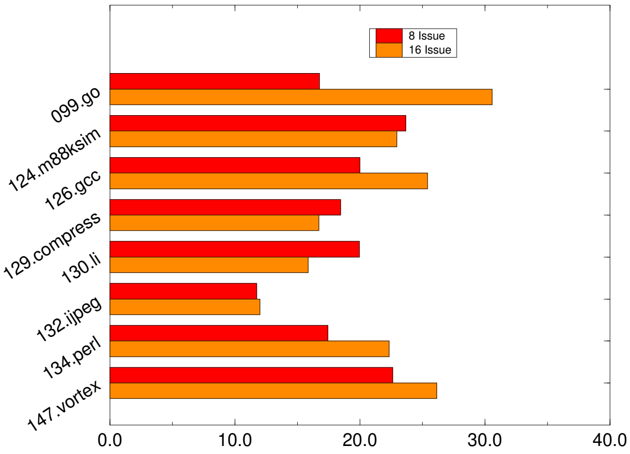

Figure 18: False mispredictions prevented in SPECint95.

6. OTHER RELATED WORK

In sections 2 and 5 we have already compared our work with closely related non-speculative and speculative value reuse techniques involving memory operations. In this section we compare our work with other approaches for optimizing memory operations.

Speculative register promotion. There are two other proposals for exploiting value-address association structure like the one we have proposed [3, 4]. However, in these works this structure is allocated and hence exploited under compiler control. This has two implications. First the compile-time decisions limit the amount of dynamic value reuse that

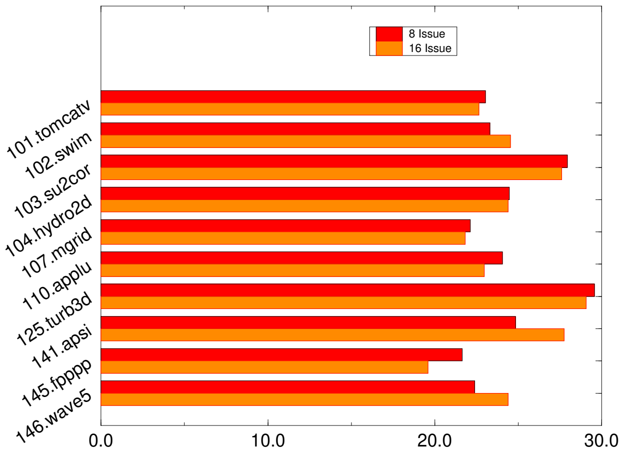

Figure 19: False mispredictions prevented in SPECfp95.

can be performed. Second the structure associates addresses with architectural registers visible to the compiler and thus value reuse that arises through physical registers is not exploited. Thus, our approach can perform value reuse to a greater extent than these techniques.

Filter cache. The load-store reuse mechanism can be viewed as a form of a filter cache [7] since it is also a small structure which sits in between the CPU and the L1 cache and reduces the number of references that are sent to the L1 cache. However, there is an important difference. The conventional filter caches studied in [7] achieve energy reduction at the cost of increased execution times. However, the technique we have presented typically resulted in significant reductions in execution time. In a related piece of work Yang and Gupta have developed an energy efficient implementation of the load and store reuse optimizations in presence of load speculation [20]. While in this paper our emphasis has been on exploiting values in register file contents, [20] focuses on energy reduction by replacing the CAM structure by a direct mapped structure. Therefore, unlike the conventional filter cache, the technique presented in [20] results in reductions in both energy consumption and execution times.

Compile-time load reuse. Work has also been done on aggressively optimizing load operations at compile time. In [1] a load reuse analysis was presented and evaluated. However, as mentioned in [1], even though very high levels of load reuse can be detected at compile time, exploitation of load reuse opportunities is difficult to perform because it requires a large number of architectural registers.

7. CONCLUDING REMARKS

In this paper we have presented the design of VAAS which enables unified implementation of several optimizations related to memory operations. We rely upon the physical register file to provide data values corresponding to a subset of memory addresses whose values are currently resident in physical registers. The design of VAAS enables us to manage the contents of physical register file as another level in the memory hierarchy. This unified implementation of several optimizations leads to a significantly simpler and less expensive hardware design than one that uses exiting implementations of the same optimizations. Our experiments indicate that our implementation of non-speculative optimizations is highly effective as it eliminates memory references due to 60% (58%) of loads in `SPECint95` (`SPECfp95`) and 25% (22.6%) of stores in `SPECint95` (`SPECfp95`). On an average over 45% of memory references are eliminated due to reuse optimizations. We have demonstrated that a VAAS based superscalar yields substantial speedups over the baseline superscalar which has one less stage.

Acknowledgements

This work is supported by DARPA award no. F29601-00-1-0183 and National Science Foundation grants CCR-0105355, CCR-0096122, EIA-9806525, and CCR-9996362. The equipment obtained through the research infrastructure grant EIA-0080123 to the University of Arizona was also used in this work.

8. REFERENCES

[1] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa, "Load-reuse analysis: design and evaluation," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 64-76, Atlanta, Georgia, May 1999.

[2] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. *ACM 25th International Symposium on Computer Architecture* (ISCA), pages 142–153, Barcelona, Spain, June 1998.

[3] Hank Dietz and Chi-Hung Chi. A new kind of memory for referencing arrays and pointers. *Supercomputing'88*, pages 360–367, Orlando, Florida, November 1988.

[4] Matthew Postiff, David Greene and Trevor Mudge. The store-load address table and speculative register promotion. *IEEE/ACM International Symposium on Microarchitecture* (MICRO), pages 235–244, Monterey, California, December 2000.

[5] Freddy Gabbay and Avi Mendelson. Using value prediction to increase the power of speculative execution hardware. *ACM Transactions on Computer Systems*, 16(3):234–270, August 1998.

[6] Stephan Jourdan, Ronny Ronen, Michael Bekerman, Bishara Shomar, and Adi Yoaz. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. *IEEE/ACM 31st Annual International Symposium on Microarchitecture* (MICRO), pages 216–225, December 1998.

[7] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. Filter cache: an energy efficient memory structure. *IEEE/ACM 30st Annual International Symposium on Microarchitecture* (MICRO), pages 184–193, December 1997.

[8] Kevin Lepak and Mikko H. Lipasti. On the value locality of store instructions. *ACM 27th Annual International Symposium on Computer Architecture* (ISCA), pages 182–191, Vancouver, Canada, June 2000.

[9] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. *ACM 7th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pages 138–147, Cambridge, USA, October 1996.

[10] Srilatha Manne, Artur Klauser, and Dirk Grunwald, "Pipeline gating: speculation control for energy reduction," *ACM 25th Annual International Symposium on Computer Architecture* (ISCA), pages 132–141, June 1998.

[11] Teresa Monreal, Antonio Gonzlez, Mateo Valero, Jos Gonzlez, and Victor Vinals. Delaying physical register allocation through virtual-physical registers. *IEEE/ACM 32nd Annual International Symposium on Microarchitecture* (MICRO), pages 186–192, Haifa, Israel, November 1999.

[12] Andreas Moshovos and Gurindar S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. *IEEE/ACM 30th Annual International Symposium on Microarchitecture* (MICRO), pages 235–245, December 1997.

[13] Andreas Moshovos and Gurindar S. Sohi. Read-after-read memory dependence prediction. *IEEE/ACM 31st Annual International Symposium on Microarchitecture* (MICRO), pages 177–185, November 1999.

[14] Andreas I. Moshovos. *Memory Dependence Prediction*. PhD thesis, University of Wisconsin - Madison, 1998.

[15] Andreas I. Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. *ACM 24th International Symposium on Computer Architecture* (ISCA), pages 181–193, June 1997.

[16] Soner Önder and Rajiv Gupta. Automatic generation of microarchitecture simulators. *IEEE International Conference on Computer Languages*, pages 80–89, Chicago, May 1998.

[17] Soner Önder and Rajiv Gupta. Dynamic memory disambiguation in the presence of out-of-order store issuing. *IEEE/ACM 32nd Annual International Symposium on Microarchitecture* (MICRO), pages 170–176, November 1999.

[18] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. *ACM 24th International Symposium on Computer Architecture* (ISCA), pages 194–205, 1997.

[19] Dean M. Tullsen and John S. Seng. Storageless value prediction using prior register values. *ACM 26th International Symposium on Computer Architecture* (ISCA), pages 270–279, May 1999.

[20] Jun Yang and Rajiv Gupta. Energy-efficient load and store reuse. *ACM/IEEE International Symposium on Low Power Electronics and Design*, Huntington, CA, August 2001.

[21] Jun Yang and Rajiv Gupta. Load redundancy removal through instruction reuse. *International Conference on Parallel Processing*, pages 61–68, Toronto, Canada, August 2000.

[22] Youtao Zhang, Jun Yang, and Rajiv Gupta. Frequent value locality and value-centric data cache design. *ACM 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pages 150–159, Cambridge, MA, November 2000.