ARB: A Hardware Mechanism for Dynamic Reordering of Memory References*

Manoj Franklin

Department of Electrical and Computer Engineering
Clemson University
221-C Riggs Hall
Clemson, SC 29634-0915, USA
mfrankl@blessing.eng.clemson.edu

Gurindar S. Sohi

Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, WI 53706, USA
sohi@cs.wisc.edu

* This work was supported by National Science Foundation grants CCR-8919635 and CCR-9410706 and by an IBM Graduate Fellowship.

Abstract

To exploit instruction level parallelism, it is important not only to execute multiple memory references per cycle, but also to reorder memory references, especially to execute loads before stores that precede them in the sequential instruction stream. To guarantee correctness of execution in such situations, memory reference addresses have to be disambiguated. This paper presents a novel hardware mechanism, called an *Address Resolution Buffer (ARB)*, for performing dynamic reordering of memory references. The ARB supports the following features: (i) dynamic memory disambiguation in a decentralized manner, (ii) multiple memory references per cycle, (iii) out-of-order execution of memory references, (iv) unresolved loads and stores, (v) speculative loads and stores, and (vi) memory renaming. The paper presents the results of a simulation study that we conducted to verify the efficacy of the ARB for a superscalar processor. The paper also shows the ARB's application in a multiscalar processor.

1. INTRODUCTION

Instruction-level parallel (ILP) processors boost performance by forming an instruction execution schedule, either statically or dynamically, in which the instructions are executed in an order that is different from the order in which they occur in the (sequential) program. This is called instruction reordering. Because memory referencing instructions account for a large fraction of the instructions in most programs, the ability to reorder memory referencing instructions is very important. A load instruction can be reordered to execute before another load instruction that precedes it in the program order without violating any memory dependencies. However, if a load is reordered to execute before a preceding store, a store reordered to execute before a preceding store, or a store reordered to execute before a preceding load, then read-after-write, write-after-write, or write-after-read dependencies, respectively, could be violated if the two instructions access the same memory location. The instruction scheduler must ensure that the reordering does not violate dependencies; this is done by determining if the reordered pair of references access the same memory location. The process of determining if two memory referencing instructions access the same memory location is called *memory disambiguation* or *memory antialiasing* [4], and is a fundamental step in any scheme to reorder memory operations.

1.1. Need for Good Dynamic Disambiguation

Developing an execution schedule, and therefore reordering of memory references, can be done statically by the compiler or dynamically by the hardware. Memory disambiguation can also be performed statically, dynamically, or at both times, in an orthogonal manner. Static disambiguation techniques, however, have limitations which makes dynamic disambiguation attractive, either to complement static disambiguation or to work all by itself [7, 11, 14]. In statically scheduled processors, dynamic disambiguation is used to complement static disambiguation, and involves disambiguating only those ambiguous references that have been reordered at compile time. By contrast, in dynamically scheduled processors, dynamic disambiguation is used to disambiguate all loads and stores in the active instruction window, that is, the window of instructions being considered for scheduling.

As ILP processors become more aggressive, the size of the active instruction window, and consequently the number of memory operations in the instruction window, becomes larger. This implies that a larger number of memory operations have to be considered in the disambiguation process. Moreover, multiple memory references might have to be executed in a given cycle, calling for multiple disambiguations in a cycle. This double-barreled impact of more aggressive ILP exploitation calls for disambiguation mechanisms that can perform both functions effectively and efficiently.

1.2. Support for Speculative Execution

To increase the opportunities for parallelism extraction, and the consequent reordering of code, ILP processors use *speculative execution*. A path of execution is predicted to be taken, and instructions from this predicted path are executed in a speculative manner. The execution is speculative because there is no assurance that these instructions have to be executed.

With speculative execution, the lifetime of an instruction can be divided into four distinct phases: *issue*, *execute*, *complete*, and *commit* (or *retire*). In the issue phase, an instruction is decoded and decisions are made as to how this instruction is to be handled. At some point after issue, an instruction enters its execute phase and the operation specified by the instruction is initiated on the specified operands. Once the operation is finished, and the result is available, the instruction completes execution. These three phases happen regardless of whether the instruction is executed speculatively or not. For speculatively executed instructions, however, there is an additional phase, the commit phase. When it is known that an instruction that was executed speculatively was indeed meant to be executed, its effects can be committed, and the state of the machine updated [8, 9, 15, 17].

With speculative execution, memory operations need special treatment. A store operation can be allowed to proceed to the memory system only when it is guaranteed to commit, otherwise the old memory value will be lost, complicating the recovery procedures in case of an incorrect speculation. Nevertheless, succeeding loads (from speculatively executed code) to the same memory location require the new uncommitted value, and not the old value. Thus, the memory operation reordering mechanism has to provide some means of forwarding uncommitted memory values to subsequent loads, and values have to be written to memory locations in the order given by the sequential semantics of the program.

1.3. Support for Dynamically Unresolved Memory References

Another issue that needs to be addressed when reordering memory operations is that of *dynamically unresolved memory references*. Ordinarily, before a memory operation can be executed, the disambiguation mechanism needs to check the addresses of all preceding memory operations in the active window to see if there is a conflict. This check is not possible if the addresses of preceding memory operations are not known. Thus, a load operation waits until the addresses of all preceding stores are known, and a store operation waits until the addresses of all preceding loads and stores are known, even though the load (or store) may be ready to execute. For example, in a program in which a linked structure is being traversed, the addresses of the memory references to the linked structure are not known until the links have been traversed. This late resolution of addresses detains all later memory operations, including those to other data structures, even though those operations are ready to execute. Detaining memory operations could also prevent non-memory operations from entering the active window and being considered for execution, further limiting the opportunities for ILP extraction.

We feel that a good memory reordering mechanism should overcome this restriction by permitting memory references to be executed before proper address disambiguation can be carried out, as *dynamically unresolved memory references*. That is, the reordering mechanism should allow the execution of loads before disambiguating them against preceding stores, or even before the addresses of the preceding stores are known[1]. Similarly, it should allow the execution of stores before disambiguating them against preceding loads and stores. (Note that compilers for statically scheduled processors that rely on run-time disambiguation allow *statically unresolved memory references* by reordering ambiguous memory references [7, 11, 14].)

1.4. Paper Objective and Organization

The objective of this paper is to propose a new hardware mechanism for supporting memory operation reordering in an aggressive ILP processor. The proposed mechanism, called an *Address Resolution Buffer (ARB)*, is very general, and is applicable to different execution models. Our emphasis in this paper, however, is on dynamically scheduled ILP processors. The ARB assists the dynamic memory operation reordering process by supporting the following features:

(1) dynamic memory disambiguation in a decentralized manner,

(2) multiple memory references per cycle,

(3) out-of-order execution of loads and stores with respect to both loads and stores,

(4) dynamically unresolved loads and stores,

(5) speculative loads and stores, and

(6) memory renaming.

The rest of this paper is organized as follows. Section 2 discusses the background and previous work. Section 3 describes the ARB and its working for dynamically scheduled processors such as a superscalar processor. Section 4 describes extensions for handling variable data sizes, and extensions (two-level hierarchical ARB) for increasing the number of references the ARB can keep track of at any time. Section 5 presents the results of a simulation study that evaluates the performance of the ARB in a superscalar processor. Section 6 describes application of the two-level hierarchical ARB in the multiscalar processor [6, 18], the erstwhile Expandable Split Window (ESW) processor [5]. Section 7 provides a summary and draws the conclusions of this research.

2. BACKGROUND AND PREVIOUS WORK

The first step in the process of dynamically reordering memory operations is the disambiguation step. Techniques for dynamic disambiguation use the following basic principle. Memory operations are arranged based on the order in which they are dynamically encountered, by giving each operation a dynamic sequence number. (These sequence numbers are assigned by the Instruction Decode Unit or some other stage in the instruction pipeline, with the help of a counter that keeps track of the current sequence number. The counter is incremented each time a memory reference is encountered, and it wraps around when the count becomes n,

[1] If and when an unresolved load is detected to have fetched an incorrect value, the processor must recover from the incorrect execution. This can be done by using the recovery facility already provided for speculative execution of code.

where n is the maximum number of memory reference instructions allowed to be present in the active instruction window at any one time.) To reorder memory operations without violating dependencies, a memory operation should not be scheduled to execute before a conflicting store that precedes the memory operation (as indicated by its sequence number). The two parameters for the disambiguation process are therefore the address and the sequence number of the references.

Existing methods implement the above principle as follows. All memory references are arranged in a queue, or some other hardware structure that functions like a queue, in the order of their sequence numbers. When a memory operation is to be scheduled for execution, this structure is searched to see if there is a conflicting operation with an earlier sequence number to the same address. This search is typically implemented by comparing the address of the memory operation to be scheduled against the addresses of all previous memory operations in the structure. That is, the dynamic disambiguation mechanism does an associative search of the addresses of all earlier, still active, memory references. This associative search can be quite complex if the number of entries to be searched is quite large. Moreover, if multiple memory references are to be executed in a clock cycle, multiple such searches need to be done in a cycle. As ILP processors become more aggressive, both the number of entries to be searched (corresponding to the number of active memory references), as well as the number of searches per cycle (corresponding to the number of memory operations to be executed per cycle) increase.

The *store queue* of the IBM 360/91 and its variants [1, 3, 12, 16] are examples of the above basic implementation. A tacit assumption in these techniques is that the ability to reorder store instructions to execute before preceding loads is not important[2]. If stores are not reordered to execute before preceding loads, or loads are always executed before succeeding stores, then the disambiguation hardware structure only needs to keep track of store instructions to ensure that dependencies are not violated by the dynamic scheduling.

In the store queue method, the addresses of pending store instructions are kept in a queue until the stores are ready to execute. When a load is considered for execution, its address is disambiguated by comparing it against the addresses of pending stores. The load is allowed to proceed if there is no conflict. (The compare can be done after fetching the data from memory too, as in the IBM System/370 Model 168.) If the load address matches a previous store address, the load is not issued to memory. Rather, it is serviced when the pending store to that address completes.

A few observations about the memory reordering capabilities of the basic store queue, as described above, are in order. First, whereas the store queue was originally described for machines that did not carry out speculative execution, incorporating speculative memory operations into the store queue is straightforward. All that has to be done is to commit memory operations from the store queue in program order, and enhance the abilities of the queue to pass a value from a complete, but not yet committed, store operation to later, pending, load operations.

Second, an address is not entered into the queue until it is known. That is, there is no support for dynamically unresolved references. A store operation whose address is unknown is stalled, likely in the issue stage of the processor. Stalling an operation typically implies that no operations, including non-memory operations, which succeed the stalled operation in program order, enter the window of execution, and therefore they can not be considered for execution. This reduces the amount of ILP that can be exploited.

The *dependency matrix* of HPS partially addresses the unresolved references problem [12]. Here, an unknown address store operation does not stall issue; rather it is made to step aside (into the memory-dependency handling mechanism), allowing succeeding instructions to enter the active instruction window. Memory dependencies are handled by two structures: a *memory write buffer*, which is very similar to the store queue described above, and a dependency matrix. Overall operation is as follows. If there are no preceding store operations with unknown addresses, memory operations proceed as they would with the store queue method. If there is a store operation with an unknown address, then succeeding memory operations are not allowed to proceed; the bits of the dependency matrix are used to determine when a memory operation can proceed. Memory operations are assigned a unique row in the dependency matrix; the rows are managed as a

[2]This assumption was then justified because the hardware instruction windows considered in those times were small, and in a typical code sequence that carries out an operation on a memory location, the load is executed first, followed by the computation instructions, followed by the store. That is, the load is ready to execute as soon as its address is known, which could be as soon as it is encountered. (It is an unresolved load if the register(s) needed to calculate the address of the load are busy when the load is encountered.) However, the store is not ready to execute (even though the store address may be known) until the computation instructions that produce the value for the store have completed.

circular queue. When an unknown address store is encountered, and it corresponds to row *i* of the dependency matrix, bits in column *i* are set to 1, When the address becomes known, the bits are cleared to 0. A memory operation, corresponding to row *j* of the dependency matrix, is allowed to proceed only when no preceding store operation has unknown address, and this fact is established by checking to see that there are no 1's in row *j* of the dependency matrix (actually only in parts of the row that correspond to preceding memory operations).

By allowing the dynamic ILP machine to move the instruction window past store instructions whose addresses are unknown, the dependency matrix opens up more opportunities for parallelism exploitation. However, the dependency matrix does not completely address the dynamically unresolved references problem that we outlined previously, where we stated the importance of allowing loads and stores (and instructions that depend upon them) to proceed, even though the address of an earlier store instruction is unknown, with the expectation that the addresses will not conflict.

To summarize, existing solutions for dynamic memory reordering have two drawbacks. First, they do not provide full speculative flexibility to the reordering process by not providing sufficient support for dynamically unresolved memory references. Second, and perhaps more important, they require (very) wide associative searches in the disambiguation step. This need for wide associativity restricts both the size of the instruction window from which ILP can be extracted, as well as the number of disambiguations that could be carried out in a cycle.

3. ADDRESS RESOLUTION BUFFER

3.1. Basic Idea

The reason why existing solutions for dynamic memory reordering require a wide associative search is that the hardware structure used to perform the disambiguation step orders references by time (or sequence number), and then searches for addresses in this temporal order. Our proposed Address Resolution Buffer (ARB) uses the complementary approach. Memory references are directed to bins based on their address, and the bins are used to enforce a temporal order amongst references to the same address. Unlike existing solutions, in the ARB the primary parameter used to initiate the disambiguation process is the address, and the secondary parameter, used to complete the disambiguation process, is the sequence number. As we shall see in the following sections, this has two important advantages. First, because different addresses map to different bins, by having several such bins, or ARB banks, the disambiguation process can be distributed (or decentralized), and multiple disambiguation requests can be dispatched in a single cycle. Second, because there are fewer references to handle in each bin, the associativity of the search is reduced. Furthermore, as we elaborate below, the basic ARB structure is easily enhanced to perform a variety of functions that are useful to the overall task of dynamic memory operation reordering, in addition to the basic disambiguation function.

3.2. ARB Hardware Structure

The ARB is divided into banks, and the banks are interleaved based on memory addresses. Each ARB bank has a few (for example 4 or 8) row entries, and each row entry has an *address* field for storing a memory address. The rest of a row entry is divided into *n stages*, numbered $\{0 \,..\, n-1\}$, corresponding to sequence numbers $\{0 \,..\, n-1\}$. The stages are logically configured as a circular queue (with a *head* pointer and a *tail* pointer) and correspond to the circular queue nature of a sliding (or continuous) instruction window. The active ARB stages, the ones from the head pointer to the tail pointer, together constitute the *active ARB window*, and correspond to the active instruction window. The active ARB stages can be thought of as ''progressing in sequential order''. Each stage has a *load bit*, a *store bit*, and a *value* field. The load bit of stage *i* is used for indicating if a load with sequence number *i* has been executed to the address in the row's address field, and the store bit is for indicating if a store with sequence number *i* has been executed to the address in the row's address field. The value field is used to record the value written by that store.

Figure 1 shows the block diagram of a 4-way interleaved 6-stage ARB. In this figure, the head and tail pointers point to stages 1 and 5 respectively, and the active ARB window comprises stages 1 through 4, corresponding to four memory references with sequence numbers 1 through 4. The oldest and youngest memory references in the active instruction window have sequence numbers 1 and 4 respectively, and have not been executed yet (this can be gathered from the absence of load marks and store marks in stages 1 and 4 of all ARB banks). (Note that here, the interpretation of ''oldest'' and ''youngest'' is based on sequential program semantics, and not on the order in which the references are executed.) A load with sequence number 2 has been executed to address 2000, and its ARB entry is in bank 0. Similarly, a store with sequence number 3 has been executed to address 2001 (ARB bank 1), and the store value is 10.

3.3. Execution of Loads

When the processor executes a load with sequence number *i*, the following actions are taken. First, the ARB bank is determined based on the load address. The address fields of the entries in this ARB bank are then checked[3] to see if the load address is already present in the ARB bank. If the address is present, then a check is made in its row entry to see if an earlier store has been executed to the same address from a preceding instruction in the active window. If so, the store with the closest sequence number is determined, and the value stored in its value field is forwarded to the load's destination register. If no preceding store has been executed or if the address is not present in the ARB bank, the load request is sent to the data cache (or main memory, if data cache is not present). If the address is not present, a free ARB row entry is also allotted to the new address; if no free entry is available, then special recovery action (c.f. Section 3.5) is taken. The load bit of stage *i* of the ARB entry is set to 1 to reflect the fact that the load with sequence number *i* has been executed to this address. When multiple memory requests need to access the same ARB bank in a cycle, priority is given to the one closest to the ARB head pointer.

Note that a load request does not proceed to the data cache if a preceding store has already been executed from the active instruction window. In order for the ARB to be not in the critical path in the case where a preceding store has not been executed, load requests can be sent to both the ARB and the data cache simultaneously; if a value is obtained from the ARB, the value obtained from the data cache can be discarded. The formal algorithm for executing a load is given below. Note that this algorithm gives the semantics for the correct working; an ARB implementation can change the order of the algorithm statements or do multiple statements in parallel so long as correctness is maintained.

```
Execute_Load(Addr, Sequence)
{
        Determine ARB Bank corresponding to Addr
        Associatively check address fields of Bank to find Row | Row's address field = Addr
        if  (Addr not present)
        {
                if  (Bank full)
                {
                        Freed_Up = Recovery_Bank_Full(Bank, Sequence)
                        if (Freed_Up == 0)
                                return(0)
                }
                Allocate new Row entry in Bank
                Enter Addr in address field of Row
                Send load request to data cache
        }
        else
        {
                Check in Row to determine closest preceding store executed to Addr
                if  (no preceding store found)
                        Send load request to data cache
                else
                        Forward the value from its value field to the processor
        }
        Set Row's load bit of stage Sequence to 1
        return(1);
}
```

Figure 2 gives the logic diagram for carrying out the execution of loads in an implementation of a 4-stage ARB. The figure shows the logic associated with a single ARB bank only. The load address is supplied to the bank, and once the appropriate ARB row entry is selected, there is only a 4-gate delay to send the value to the processor if a value is present in the ARB. (This delay can be reduced to 3 if the complementary values of the

---

[3] It deserves emphasis that this associative search is only within a single ARB bank and not within the entire ARB, and that the search is for a single key, analogous to the tag match function in a cache.

store marks are also kept in the ARB row entries.) Note that in this design, an ARB bank can service multiple loads in the same cycle, if they are all to the same address. Furthermore, a load request is sent to the ARB and the data cache simultaneously.

3.4. Execution of Stores

When the processor performs the execute phase of a store, the ARB bank is determined, and the bank is searched to see if the store address is already present in the ARB. If not, an ARB row entry is allotted to the new address. The store bit of stage *i* of the ARB row entry is set to 1 to reflect the fact that the store with sequence number *i* has been executed. The value to be stored is deposited in the row entry's value field for stage *i*. If the memory address was already present in the ARB bank, then a check is also performed in the active ARB window to see if any load has already been performed to the same address from a succeeding instruction, with no intervening stores in between. If so, the ARB informs the processor control unit to initiate recovery action so that all instructions (in the processor hardware window) including and beyond the closest incorrect load are squashed[4]. On its part, the ARB moves the tail pointer backwards so as to point to the sequence number of the closest incorrect load; furthermore, the load bit and store bit columns corresponding to the sequence numbers stepped over by the tail pointer are also cleared immediately. The formal algorithm for performing the execute phase of a store is given below.

```
Execute_Store(Addr, Value, Sequence)
{
        Determine ARB Bank corresponding to Addr
        Associatively check address fields of Bank to find Row | Row's address field = Addr
        if  (Addr not present)
        {
                if  (Bank full)
                {
                        Freed_Up = Recovery_Bank_Full(Bank, Sequence)
                        if (Freed_Up == 0)
                                return(0);
                }
                Allocate new Row entry in Bank
                Enter Addr in address field of Row
        }
        else
        {
                Check in Row to determine sequence number S_L of closest succeeding load
                                        executed to Addr, with no intervening stores
                if  (succeeding load with no intervening stores found)
                        Squash(S_L)
        }
        Enter Value in the value field of stage Sequence of Row
        Set Row's store bit of stage Sequence to 1
        return(1);
}
```

```
Squash(Sequence)
{
        for each ARB Bank do
        {
                for each Stage from Sequence to ARB Tail do
                        Clear all load marks and store marks in Stage
```

[4] This recovery is very similar to what happens when an incorrect branch prediction is detected by the processor.

```
        for each Row in Bank do
                {
                        if (there is no load mark or store mark in Row)
                                Clear Addr field of Row
                }
        }
        Set ARB Tail = Sequence
}
```

Figure 3 gives the logic diagram and the logic equations for generating the squash signals in a 4-stage ARB. This figure also shows only a single ARB bank. Once the appropriate ARB row entry is selected, only 2 gate delays (if the complementary values of the load marks are also stored in the ARB row entries) are required to generate the squash signals. Multiple stores can be performed to the same ARB bank, so long as they are all to the same address. Only one squash signal is generated from an ARB bank. If multiple squash signals are generated from different ARB banks, the one closest to the ARB head pointer is selected.

3.5. Reclaiming the ARB Row Entries

The last part of the lifetime of a memory reference is the commit phase. The processor commits memory references as per the sequence number ordering, as and when the references can be committed. When the processor commits a memory reference with sequence number *i*, the ARB bank is determined, and the row entry corresponding to that address is determined. If the committed reference is a store, then the corresponding store value in the ARB is sent to the data cache. For both loads and stores, the corresponding load mark or store mark in the ARB is erased. An ARB row is reclaimed for reuse when all the load and store bits associated with the row are cleared. Every cycle, it is possible to commit any number of loads and as many stores as the number of write ports to the data cache. The ARB head pointer is moved forward by one stage for every committed memory reference. The formal algorithm for committing a memory reference is given below.

```
Commit_Memory_Reference(Addr)
{
        Determine ARB Bank corresponding to Addr
        Associatively check address fields of Bank to find Row | Row's address field = Addr
        if (Row has a store mark in stage ARB Head)
                Send value in stage ARB Head of Row to Data Cache
        Clear load/store mark of stage ARB Head in Row
        if (there is no load/store mark in Row)
                Clear Addr field of Row
        Advance ARB Head by one stage
}
```

When a memory reference with sequence number *i* is executed and no row entry is obtained because the relevant ARB bank is full (with other addresses), then special recovery actions are taken. The ARB bank is checked to find addresses to which loads and stores have been executed only from stages succeeding to stage *i* in the active ARB window. If such addresses are found, one of them (preferably the one whose oldest executed reference is closest to the ARB tail) is selected, and the instructions including and beyond the reference corresponding to the sequentially oldest load mark or store mark in that entry are squashed. This action will enable that row entry to be reclaimed, and be subsequently allotted to the memory reference occurring in stage *i*. If no ARB row entry is found in which the oldest executed reference is from a succeeding stage, then the memory reference at stage *i* is stalled until (i) a row entry becomes free in that bank, or (ii) this reference is able to evict from that bank a row entry corresponding to another memory address. One of these two events is guaranteed to occur, because references only wait (if at all) for previous references to be committed. Thus, by honoring the sequential program order in evicting ARB row entries, deadlocks are prevented. The formal algorithm for handling the situation when an ARB bank is full is given below.

```
Recovery_Bank_Full(Bank, Sequence)
{
      Check in Bank for rows with no load/store marks before stage Sequence
      if  (there is any such row)
      {
            Determine row whose earliest load/store mark is closest to ARB tail
            Squash(sequence number of earliest load/store mark)  /* c.f. Sec. 3.4 */
            return(1)
      }
      return(0)
}
```

3.6. ARB Working Example

The concepts of ARB can be best understood by going through an example. Consider the sequence of loads and stores in Table 1, which form a part of a sequential piece of code (this example code does not contain instructions other than memory references, for the purpose of clarity). The ‘‘Sequence Number’’ column in the table gives the sequence number assigned by a dynamically scheduled processor to these memory references. The ‘‘Correct Address’’ column gives the addresses of the memory references in a correct execution of the program. The ‘‘Store Value’’ column gives the values stored by the two store instructions. The ‘‘Exec. Order’’ column shows the order in which the loads and stores are executed by the processor. For simplicity of presentation, the example does not consider the execution of multiple references in the same cycle. Figure 4 shows the progression of the instruction window contents and the ARB contents as the memory references are executed. There are 7 sets of figures in Figure 4, one below the other, each set representing a distinct clock cycle. The maximum size of the instruction window and the number of ARB stages in this example are both 4, and the ARB has a single bank. The active instructions in the instruction window are shown in darker shade. The head and tail units are marked by H and T, respectively. Blank entries in the ARB indicate irrelevant data.

Assume that all 4 instructions have been fetched into the instruction window, as shown by the shaded slots in the first set of figures. The 4 references have been allotted their sequence numbers from 0 to 3. The first set of figures in Figure 4 depicts the execution of the load with sequence number 2. This load is to address 100, and because there is no ARB row entry for address 100, a new ARB row entry is allotted for address 100. The load bit of stage 2 of this row entry is set to 1. The load request is sent to the data cache. Notice that an earlier store (with sequence number 0) is pending to the same address; so the value returned from the data cache (say 140) is incorrect. This causes the second load (the one with sequence number 3) to be issued to the incorrect address 140 instead of the correct address 120 (c.f. third set of figures in Figure 4). Notice that processors allowing speculative memory references will have some provision to prevent traps due to accesses to illegal addresses. The execution of the remaining memory references is depicted in the remaining sets of figures in Figure 4.

3.7. Novel Features of the ARB

The ARB performs dynamic memory disambiguation, and allows loads and stores to be executed out-of-order with respect to preceding references. Furthermore, it allows multiple memory references to be issued per cycle. It uses interleaving to decentralize the disambiguation hardware, and thereby reduce the associative search involved in the disambiguation process. Besides these features, the ARB efficiently supports the following features that are useful in dynamically reordering memory operations in an aggressive ILP processor.

Speculative Loads and Stores

The ARB supports speculative execution of loads and stores. It provides a good hardware platform for storing speculative store values, and correctly forwards them to subsequent speculative loads. Moreover, the speculative store values are not forwarded to the memory system until the stores are guaranteed to commit. Recovery operations are straightforward, because they involve only a movement of the tail pointer and the clearing of the appropriate load bit and store bit columns; the incorrect speculative values are automatically discarded.

Dynamically Unresolved Loads and Stores

The ARB supports dynamically unresolved loads and stores in processors that have provision for recovery (which may have been provided to support speculative execution of code or for fault-tolerant purposes). Thus, the ARB allows a load to be executed the moment its address is known, and even before the addresses of its preceding stores are known. Similarly, it allows a store to be executed the moment its address and store value are both known. Allowing dynamically unresolved loads could be important, because loads often reside in the critical path of programs, and undue detainment of loads could inhibit parallelism.

Memory Renaming

The ARB also supports memory renaming. Because it has the provision to store up to n values per address entry (where n is the number of ARB stages), it allows the processor to have up to n dynamic names for a memory location. Memory renaming is analogous to register renaming; providing more physical storage allows more parallelism to be exploited [2]. However, if not used with caution, the memory renaming feature of the ARB could lead to untoward recovery actions because of loads inadvertently fetching incorrect values from the ARB when multiple stores to the same address are present in the active instruction window. The full potential offered by the memory renaming capability of the ARB is a research area that needs further investigation.

4. ARB EXTENSIONS

4.1. Handling Variable Data Sizes

Many instruction set architectures allow memory references to have byte addressability and variable data sizes, such as bytes, half-words (16 bits), and full-words (32 bits). If a machine supports partial-word stores, and the ARB keeps information on a full word-basis (to reduce overheads, because most of the memory references are to full words), then a minor extension is needed to handle partial-word stores. This is because subsequent loads to the same word may require the full-word value. The extension that we adopt for this is as follows: when a partial-word store is executed, the value corresponding to the full-word is stored in the ARB. This is accomplished by treating a partial-word store as the combination of a full-word load followed by a full-word store. Thus, when a partial-word store with sequence number i is executed, both the load bit and the store bit of stage i in the ARB entry are set to 1.

4.2. Increasing the Effective Number of ARB Stages—Two-Level Hierarchical ARB

The number of stages in the ARB is a restricting factor on the size of the instruction window in which memory references can be reordered. The average size of the instruction window is upper bound by n/f_m, where n is the number of ARB stages and f_m is the fraction of instructions that are memory references. For instance, if n = 8 and f_m = 1/4, the average window size is limited to 32 instructions. The number of ARB stages cannot be increased arbitrarily, because the complexity of the logic that checks for out-of-order accesses (c.f. section 3.4) increases super-linearly with the number of ARB stages. However, there is an easy way to solve this problem—map multiple memory references to the same ARB stage. One way to map multiple references to an ARB stage is to divide the stream of memory references into *groups* (with the memory references in each group having consecutive sequence numbers). Dividing the memory reference stream into groups divides the memory disambiguation job into two subjobs: (i) *intra-group memory disambiguation* and (ii) *inter-group memory disambiguation*. The first guarantees that memory dependency violations do not occur from a group. The second guarantees that memory dependency violations do not occur from multiple groups, given that there are no memory dependency violations within each group.

The hardware structure that we propose to use for performing the intra-group and inter-group memory disambiguations is a two-level hierarchical ARB. The bottom level of the hierarchy consists of several local ARBs, and the top level consists of a single global ARB. There are as many local ARBs as the maximum number of memory reference groups allowed in the instruction window, and each local ARB is responsible for carrying out the intra-group disambiguation of its group.

The global ARB has as many stages as the number of local ARBs; each stage corresponds to a group of memory references, as opposed to a single memory reference or sequence number. The load bits for a stage indicate the loads that have been executed from the corresponding group, and the store bits indicate the stores that have been executed from the corresponding group. The *value* fields associated with each stage are used to store the values of the stores that have been executed from that group. If multiple stores are executed from a group to the same address, the sequentially latest value is stored in the value field of the global ARB.

Figure 5 shows the block diagram of a two-level hierarchical ARB. In the figure, the global ARB is interleaved into 4 banks as before. Each ARB bank can hold up to 4 address entries, and has 6 stages, corresponding to 6 memory reference groups. Each group has an 8-stage local ARB, which is not interleaved, and can store up to 4 address entries. The active instruction window encompasses memory reference groups 1 through 4. Hierarchical ARBs help to amplify the number of memory references in the active instruction window. Specifically, a two-level hierarchical ARB allows as many memory references in an instruction window as the sum of the number of stages in all the local ARBs. Thus, if there are 6 local ARBs and each of them has 8 stages, then altogether $6 \times 8 = 48$ memory references can be present in the instruction window at the same time. Consequently, the upper limit to the instruction window size increases to approximately 200 instructions (assuming f_m to be 1/4).

To execute a load or a store from memory reference group G, the request is first sent to the G^{th} local ARB. The local ARB handles the reference much the same way as the ARB of Section 3, with the exception that (i) when a load ''misses'' in the local ARB, instead of forwarding it to the data cache, it is forwarded to the global ARB, and (ii) when a store is executed to address A, if it is found that the store is the sequentially latest store encountered so far from group G to address A, then the store request is also sent to the global ARB. The global ARB processes the requests it receives in the same way as the ARB described in Section 3. The formal algorithms for executing loads and stores in a two-level hierarchical ARB are given in the appendix. It is important to note that the reason for using a hierarchical ARB structure is not to minimize the traffic to the (global) ARB, but to reduce the number of ARB stages over which checking for the closest preceding store (or closest succeeding load) has to be done when a load (or store) request is sent to the ARB.

5. PERFORM ANCE EVALUATION

The previous sections introduced the ARB and described its working. This section studies its effectiveness, and presents the results of an empirical study that evaluates the ARB's performance in a superscalar processor. Because the emphasis in this paper is on the presentation of a new technique for memory operation reordering, we limit ourselves to a basic evaluation of its effectiveness. Very detailed evaluation studies of the ARB, assessing the impact of the number of banks, bank size, their variations with the issue strategy, etc., are not the thrust of this paper. Accordingly, our evaluation is limited to a comparison of one ARB organization with equivalent organizations of existing solutions.

5.1. Experimental Framework

Our methodology of experimentation is simulation. We have developed a superscalar simulator that uses the MIPS R2000 - R2010 instruction set and functional unit latencies [10]. This simulator accepts executable images of sequential programs (compiled for MIPS R2000-based machines), and simulates their execution, keeping track of relevant information on a cycle-by-cycle basis. It models speculative execution, and is not trace driven. The simulator also incorporates a mini-operating system to handle the system calls made by the simulated program. Because of the detail at which the simulation is carried out, and because the entire memory system is modeled, the simulator is slow. This speed restricts our ability to explore the design space in great detail using substantial runs of large benchmark programs. In order to do a reasonable study, we fixed several parameters. The parameters that were fixed for this study are listed below.

- The benchmarks are run for 100 million instructions each (less for *compress*, as it finishes earlier).
- The processing paradigm used for the studies is a superscalar processor that performs speculative execution. It constructs a dynamic sliding instruction window of up to 64 instructions by using a two-level branch prediction scheme [19]. The degree of superscalar execution is 8, *i.e.*, up to 8 instructions can be fetched in a cycle, and up to 8 instructions can be issued in a cycle.
- The data cache is 64Kbytes, 4-way set-associative, non-blocking, and has an access latency of 1 cycle. The interleaving factor of the data cache is 32. The data cache miss latency to get the first word is 4 cycles (assuming the presence of a second level data cache with 100% hit ratio).
- The instruction cache is 64Kbytes, 4-way set-associative, and has an access latency of 1 cycle.
- The ARB used is a two-level hierarchical ARB. The global ARB has 8 stages and each local ARB also has 8 stages. The global ARB and the local ARBs are interleaved into 32 banks and 4 banks, respectively. Each ARB bank has 8 row entries, and is fully associative.

We have simulated 3 hardware structures for reordering memory references—the store queue, the dependency matrix, and the ARB. The store queue described in Section 2 is augmented with capabilities for speculative execution. All 3 structures allow up to 8 memory references to be executed per cycle. Both the store queue and the dependency matrix do a 64×63 associative compare in the worst case. In that sense, the store queue and dependency matrix configurations used in this study are hypothetical at best, based on current technology.

For the ARB, we have simulated two different schemes. The first scheme is same as the two-level hierarchical ARB described in the previous section. In the second scheme, the ARB state information is slightly expanded to keep track of pending stores that have not yet been executed, but whose addresses are known. When a store address becomes known, it is pre-entered into the ARB and the state information is updated to reflect the fact that a store with a particular sequence number is pending to that address. When a load is executed, if the closest preceding store to the same address in the ARB is a pre-entered store, then the load instruction waits until the relevant store value becomes available. If the store information were not pre-entered, then the load would have fetched an incorrect value from the data cache, only to result in a squashing of itself and other (useful) instructions. The pre-entering of store information thus helps to reduce the number of recovery actions due to incorrect dynamically unresolved references. Notice that for the dependency matrix, pre-entering of store addresses is effectively done, if dependency bits are cleared when the store addresses are first known.

5.2. Benchmarks and Performance Metrics

For benchmarks, we use a subset of the SPEC '92 benchmarks. Benchmark programs are compiled to MIPS R2000 - R2010 executables using the MIPS compiler.

Execution time is the sole metric that can accurately measure the performance of an integrated software-hardware computer system. Metrics such as instruction issue rate and instruction completion rate, are not very accurate in general because the compiler may have introduced many redundant operations. However, to use execution time as the metric, the programs have to be run to completion, which further taxes our already time-consuming simulations. Furthermore, we use the same executable for all the simulations, with no changes instituted by a compiler. Due to these reasons, we use instruction completion rate as the primary metric. Note that speculatively executed instructions whose execution was not required are not counted while calculating the instruction completion rate. We also use another metric—percentage incorrect references—to throw more light on the performance of the ARB. This metric indicates the fraction of memory references that were incorrect (because of executing them as dynamically unresolved references), and resulted in recovery actions.

5.3. Performance Results

Table 2 presents the instruction completion rates that we obtained in our simulation experiments for the four schemes. It can be seen that both ARB schemes perform very favorably compared to the hypothetical 64-entry store queue and dependency matrix. The ability to support dynamically unresolved references even gives the ARB (with no pre-entering of store information) a small performance advantage over the 64-entry dependency matrix for some of the benchmarks. When store information is pre-entered in the ARB, its performance results are better than the other schemes for all the benchmarks. The results show that decentralizing the memory disambiguation mechanism by interleaving has great potential for use in future ILP processors.

It is worthwhile to study how many of the (dynamically unresolved) references resulted in incorrect execution for the ARB schemes. Table 3 presents the percentage of incorrect references for the two ARB schemes. It can be seen that supporting dynamically unresolved references causes only a negligible percentage of the references to be incorrect. The fundamental reason for this phenomenon is that compilers tend to keep frequently accessed variables in registers, and if register spilling is less, then it is rare that a value is written to a memory location, and then immediately (*i.e.* within the span of an instruction window) read to a CPU register.

With improvements in the instruction issue strategies in the future, the dynamic instruction window size and the number of memory references in the dynamic window will also increase. In this scenario, the traditional memory reordering schemes such as the store queue and the dependency matrix will become less feasible because of their wide associative compares. The 2-level hierarchical ARB, on the other hand, can easily support window sizes to the tune of 200 instructions, and can be extended to even more levels of hierarchy.

6. APPLICATION TO MULTISCALAR PROCESSOR

The discussion of ARB in section 3 was based primarily on the superscalar processor as the underlying execution model. In this section, we demonstrate how the ARB can be used in a different execution model, namely the multiscalar model [5, 6, 18].

6.1. Multiscalar Processor

The multiscalar processor was earlier known as the ESW (Expandable Split Window) processor [5]. True to its name at inception, the multiscalar processor splits a large window of instructions into multiple tasks, and exploits parallelism by overlapping the execution of these tasks. The processor consists of several independent, identical *execution units*, each of which is equivalent to a typical datapath found in modern processors. The execution units conceptually form a circular queue, with hardware pointers to the head and tail of the queue. These pointers are managed by a control unit, which also performs the function of assigning tasks to the execution units. Every cycle, the control unit assigns a new task to the tail unit (using control flow prediction [13] if required) unless the circular unit queue is full. The active units, the ones from the head to the tail, together constitute the large dynamic window of instructions, and they contain tasks in the sequential order in which the tasks appear in the dynamic instruction stream. In any given cycle, up to a fixed number of ready-to-execute instructions begin execution in each of the active execution units. It is possible to have out-of-order execution in a unit, if desired. When all the instructions in the unit at the head have completed execution, the unit is committed, and the head pointer is moved forward to the next unit. Further details of the multiscalar processor can be had from [5, 6, 18].

6.2. The Problem of Memory Reference Reordering in the Multiscalar Processor

In a multiscalar processor, at any given time, many sequential tasks may have been initiated and executed in parallel, but not all instructions of these tasks may have been fetched at that time. This is illustrated in Figure 6. Figure 6(i) shows 4 multiscalar tasks, which as per sequential program semantics, follow one after the other. Memory reference instructions in these tasks are specifically identified. Figure 6(ii) illustrates the scenario when these tasks are executed in a 4-unit multiscalar processor. In the figure, the vertical axis indicates time in cycles; unshaded rectangular blocks indicate instructions that have been fetched in the corresponding cycles. Each cycle, a new task is allocated to a free execution unit. In cycle 0, execution unit 0 starts fetching instructions from task 0; in cycle 1, execution unit 1 starts fetching instructions from task 1; and so on. At the end of cycle 4, the load instructions in tasks 1, 2, and 3 have been fetched, but the store instruction in task 0 has not been fetched (as indicated by the shaded rectangular block in Figure 6(ii)). Furthermore, the multiscalar execution mechanism is unaware of the presence of the store instruction in task 0 until the store is fetched. In general, a load instruction cannot therefore be guaranteed to be free of conflicts until all instructions of all preceding tasks have been fetched. If a memory operation reordering structure such as a store queue or a dependency matrix is used, then much of the execution in a multiscalar processor is serialized because the loads have to wait until all preceding instructions are fetched. This is in contrast to the case with an ARB, which allows the loads to be executed as dynamically unresolved loads.

6.3. Two-Level Hierarchical ARB for a Multiscalar Processor

The memory disambiguation job for a multiscalar processor can be divided into two subjobs: (i) intra-task memory disambiguation, and (ii) inter-task memory disambiguation. This division falls naturally to that adopted by the two-level hierarchical ARB—the set of memory references in a multiscalar task can be considered as a single group for memory disambiguation purposes.

The global ARB of the two-level hierarchical ARB in a multiscalar processor has as many stages as the number of execution units in the processor. The load bits for a stage indicate the loads (possibly unresolved within the execution unit) that have been executed from the corresponding unit of the multiscalar processor, and the store bits indicate the stores (possibly unresolved within its execution unit) that have been executed from that unit. The *value* fields associated with each stage have the same function as before, namely, to store the values of the stores that have been executed from that unit.

When the execution unit at the head is committed, all load and store marks in its local ARB and the head stage of the global ARB are erased immediately. Also, all store values stored in that stage are forwarded to the data cache. If there are multiple store values in a stage, this forwarding could cause a traffic burst, preventing that ARB stage from being reused until all the store values have been forwarded. One simple solution to alleviate this problem is to use a write buffer to store all the values that have to be forwarded from a stage. Another solution is to have more physical ARB stages than the number of multiscalar stages.

7. SUMMARY AND CONCLUSIONS

We have proposed a hardware structure called an Address Resolution Buffer (ARB) for dynamically reordering memory references. This structure reduces the width of the associative search involved in memory address disambiguation by using the concept of interleaving, and by altering the order of operations (sequence number checking and address comparison) needed to carry out the disambiguation process. It has the full power of a hypothetical scheme that reorders memory references by means of associative compares of the memory addresses of all loads and stores in the entire instruction window. It allows speculative loads and speculative stores by keeping the uncommitted store values in the structure and forwarding them to subsequent loads that require the value. Furthermore, it allows memory references to be executed before they are disambiguated from the preceding stores and even before the addresses of the preceding stores are known. The ARB also allows memory renaming; it allows as many distinct values for a memory location as the number of stages in the ARB. The ARB is, to the best of our knowledge, the first decentralized design for performing dynamic disambiguation in a large window of instructions.

We have also proposed several extensions to the ARB, such as hierarchical ARBs, and extensions for handling variable data sizes. We also presented the results of a simulation study that evaluates the potential of ARB for the superscalar processing model. The ARB was found to perform better than (hypothetical) schemes using store queue and dependency matrix that do associative compares of all addresses within the instruction window. The ARB has a less associative search (because of interleaving), yet better performance (because of supporting dynamically unresolved references). The results show that ARB has great potential to meet the memory reordering demands of future ILP processors, which perform speculative execution and reordering of code within a large instruction window.

Finally, we demonstrated the application of the ARB for the multiscalar processing model, for which traditional reordering structures such as the store queue and the dependency matrix are unsuitable, because the processor advances past blocks of code that have not yet been fetched (and the store addresses in those blocks not known until later). The two-level hierarchical ARB uses the concept of splitting a large job (memory disambiguation within a large window) into smaller subjobs, and it was shown to tie well with the multiscalar execution model.

REFERENCES

[1] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, ‘‘The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling,’’ *IBM Journal of Research and Development*, pp. 8-24, January 1967.

[2] T. M. Austin and G. S. Sohi, ‘‘Dynamic Dependency Analysis of Ordinary Programs,’’ *Proc. 19th Annual International Symposium on Computer Architecture*, 1992.

[3] L. J. Boland, G. D. Granito, A. U. Marcotte, B. U. Messina, and J. W. Smith, ‘‘The IBM System/360 Model 91: Storage System,’’ *IBM Journal*, pp. 54-68, January 1967.

[4] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, 1986.

[5] M. Franklin and G. S. Sohi, ‘‘The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism,’’ *Proc. 19th Annual International Symposium on Computer Architecture*, pp. 58-67, 1992.

[6] M. Franklin, ‘‘The Multiscalar Architecture,’’ Ph.D. Thesis, Computer Sciences Department, University of Wisconsin-Madison, 1993. Also Technical Report TR 1196, Computer Sciences Department, University of Wisconsin-Madison, 1993.

[7] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu, B. Heggy, and M. L. Soffa, ‘‘Architectural Support for Register Allocation in the Presence of Aliasing,’’ *Proc. Supercomputing ’90*, pp. 730-739, November 1990.

[8] W. W. Hwu, ‘‘Exploiting Concurrency to Achieve High Performance in a Single-chip Microarchitecture,’’ Ph.D. Thesis, Report No. UCB/CSD 88/398, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1988.

[9] M. Johnson, *Superscalar Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1990.

[10] G. Kane, *MIPS R2000 RISC Architecture*. Englewood Cliffs, New Jersey: Prentice Hall, 1987.

[11] A. Nicolau, ‘‘Run-Time Disambiguation: Coping With Statically Unpredictable Dependencies,’’ *IEEE Transactions on Computers*, vol. 38, pp. 663-678, May 1989.

[12] Y. N. Patt, S. W. Melvin, W. W. Hwu, and M. Shebanow, ‘‘Critical Issues Regarding HPS, A High Performance Microarchitecture,’’ in *Proc. 18th Annual Workshop on Microprogramming*, Pacific Grove, CA, pp. 109-116, December 1985.

[13] D. N. Pnevmatikatos, M. Franklin, and G. S. Sohi, ‘‘Control Flow Prediction for Dynamic ILP Processors,’’ *Proc. The 26th Annual International Symposium on Microarchitecture (MICRO-26)*, pp. 153-163, 1993.

[14] G. M. Silberman and K. Ebcioglu, ‘‘An Architectural Framework for Migration from CISC to Higher Performance Platforms,’’ *Proc. International Conference on Supercomputing*, pp. 198-215, 1992.

[15] J. E. Smith and A. R. Pleszkun, ‘‘Implementing Precise Interrupts in Pipelined Processors,’’ *IEEE Transactions on Computers*, vol. 37, pp. 562-573, May 1988.

[16] J. E. Smith, ‘‘Dynamic Instruction Scheduling and the Astronautics ZS-1,’’ *IEEE Computer*, pp. 21-35, July 1989.

[17] G. S. Sohi, ‘‘Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers,’’ *IEEE Trans. on Computers*, vol. 39, pp. 349-359, March 1990.

[18] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, ‘‘Multiscalar Processors,’’ *Proceedings of 22nd Annual International Symposium on Computer Architecture*, 1995.

[19] T. Y. Yeh and Y. N. Patt, ‘‘Alternative Implementations of Two-Level Adaptive Training Branch Prediction,’’ *Proceedings of 19th Annual International Symposium on Computer Architecture*, pp. 124-134, 1992.

APPENDIX

The appendix gives the formal algorithms for performing the execute phase of loads and stores. Execution of a load starts with Execute_Load_Local_ARB(), and execution of a store starts with Execute_Store_Local_ARB(). A memory reference is committed only after it is known that all references in its group are guaranteed to be committed. The algorithms given below are meant to depict the semantics for the correct working of the two-level hierarchical ARB; an implementation can change the order of algorithm statements or do multiple statements in parallel so long as it preserves correctness.

```
Execute_Load_Local_ARB(ARBnum, Addr, Sequence)
{
	Determine local ARBnum's Bank corresponding to Addr
	Associatively check address fields of Bank to find Row | Row's address field = Addr
	if  (Addr not present)
	{
		if  (Bank full)
		{
			Freed_Up = Recovery_Bank_Full(Bank, Sequence mod Num_Local_Stages)
			if (Freed_Up == 0)
				return(0)
		}
		Allocate new Row entry in Bank
		Enter Addr in address field of Row
		if  (Execute_Load_Global_ARB(Addr, Sequence) == 0)
			return(0)
	}
	else
	{
		Check in Row to determine closest preceding store executed to Addr
			if  (no preceding store found)
			if  (Execute_Load_Global_ARB(Addr, Sequence) == 0)
				return(0)
		else
			Forward the value from its value field to the processor
	}
	Set Row's load bit of stage (Sequence mod Num_Local_Stages) to 1
	return(1);
}
```

```
Execute_Load_Global_ARB(Addr, Sequence)
{
	Determine global ARB's Bank corresponding to Addr
	Associatively check address fields of Bank to find Row | Row's address field = Addr
	if  (Addr not present)
	{
		if  (Bank full)
		{
			Freed_Up = Recovery_Bank_Full(Bank, Sequence / Num_Global_Stages)
			if (Freed_Up == 0)
				return(0)
		}
		Allocate new Row entry in Bank
		Enter Addr in address field of Row
		Send load request to data cache
	}
	else
	{
		Check in Row to determine closest preceding store mark in Row
		if  (no preceding store found)
			Send load request to data cache
		else
```

```
            Forward the value from its value field to the processor
      }
      Set Row's load bit of stage (Sequence / Num_Global_Stages) to 1
      return(1);
}
```

```
Execute_Store_Local_ARB(ARBnum, Addr, Value, Sequence)
{
      Determine local ARBnum's Bank corresponding to Addr
      Associatively check address fields of Bank to find Row | Row's address field = Addr
      if  (Addr not present)
      {
            if  (Bank full)
            {
                  Freed_Up = Recovery_Bank_Full(Bank, Sequence mod Num_Local_Stages)
                  if (Freed_Up == 0)
                        return(0)
            }
            Allocate new Row entry in Bank
            Enter Addr in address field of Row
            if  (Execute_Store_Global_ARB(Addr, Value, Sequence) == 0)
                  return(0)
      }
      else
      {
            Check in Row to determine stage number S_L of closest succeeding load mark,
                                    with no intervening store marks
            if  (succeeding load with no intervening stores found)
            {
                  Squash_Global_ARB((ARBnum + 1) mod Num_Local_Stages)
                  Squash_Local_ARB(ARBnum, S_L)
                        if  (Execute_Store_Global_ARB(Addr, Value, Sequence) == 0)
                        return(0)
            }
            else
            {
                  Check in Row for succeeding stores executed to Addr
                  if  (no succeeding stores found)
                        Execute_Store_Global_ARB(Addr, Value, Sequence)
            }
      }
      Enter Value in the value field of stage (Sequence mod Num_Local_stages) of Row
      Set Row's store bit of stage (Sequence mod Num_Local_stages) to 1
      return(1);
}
```

```
Execute_Store_Global_ARB(Addr, Value, Sequence)
{
      Determine global ARB's Bank corresponding to Addr
      Associatively check address fields of Bank to find Row | Row's address field = Addr
      if  (Addr not present)
      {
            if  (Bank full)
            {
                  Freed_Up = Recovery_Bank_Full(Bank, Sequence / Num_Global_Stages)
                  if (Freed_Up == 0)
                        return(0)
            }
            Allocate new Row entry in Bank
            Enter Addr in address field of Row
      }
      else
      {
```

```
        Check in Row to determine stage number S_L of closest succeeding load mark,
                                with no intervening store marks
                if  (succeeding load mark with no intervening store marks found)
                        Squash_Global_ARB(S_L)

        }
        Enter Value in the value field of stage (Sequence / Num_Global_Stages) of Row
        Set Row's store bit of stage (Sequence / Num_Global_Stages) to 1
        return(1);
}
```

```
Commit_Memory_Reference_Local_ARB(Addr)
{
        Determine local ARB Bank corresponding to Addr
        Associatively check address fields of Bank to find Row | Row's address field = Addr
        if (there is no store mark in any other stage of Row)
        {
                Commit_Memory_Reference_Global_ARB(Addr)
                if (ARB Head + 1 == ARB Tail)
                        Advance Global ARB Head by 1 stage
        }
        Clear load/store mark of stage ARB Head of Row
        if (there is no load mark or store mark in Row)
                Clear Addr field of Row
        Advance local ARB Head by 1 stage
}
```

```
Commit_Memory_Reference_Global_ARB(Addr)
{
        Determine global ARB Bank corresponding to Addr
                Associatively check address fields of Bank to find Row | Row's address field = Addr
        if  (Addr present)
        {
                if (there is a store mark in stage Global ARB Head of Row)
                        Send value in stage ARB Head of Row to Data Cache
                Clear load mark and store mark of stage Global ARB Head of Row
                if (there is no load mark or store mark in Row)
                        Clear Addr field of Row
        }
}
```

```
Squash_Local_ARB(ARBnum, Starting_Stage)
{
        for local ARBnum's each Bank do
        {
                for each Row in Bank do
                {
                        for each Stage from ARB Tail (not including) to Starting_stage do
                        {
                                if (there is a store mark in Stage)
                                {
                                        Check in Row to find stage Store_Stage with
                                                        closest preceding store mark
                                        Addr = Addr field of Row
                                        if  (preceding store mark found)
                                        {
                                                Store Value = Value field of Store_Stage
                                                Sequence = ARBnum * Num_Global_Stages + Store_Stage
                                                Execute_Store_Global_ARB(Addr, Value, Sequence)
                                        }
                                        else
                                                Delete_from_Global_ARB(Addr, Stage)
```

```
                }
                        else if (there is a there is a load mark in Stage)
                        {
                                if (there is no previous load mark or store mark)
                                        Delete_from_Global_ARB(Addr field of Row, Stage)
                        }
                        Clear load/store marks of Stage
                        if (there is no load mark or store mark in Row)
                                Clear Addr field of Row
                }
        }
    }
    Set ARB Tail = Starting_Stage
}
```

```
Squash_Global_ARB(Starting_Stage)
{
    for each Stage from Starting_Stage to ARB Tail (not including) do
        Clear_Local_ARB(Stage)
    for global ARB's each Bank do
    {
        for each Stage from Starting_Stage to ARB Tail do
            Clear all load marks and store marks in Stage
        for each Row in Bank do
        {
            if (there is no load mark or store mark in Row)
                Clear Addr field of Row
        }
    }
    Set ARB Tail = Starting_Stage
}
```

```
Delete_from_Global_ARB(Addr, Stage)
{
    Determine global ARB's Bank corresponding to Addr
    Associatively check address fields of Bank to find Row | Row's address field = Addr
    if (Addr present)
    {
        Clear load/store marks of Stage in Row
        if (there is no load mark or store mark in Row)
            Clear Addr field of Row
    }
}
```

```
Clear_Local_ARB(ARBnum)
{
    for local ARBnum's each Bank do
    {
        for each Stage do
            Clear all load marks and store marks in Stage
        for each Row in Bank do
            Clear Addr field of Row
    }
    Set ARB Tail = 0
}
```

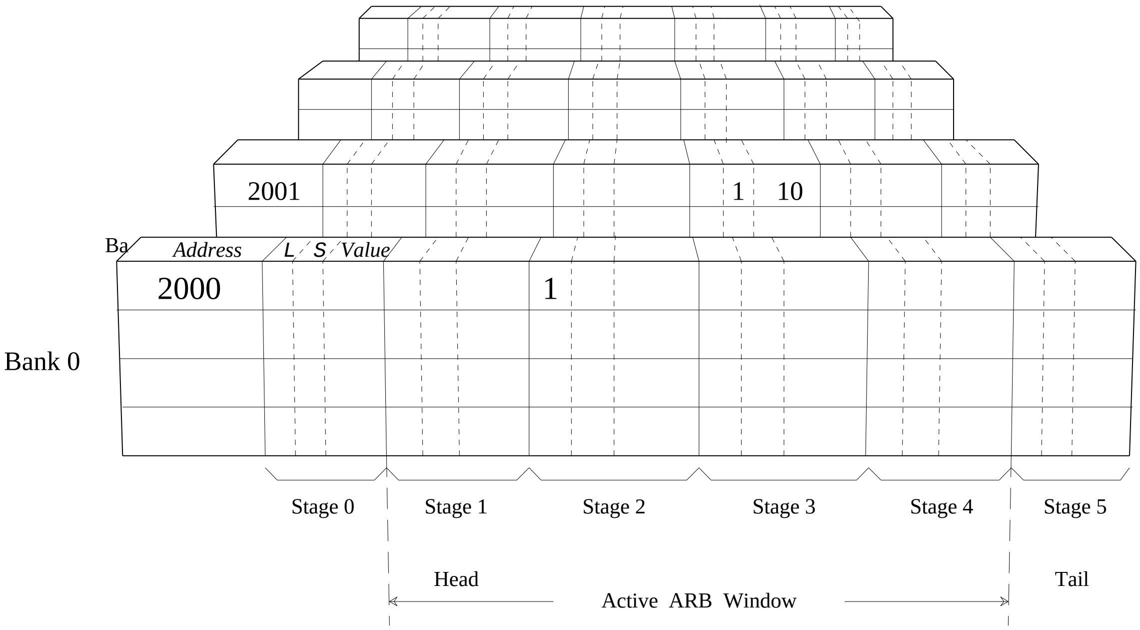


Figure 1: A 4-Way Interleaved, 6-stage ARB

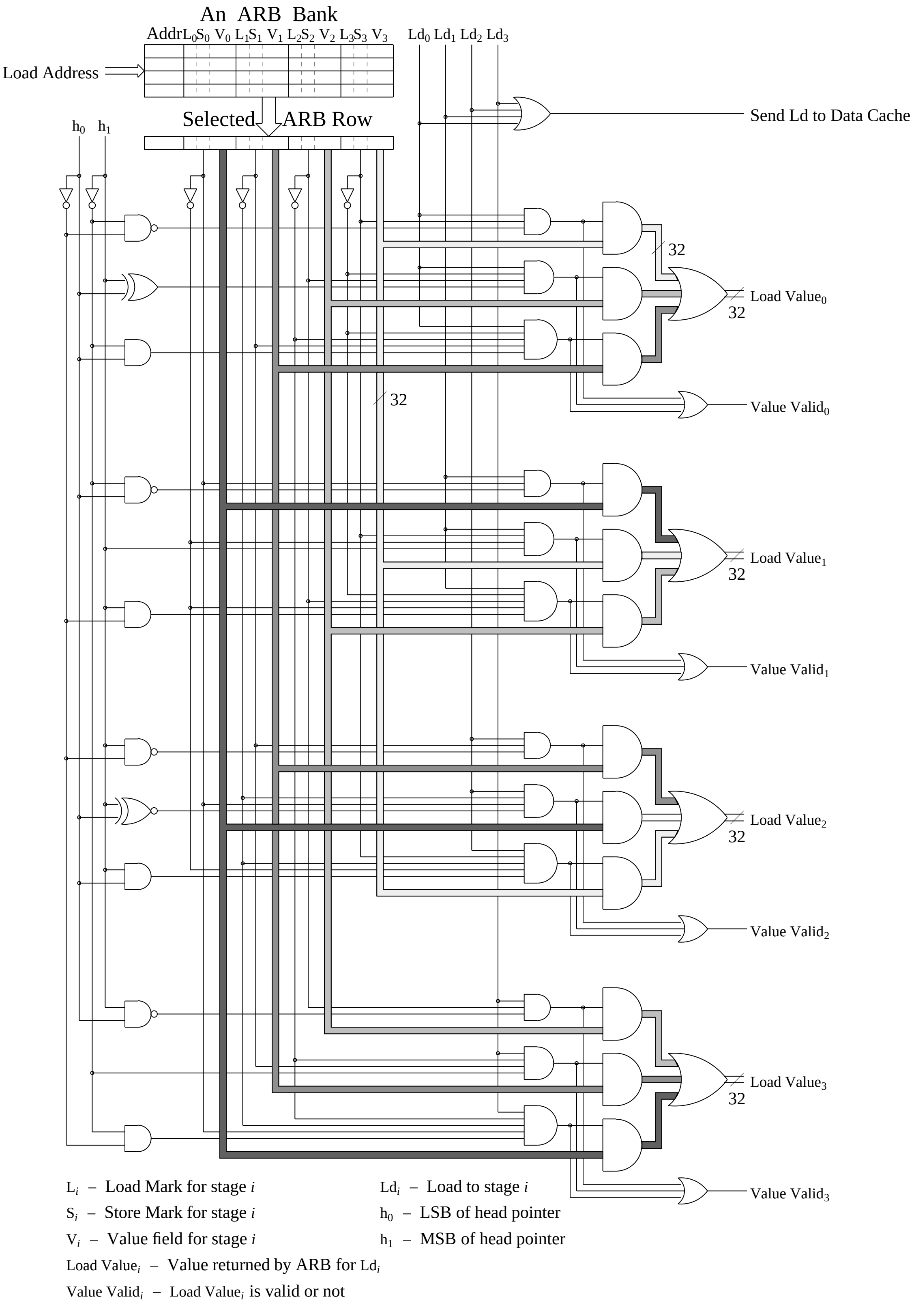
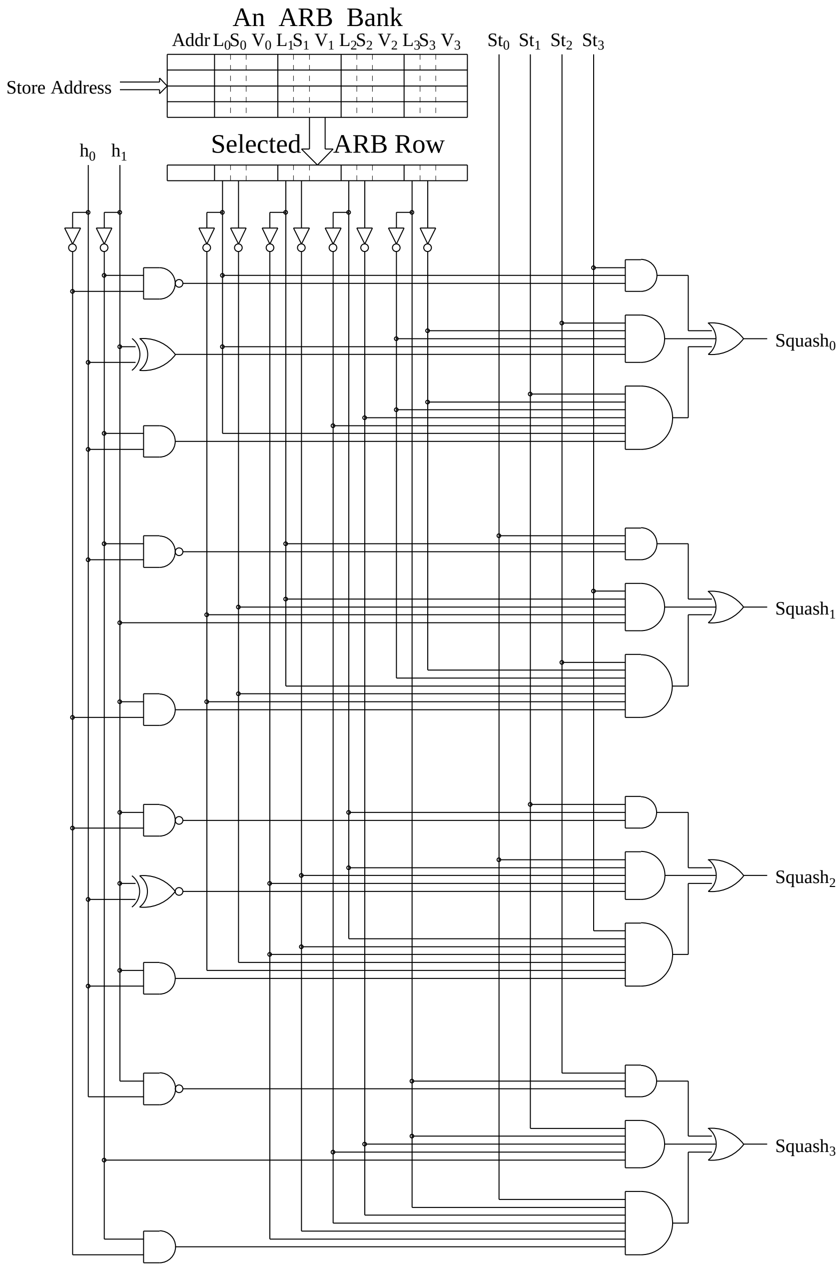


Figure 2: Logic Diagram for Executing Loads in a 4-stage ARB



L_i – Load Mark for stage *i*

S_i – Store Mark for stage *i*

V_i – Value field for stage *i*

St_i – Store to stage *i*

h_0 – LSB of head pointer

h_1 – MSB of head pointer

$Squash_i$ – Squash instructions from sequence number *i* onwards

$$\mathrm{Squash}_i \;=\; \mathrm{L}_i \wedge \left(\bigvee_{j=head}^{j=i-1} \left(\mathrm{St}_j \bigwedge_{k=j+1}^{k=i-1} \overline{\mathrm{L}}_k\, \overline{\mathrm{S}}_k \right) \right)$$

j varies as $\begin{cases} head \le j < i & \text{if } head \le i \\ head \le j \le n-1,\ 0 \le j < i & \text{if } head > i \end{cases}$ k varies as $\begin{cases} j < k < i & \text{if } j < i \\ j < k \le n-1,\ 0 \le k < i & \text{if } j > i \end{cases}$

Figure 3: Logic Diagram for Detecting Out-of-Order Load-Stores in a 4-stage ARB

Table 1: Example Sequence of Loads and Stores

Code	Sequence Number	Correct Address	Store Value	Exec. Order	Remarks
STORE R2, 0(R1)	0	100	120	4	Execution got delayed as R1 was unavailable
STORE R4, 0(R2)	1	120	50	2	
LOAD R3,–20(R2)	2	100		1	Unresolved load fetched incorrect value 140
LOAD R5, 0(R3)	3	120		3	

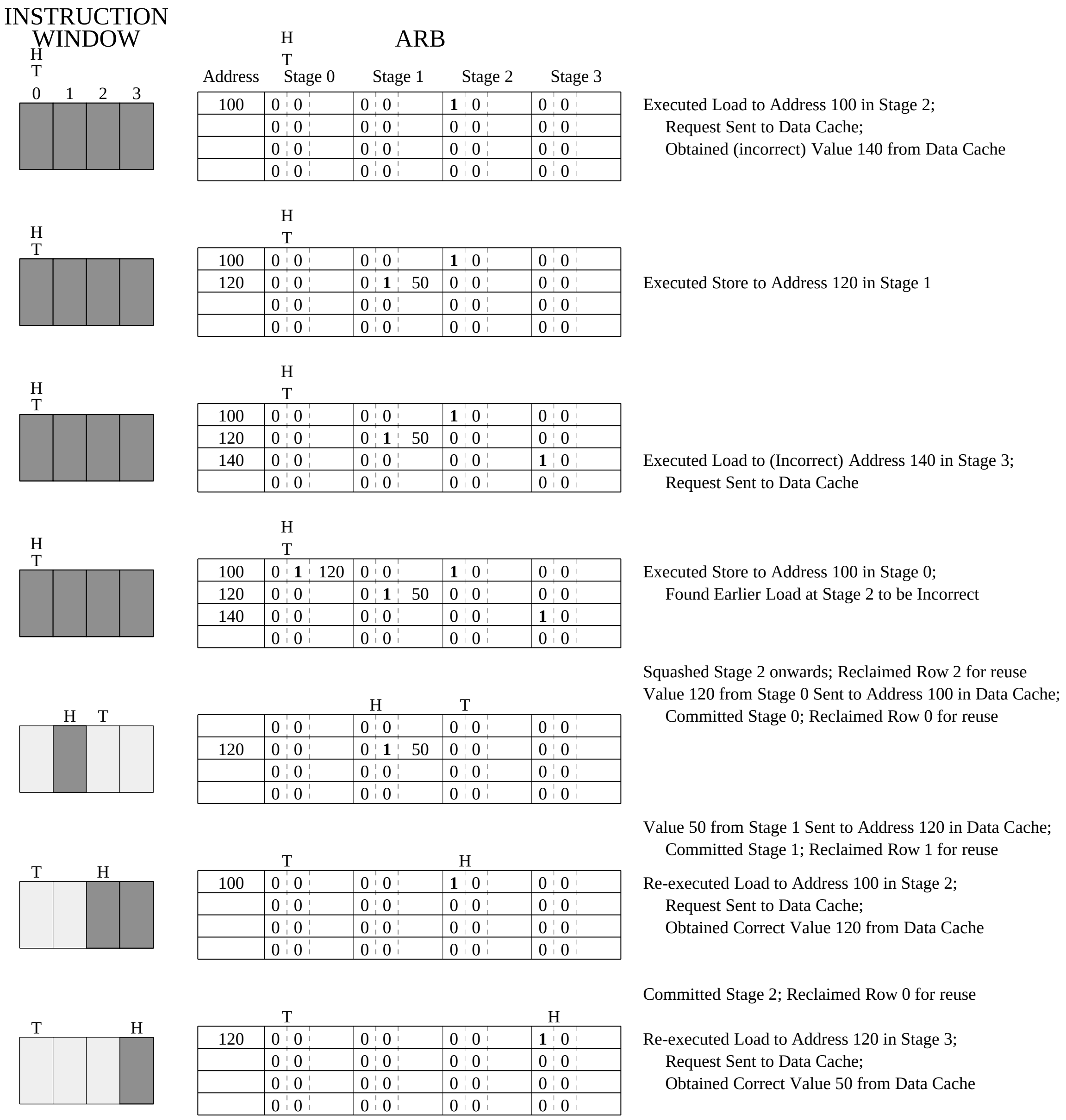


Figure 4: Contents of the Instruction Window and the ARB After the Execution of Each Memory Reference in Table 1

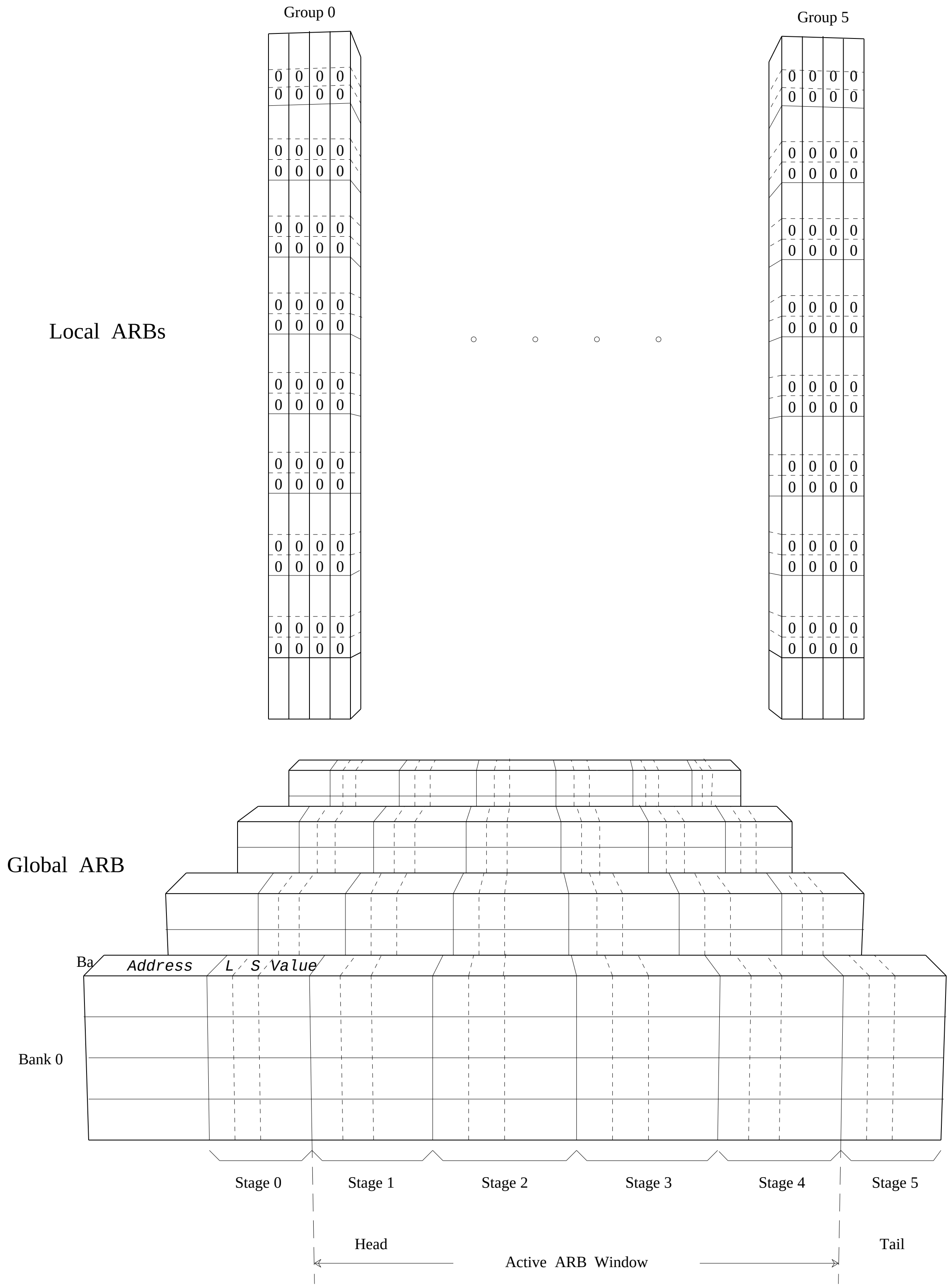


Figure 5: A Two-Level Hierarchical ARB

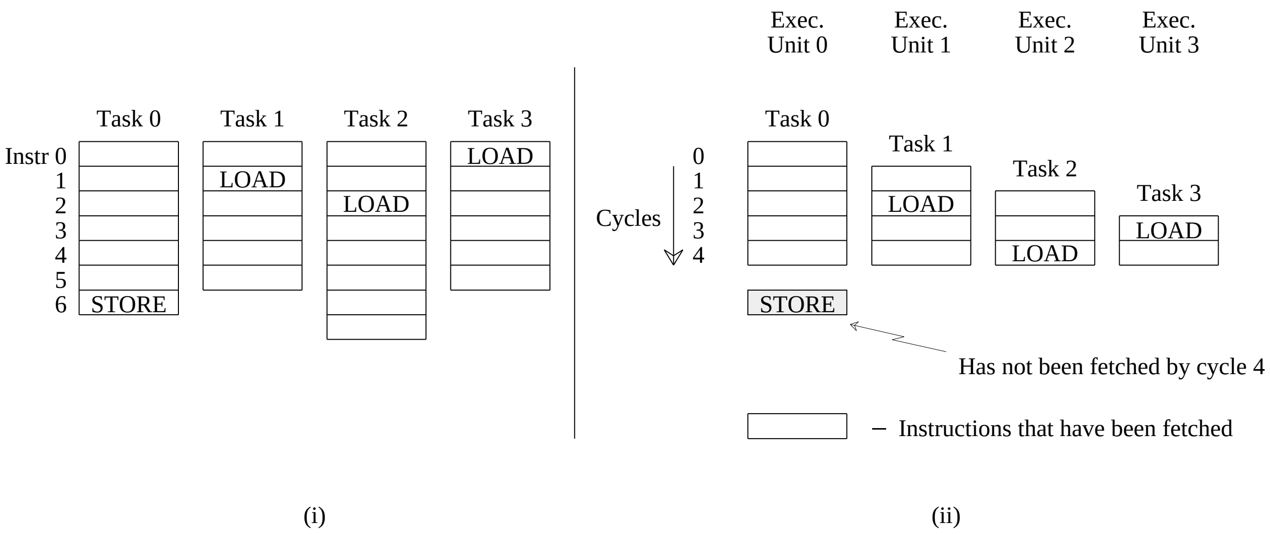


Figure 6: (i) 4 Multiscalar Tasks; (ii) Execution of the Tasks by 4 Multiscalar Execution Units

Table 2: Instruction Completion Rates

Benchmark	Instruction Completion Rates with			
	Store Queue	Dependency Matrix	ARB	
			No pre-entering of Store Information	Pre-entering of Store Information
compress	2.08	2.47	2.59	2.69
dnasa7	3.13	3.30	3.32	3.34
doduc	2.84	2.84	2.81	2.93
espresso	2.46	2.56	2.56	2.59
fpppp	3.39	3.57	3.47	3.69
sc	2.38	2.50	2.63	2.67
xlisp	2.25	2.49	2.53	2.57

Table 3: Percentage Incorrect References with ARB

Benchmark	Percentage Incorrect References	
	No pre-entering of Store Information	Pre-entering of Store Information
compress	0.69%	0.00%
dnasa7	0.04%	0.00%
doduc	0.96%	0.00%
espresso	0.36%	0.00%
fpppp	0.95%	0.00%
sc	0.23%	0.02%
xlisp	0.42%	0.00%