Multi-Version Caches for Multiscalar Processors

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

Abstract

The multiscalar processor, proposed recently for exploiting instruction-level parallelism, is a collection of execution units that are connected together using a ring-type network. This paper investigates the issue of decentralizing the memory system in the multiscalar processor, and proposes a decentralized scheme called multi-version caches. The central idea of this scheme is to have multiple versions of the (top level of) the data cache, which provide a hardware platform for storing the speculative memory values produced in the execution units. The multiple versions also allow precise state to be maintained at inter-unit boundaries, which facilitates easy recovery actions in times of incorrect task prediction. The paper also presents the results of simulation studies conducted using the SPEC benchmarks to evaluate multi-version caches. These results suggest that multi-version caches are an excellent choice for decentralizing the memory system in the multiscalar processor.

1 Introduction

Recent advances in VLSI technology have enabled millions of transistors to be placed in a single chip. Technology projections indicate that by A.D. 2000, about 100 million transistors can be incorporated in a chip [5]. One way these resources are being used is for incorporating performance-enhancing hardware features, targeted towards exploiting different kinds of parallelism present in the program. One processor, which shows great potential for exploiting instruction-level parallelism (ILP), is the multiscalar processor [2-4]. This processor exploits ILP by executing multiple blocks of code in parallel with the help of multiple execution units that are connected together in a decoupled and decentralized manner using a ring-type network. A major tenet of the multiscalar processing paradigm is to permit decentralization of all critical resources in the processor, so that the number of execution units can be scaled up, as and when allowed by technology improvements.

One feature of the multiscalar processor, which stems from its adherence to sequential execution semantics, is that its execution units share the same register space and the same memory address space. This means that the execution units need to synchronize at the level of registers and memory locations. Current implementations of the multiscalar processing paradigm use decentralized structures for carrying out inter-unit register communication, but use centralized structures for carrying out inter-unit memory communication [2], which hinder scalability. This paper proposes a scheme called multi-version caches to decentralize inter-unit memory communication in the multiscalar processor.

The rest of this paper is organized as follows. Section 2 provides background information on the multiscalar architecture. Section 3 presents multi-version caches, and section 4 describes a simulation-based evaluation of multi-version caches. These results show that multi-version caches are a viable option for decentralizing the memory system in multiscalar processors. Section 5 presents the conclusions of this study.

2 Multiscalar Processor

2.1 The Processing Model

Figure 1 gives the block diagram of a multiscalar processor. It consists of multiple execution units (EUs) that are connected by a unidirectional ring-type network. This provision, along with hardware head (H) and tail (T) pointers, imposes a sequential order among the units, the head pointer indicating the oldest active unit (when the queue is not empty).

A program executes on the multiscalar processor as follows. Each cycle, if the tail execution unit is idle, a prediction is made by the global control unit (GCU) to determine the next task in the dynamic instruction stream; a task is a subgraph within the control flow graph of the executed program. This predicted task is invoked on the tail unit. Upon a successful invocation, the tail pointer is advanced, and the invocation process continues at the new tail in the next cycle. Each

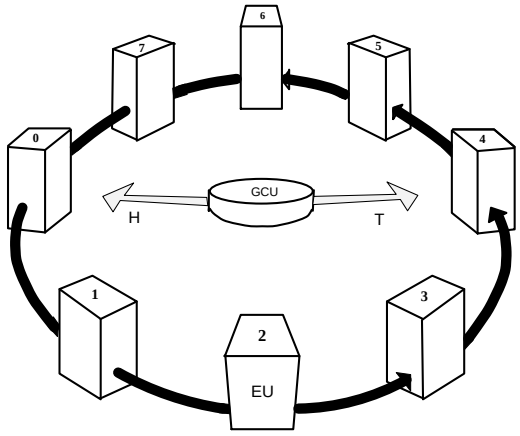

Figure 1: An 8-Unit Multiscalar Processor

active unit executes the instructions in its task, maintaining the appearance of sequential semantics within the tasks and between the tasks. When the head unit completes its task, the task is committed. Upon a successful commit, the head pointer is advanced, causing that unit to become idle. In the event an incorrect execution (due to incorrect task prediction) is detected, all units between the incorrect prediction point and the tail are discarded in what is known as a *squash*. The tail pointer is adjusted to the unit at the point of incorrect prediction, and the invocation process resumes at the correct target. Notice that the tasks being executed in parallel can have both control and data dependencies between them.

References [1] [2] [3] describe different facets of the multiscalar processor in great detail. We shall nevertheless briefly review its memory system here, as it is essential for a good appreciation of the decentralization techniques presented in this paper.

2.2 Memory System

2.2.1 Requirements

The data memory system of the multiscalar processor needs to meet the following requirements: (i) low latency, (ii) high bandwidth, (iii) support for speculative execution of memory references, (iv) dynamic memory disambiguation, and (v) dynamically unresolved memory references. Low latency and high bandwidth are required to support high issue rates. Support for speculative execution involves committing stores in program order (when they are guaranteed to commit), and forwarding uncommitted store values to subsequent (speculatively executed) loads that require them. Support for dynamic memory disambiguation is required, because the multiscalar processor may execute memory references from different execution units out-of-order. Finally, to extract lots of parallelism at run time from a large instruction window, an ILP processor should support dynamically unresolved references [4]. That is, the disambiguation mechanism should allow the execution of memory references before disambiguating them from the preceding stores, or even before the addresses of the preceding stores are known.

2.2.2 ARB

The scheme now used for carrying out data memory communication in the multiscalar processor is called an *address resolution buffer (ARB)*. The basic idea behind the ARB is to allow out-of-order issue and execution of memory references, but provide a hardware platform to order the references sequentially. Memory references from different units are allowed to proceed to the memory system in any order (as and when they are ready to proceed), and the ARB detects out-of-sequence load-store and store-store references to the same memory location as and when they occur. Speculative execution is supported by keeping uncommitted store values in the ARB and forwarding these values to subsequent loads, when they are executed. High bandwidth is provided by means of interleaving the ARB. Figure 2 shows how the ARB banks are connected to the execution units. If multiple requests go to an ARB bank in a cycle, only one of the requests is serviced. Notice also that memory requests from the execution units have to pass through an interconnect (ICN) to get to the ARB banks.

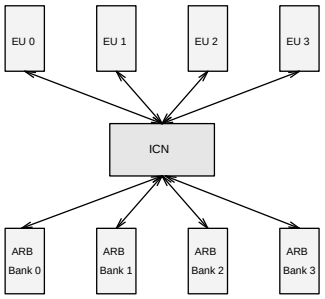

Figure 2: ARB in a Multiscalar Processor

The centralized nature of the memory system in the present multiscalar implementation is primarily because the ARB (and the data cache) are not on the same side of the interconnect as the execution units. The solution that we propose is to bring the address disambiguation system and at least the top level of the memory hierarchy on the same side of the interconnect as the execution units.

3 Multi-Version Caches

3.1 Basic Organization

The hardware structure that we propose for decentralizing memory communication in the multiscalar processor is called a multi-version data cache. The basic idea behind the multi-version data cache is to provide each execution unit with a different version of the architectural memory space, in a separate *local data cache (LDC)*, as shown in Figure 3. Memory communication occurring within a task is carried out by accessing the LDC, and memory communication occurring across tasks is carried out using a *unidirectional ring-type forwarding network*. The advantage of this type of forwarding is that it does not require an expensive crossbar-like connection (as with the ARB originally proposed for the multiscalar processor [2]).

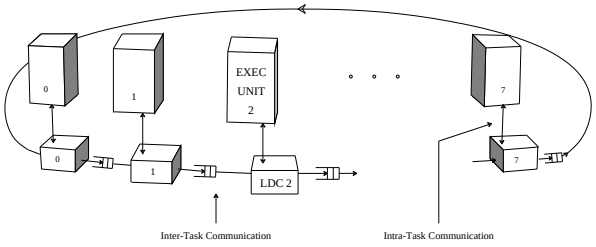

Figure 3: Multi-Version Data Cache for an 8-Unit Multiscalar Processor

Each LDC consists of two caches — called *L1 cache* and *future cache* — organized as a hierarchy, as shown in Figure 4. The L1 cache is an ordinary cache, of reasonable size. The future cache is a special cache, having only a few (typically 16-32) entries. It serves as a hardware platform for storing the speculative memory values produced in its execution unit, and forms the top-most level of the data cache hierarchy. The data array portion of a future cache consists of 3 arrays — *previous value array*, *current value array*, and *subsequent value array* — allowing up to 3 instances for each memory location currently mapped to the future cache. The subsequent sections of the paper describe the purpose and function of these three data arrays.

3.2 Intra-Task Communication

Intra-task memory communication refers to the reading of a store value by load instruction(s) of the same task. All intra-task memory communication (occurring in an execution unit) is carried out using the *current value array* present in the unit's future cache. When a store instruction is executed, the store value is written to the current value array in its future cache.

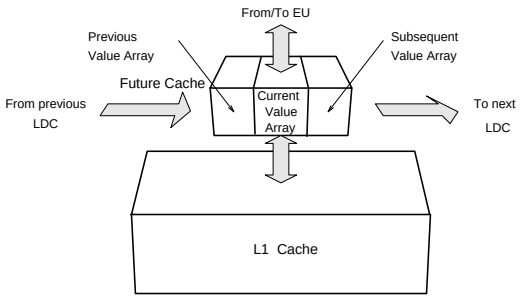

Figure 4: Block Diagram of a Local Data Cache

Subsequent loads from the same task to that memory location take the latest store value (to that location) from the current value array. Because each unit has a separate current value array, intra-task memory communication occurring in different units can all be carried out in parallel.

3.3 Inter-Task Communication

Inter-task memory communication refers to the reading of a store value by load instruction(s) of succeeding task(s). In a distributed environment where multiple versions of a datum can be present, there are 3 options for carrying out inter-task communication. They are: (i) distributed writes and localized reads (*write broadcast*), (ii) localized writes and distributed reads, and (iii) distributed writes and distributed reads. In the first option, write operations update all the relevant non-local data caches, and read operations proceed only to the local data cache. In the second option, write operations update only the local data cache, and read operations pick the correct values from appropriate non-local data caches. The third option is a combination of the first two, in that both the writes and reads are performed over multiple data caches.

3.3.1 Forwarding of Store Values

In the first option discussed above, a load instruction requiring a memory value need access only its local data cache[1], whereas in the other two options, the load has to access the non-local data caches. Because loads are more frequent than stores, we use the first option in the multi-version data cache. Our philosophy is that a memory load operation should not proceed to the other units in search of the (latest) instance it needs, but that the latest instance should be made available to all units that might need it. The way this is achieved is by forwarding from a task (to the subsequent tasks) the memory values produced by stores in

[1] If the load request misses in the LDC, then the request is sent to the common, second-level cache.

that task, as and when these values are produced. The forwarding of memory values is carried out with the help of the ring-type forwarding network, which forwards values from one execution unit to the successor unit only, as shown in Figure 3.

3.3.2 Function of Previous and Subsequent Value Arrays

When memory instances produced by preceding tasks (in the preceding units) arrive at the LDC of an execution unit E, they are stored in the *previous value array* of E's future cache. The previous value array therefore is the hardware platform for storing the latest memory instances produced in the preceding tasks. The only reason for using separate data arrays for intra-task communication and inter-task communication is to ease recovery actions at times of incorrect task predictions.

Next consider the memory values produced by subsequent tasks (in the subsequent units) of the active instruction window. If these values are allowed to propagate beyond the tail unit and then onto the head unit and into the active window, they will defile the previous value arrays of the active execution units with incorrect values. This will not only result in incorrect execution, but also make it impossible to recover in times of incorrect task prediction. If, on the other hand, these values are stopped before the head unit, then each time a finished task is committed, memory values have to be copied to the head unit's previous value array from the previous unit. The rate with which the memory values are copied will decide the rate with which new tasks can be initiated. To avoid the significant overhead of continually copying memory values across (adjacent) execution units, a separate data array, called the *subsequent value array*, is used as the hardware platform for storing the memory values produced in the succeeding units of the active window. Thus, the subsequent value array of an active execution unit maintains the "future" memory instances that are potentially required for the execution of subsequent tasks that will be allocated to that unit in the future.

It is important to note that it is not a necessity to have three data arrays per future cache. We saw that the subsequent value array is primarily introduced for performance reasons. Similarly, the previous value array can be merged with the current value array of the same unit. However, recovery actions (in times of incorrect task prediction) become more complicated. Thus, the previous value array is for fast recovery (*i.e.*, going back into the past), whereas the subsequent value array is for fast progress (*i.e.*, going forward into the future).

3.4 Working

3.4.1 Execution of a Load

When a load is executed from an execution unit, first the future cache is checked to see if the load address is already present in it. If the address is present, then the corresponding *current value* is sent to the destination register of the load (if the current value is valid). If the current value is not valid, then the *previous value* is sent to the destination register of the load (if the previous value is valid). If both the current value and the previous value of the address are not valid, or if the address is not present in the future cache, then the load request is sent to the L1 cache of the same unit[2], which supplies the value (possibly by getting it from the common, L2 cache). When the L1 cache supplies the value to the future cache, a copy of the value is kept as the previous value in the relevant future cache entry. Furthermore, the state information of that future cache entry is updated to reflect the fact that a load has been performed by the current task to that address.

3.4.2 Execution of a Store

When a store is executed from an execution unit, a similar check is performed within the future cache to see if the store address is already present. (If no entry is found, a free entry is allotted to the new address.) A copy of the store value is stored as the *current value* of that address. The store value and other particulars of the store are also entered into the forwarding queue of that execution unit, so as to forward the values to the subsequent units.

3.4.3 Detecting Inter-Task Memory Dependency Violations

When a store value that is forwarded through the forwarding network arrives at an execution unit, first of all, the future cache of the unit is checked to see if the store address is already present. If the address is not present, then no memory dependency violation has occurred; all that needs to be done is to allocate an entry to that address, and to store a copy of the store value as the *previous value* of that address. If the address is already present, then a check is made to see if a load has been performed to that address from this execution unit, and if the load has used a different value. If a memory dependency violation is detected, then recovery action is initiated. Such recovery action

[2] It is worthwhile to point out that in order for the future cache to be not in the critical path when a load "misses" in the future cache, load requests can be sent to both the future cache and the L1 cache simultaneously; if a value is obtained from the future cache, the value obtained from the L1 cache can be discarded.

might involve, for instance, squashing the execution of the tasks in this execution unit and the subsequent units.

3.4.4 Committing an Execution Unit

When the execution unit at the head is committed, the unit is no longer a part of the active multiscalar window. The following actions are performed as part of the committing process. For each memory location currently mapped to its future cache, among the previous, current, and subsequent instances, the latest one is renamed as the new previous instance. The latest instance among the three is identified as follows: if a subsequent instance is present, then it is the latest; else if a current instance is present, then it is the latest; else the previous instance is the latest. An important aspect to be noted here is that to rename the instances, data needs to be moved only logically. This logical move is easily implemented by updating the state information associated with the three instances of a memory location in the execution unit. Thus, the future cache's previous array, current array, and subsequent array are renamed on a per-location basis as in the multi-version register file of the multiscalar processor [2].

3.4.5 Rollback

Any processor performing speculative execution has to be prepared to rollback its execution when incorrect speculation is detected; the multiscalar processor is no exception to this rule. When recovery action is initiated in the multiscalar processor due to events such as incorrect task prediction and incorrect dynamically unresolved references, special actions are taken to make the multi-version cache state in line with the recovery actions. These special recovery involves 2 actions: (i) clearing the load marks in the future cache of all squashed units, and (ii) purging from all future caches the memory instances produced by the squashed tasks.

3.4.6 Handling Contentions in Future Cache

One last thing that we need to address is contentions in the future cache. That is, when a memory instance is to be written to the future cache, what if there is no free entry to enter that information? This situation also warrants special recovery actions. If the value to be entered is a *subsequent value*, then instruction execution and memory value forwarding in the subsequent units are stalled until an entry becomes available to place this subsequent value.

If the value to be entered is produced in the same execution unit, then the future cache is checked to see if there are any address entries with only subsequent values. If any such entry is found, one of those entries is selected, and the execution units including and beyond the unit that produced this subsequent value are squashed. This squashing action will enable that entry to be reclaimed, and be subsequently allotted to the memory instance that is to be entered. If no entry is found (in the future cache) with only subsequent values, then instruction execution and memory value forwarding in this unit and the subsequent units are stalled until an entry becomes available to place this current value.

If the value to be entered in the future cache is produced by a previous unit, then also the future cache is checked to see if there are any address entries with only subsequent values. If any such entry is found, one of those entries is selected, and the execution units including and beyond the unit that produced this subsequent value are squashed. If no entry is found (in the future cache) with only subsequent values, then squashing this execution unit and subsequent units will reclaim at least one entry.

3.5 Comparison with Multiprocessor Caches

It is worthwhile to do a brief comparative study of the multi-version caches in the multiscalar processor with the multiple caches used in shared-memory multiprocessors. In both cases, each PE has its own local data cache, and has a direct connection only to its local data cache. Furthermore, both cases have to maintain consistency amongst the multiple data caches — *i.e.*, a load to a memory location should fetch the "latest" update to that memory location. However, the notion of the "latest" instance of a memory location is somewhat different in the two cases, primarily because of the distinction between the two processing paradigms. In a multiprocessor, the "latest" instance depends on the type of consistency model implemented[3], such as sequential consistency (total order consistent with program order), processor consistency, weak ordering, and release consistency. The mechanism for maintaining coherency amongst the multiple LDCs in a multiprocessor will also depend on the consistency model, and the type of interconnection between the multiple LDCs (e.g. bus, hypercube, omega network, etc). In the multiscalar processor, the "latest" instance is the one that adheres to sequential consistency, and the way coherency is maintained amongst the multiple LDCs is by way of a unidirectional ring-type connection that forwards store values from each unit to the subsequent unit.

[3] This is because, the PEs in a multiprocessor are executing in parallel codes that are both control independent and data independent. Data dependencies, if any, between these codes are enforced by the use of special synchronization variables, and not through ordinary memory locations.

4 Experiments and Results

4.1 Experimental Framework

Our methodology of experimentation is simulation, and the experimental framework is same as the one used in [2]. The L1 caches are of 4K words, with 1 word blocks. The L2 cache is of 32K words, with 2-word blocks. Both the L1 and L2 caches are direct-mapped. Rest of the parameters (such as number of instructions simulated) are same as those in [2].

Benchmark programs were compiled for a DECstation 3100 using the MIPS C compiler. For measuring performance, we use instruction completion rate as the metric, with the following steps adopted to make the metric as reliable as possible: (i) nops are not counted while calculating the instruction completion rate, and (ii) speculatively executed instructions whose execution was not required are not counted while calculating the instruction completion rate, and Thus, our metric can be called as *useful instruction completion rate (UICR).*

4.2 Experimental Results

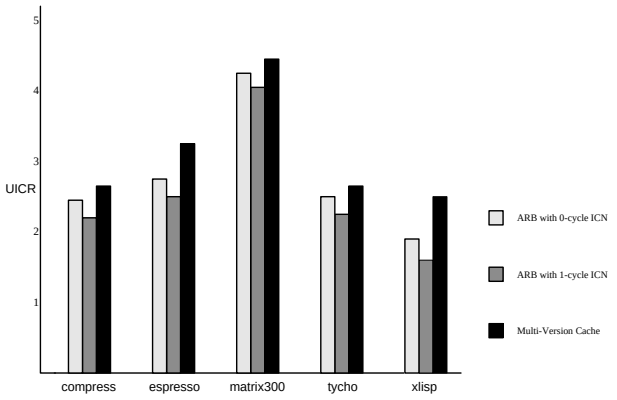

Figure 5: UICR with 8 Execution Units

Figure 5 presents the results obtained with an 8-unit multiscalar processor. The X-axis denotes the benchmarks, and the Y-axis denotes the useful instruction completion rate (UICR). The lightly shaded portions of the figure correspond to the results for a multiscalar processor with an ARB and an idealized 0-cycle interconnection network (ICN) connecting the execution units and the ARB. The next dark shade corresponds to the results with the the ARB and a more realistic, 1-cycle ICN. The darkly shaded region corresponds to the results with the multi-version data cache. On inspecting the figure, we can see that the multi-version data cache scheme performs not only as good as the ARB schemes, but even better than them, for all the benchmarks. This is because in the ARB scheme, many ARB cycles are lost because of multiple memory requests (from different execution units) attempting to access the same ARB bank (although they are to different memory addresses) in a cycle. In the case of the multi-version data cache, these requests (from different execution units) go to their local data caches only, and so no bank conflicts occur.

5 Conclusions

We have developed a scheme for decentralizing the memory system in multiscalar processors, and our strategy is to use a separate local data cache (LDC) for each execution unit in the processor. These data caches can hold different versions of a memory location, and can be temporarily incoherent. The temporary incoherency is corrected by forwarding the store values from each LDC to the subsequent LDCs through the unidirectional ring-type network. We also evaluated the performance of multi-version caches using a simulation study. Based on the results obtained, we believe that multi-version caches are an excellent choice for use in the multiscalar processor.

Acknowledgements

This work was supported by the NSF Research Initiation Award Grant CCR 9410706.

References

[1] S. E. Breach, T. N. Vijaykumar, G. S. Sohi, "The Anatomy of the Register File in a Multiscalar Processor," *Proc. 27th Annual International Symposium on Microarchitecture (MICRO-27)*, 1994.

[2] M. Franklin, "The Multiscalar Architecture," *Ph.D. Thesis, Technical Report TR 1196*, Computer Sciences Department, University of Wisconsin-Madison, 1993.

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

[4] M. Franklin and G. S. Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References," *(To appear in) IEEE Transactions on Computers.*

[5] P. P. Gelsinger, P. A. Gargini, G. H. Parker, and A. Y. C. Yu, "Microprocessors circa 2000," *IEEE Spectrum*, vol. 26, no. 10, pp. 43-47, October 1989.