Improving I/O Performance with a Conditional Store Buffer

Lambert Schaelicke and Al Davis
Computer Science Department
University of Utah
Salt Lake City, UT 84112
lambert@cs.utah.edu ald@cs.utah.edu

Abstract

Microprocessor I/O performance is becoming increasingly critical in order to support efficient communication interfaces as modern microprocessors continue to be used in a variety of multiprocessor configurations. Numerous performance enhancements have been made to improve processor performance by improving the latency and bandwidth to main memory or creating efficient mechanisms to hide main memory latency. These include speculative out of order instruction execution, lock-up free caches, and improved memory bus designs. Sadly these improvements are not directly applicable to improved I/O system performance and may even complicate high performance I/O system design. This paper introduces and analyzes the design of a simple mechanism called the **conditional store buffer**. *The conditional store buffer improves I/O write performance by making better use of the system bus to increase effective I/O bandwidth, while greatly reducing synchronization overhead. The cost is a minor increase in hardware complexity.*

1 Introduction

Data transfers in computing systems can be roughly split into two categories: memory transfers and I/O transfers. Both types of transactions utilize a common system bus. A great deal of research and subsequent progress has focused on increasing the effective bandwidth and reducing the latency of memory transfers since these metrics have a significant impact on uniprocessor program performance. Meanwhile, I/O performance has received comparatively minor levels of attention from processor designers. This lack of attention creates a significant I/O bottleneck as interest moves towards supporting efficient fine grain multiprocessor communication in clusters and networks of workstations. The problem is exacerbated by system bus occupancy and synchronization overheads [5][10] when the nodes of these systems are themselves shared memory symmetric multiprocessors.

Many of the optimizations used to reduce or hide main memory latency cannot be directly applied to I/O transfers. The main problem is that I/O transfers potentially have side-effects, such as initiating a DMA transfer, clearing status bits, or removing data from a receive queue. This forces I/O transfers to be done in strict program order, non-speculatively, and exactly once. The result is that I/O stores usually bypass the cache hierarchy and every store directly initiates a bus transactions. This wastes precious system bus bandwidth since the bus has been optimized for cache line sized transfers, significantly limits effective I/O bandwidth, and incurs high levels of synchronization overhead. The purpose of this paper is to provide a simple and cheap solution to this problem called the ***conditional store buffer*** or CSB.

The key feature of the CSB is that it is software controlled, uncached, and allows multiple I/O stores to be combined. The result is that I/O performance can be significantly improved for a minor increase in hardware complexity. System bus designs are optimized for main memory accesses and therefore provide maximum effective bandwidth for cache line sized transactions. The CSB design allows I/O stores to be combined into a single bus transaction up to a maximum size of one cache line. This permits I/O performance to approach that of main memory bandwidth and removes the need for costly synchronization operations in order to access I/O devices. The CSB also permits user-level code to explicitly control *which* stores will be combined and *when* the combined set of stores will be issued on the system bus. This guarantees that the sequence of combined store instructions is atomic and provides the necessary *exactly once* semantics for the resulting bus transaction.

Figure 1 shows a block diagram of a conventional system with a 2-level cache hierarchy and a typical uncached load and store capability that has been enhanced with the addition of a CSB. Conventional uncached loads and stores can be handled in the normal manner [8], while the combining store instructions are handled by the CSB. The remainder of the paper discusses the motivations for such an architecture in more detail, describes the design of the CSB, and analyzes the performance potential via simulation results. The paper concludes with a qualitative analysis, a discussion of related work that influenced the CSB design, and a short summary section..

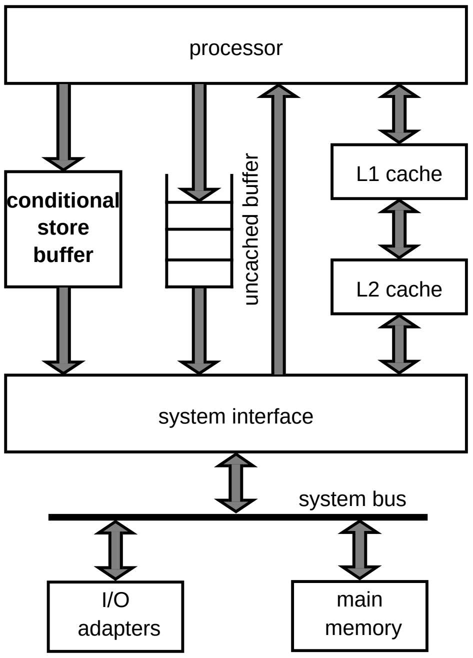

Figure 1: Architectural Model

2 Motivation

The recent interest in clusters of workstations creates increased pressure on I/O system performance. The speedup of a parallel application is ultimately limited by the per-message overhead. A Berkeley NOW project study [7] shows that program performance is more sensitive to communication overhead than to latency. Another study by Mukkerjee and Hill [10] found that the message data transfer between the memory and the network interface accounts for 20% to 42% of the total execution time for a set of parallel scientific applications. The average message size for these applications ranges from 19 to 230 bytes. Due to the overhead involved in setting up a DMA transfer between the network interface and main memory, short messages are more efficiently transferred using programmed I/O (PIO). Simulation results for a sender-based message passing protocol [3] show that PIO is better than DMA for messages shorter than 128 bytes.

Since I/O operations require an *in-order exactly-once* semantics, transactions that target I/O devices can not be reordered or executed multiple times. The result is that I/O operations usually bypass the cache hierarchy. However, issuing a bus transaction for each individual uncached load or store instruction makes poor use of the system bus. The result is increased bus occupancy, control overhead and latency, resulting in lower bandwidth I/O transactions. To alleviate this problem, some processor implementations have introduced combining store buffers that try to reduce the number of bus transactions. For instance, the PowerPC 620 combines up to two uncached stores of the same size to consecutive addresses into a single bus transaction. The accelerated store buffer in the R10000 [8] detects sequential access patterns and combines subsequent stores into a complete cache line if possible.

These hardware controlled schemes fail if the sequence of stores is interrupted by a store to a different address. In this case, the system interface must issue multiple single-beat bus transactions. These combining store buffers are therefore software-transparent optimizations and do not provide precise control over the resulting bus transactions. Furthermore, combining uncached stores involves a trade-off between latency and bus utilization. To reduce I/O latency, the store operation should be issued on the system bus as soon as possible. On the other hand, combining is more successful if transactions remain in the uncached buffer for a long time. These trade-offs are different from those faced by write buffers that are part of the cache hierarchy [12], where pending stores are kept coherent with successive loads or other caches, and store latency to memory is only a secondary concern.

Several I/O adapter designs have achieved low latency by taking advantage of the fact that individual bus transactions are atomic. For instance, the Atoll network interface [2] uses a single store instruction to initiate a DMA operation, where both the source address and the message length are packed into a single 64-bit word. The HP Medusa network adapter [1] maintains kernel packet buffer (mbuf) descriptors in on-board hardware FIFOs. A single 32-bit store instruction can then be used to push a new descriptor onto the transmit FIFO to initiate a packet transfer. The primary advantage of these schemes is that they don’t require costly locking overhead in order to access the device. However, in many cases the amount of data that needs to be transferred atomically exceeds the size of a single uncached bus transaction. Unfortunately, current combining store buffers are software transparent. Hence they do not solve this dilemma since the software cannot guarantee that certain transactions will be combined.

The CSB combines the advantages of higher bandwidth through the use of burst transactions with the ability to explicitly combine several uncached stores into an atomic transaction under software control. It introduces the notion of a combining store instruction to communicate to the CSB and control which stores are to be combined. A dedicated flush instruction at the end of the store sequence triggers the atomic bus transaction.

A difficulty occurs when multiple processes compete for access to the CSB. To avoid deadlock problems while achieving low latency device access, the CSB design uses an optimistic non-blocking synchronization scheme similar to the load-linked/store-conditional instruction pair functionality of the MIPS architecture.

3 The Conditional Store Buffer Design

3.1 Instruction Set Architecture Modifications

The CSB design requires two additional architectural modifications. First, software must be able to specify which store instructions should be combined. The obvious choice is to introduce a new instruction *store combine*. However, adding new instructions to an existing architecture should not be taken lightly. We therefore use existing memory mapping hardware to indicate which addresses should be combined. Several architectures already encode cache policies and other memory attributes in page table entries. The PowerPC allows the specification of write-through or write-back caching, along with other attributes, on a per-page basis. In the R10000, the accelerated uncached buffer is enabled by a bit in the page table entry. Hence, the encoding of one additional attribute is a minor extension to existing TLB designs.

The second modification involves the addition of a conditional-flush capability. The purpose of this instruction is twofold. It has to trigger the flushing of the CSB (if no conflict was detected), and signal the success of the flush to the program. Our design uses the SPARC atomic swap instruction, with a slight semantic variation if the destination address is in uncached combining address space.

The implementation of non-blocking synchronization also requires that the current process ID be available to the CSB at runtime, in order to detect conflicting accesses by competing processes. Several architectures store the process ID in a supervisor mode register to detect aliasing of cache or TLB entries. For instance, MIPS defines an 8-bit space identifier that helps to avoid flushing the TLB on every context switch; PA-RISC uses an 18 bit address space identifier to de-alias references to virtually addressed caches; and the Alpha 21164 stores a 7 bit process ID in a privileged register. This indicates that the process ID can be communicated to the CSB without extensive hardware modification.

3.2 Conditional Store Buffer Implementation

Figure 2 shows the structure of the conditional store buffer. The data buffer provides space for one cache line worth of data, the process ID, and the cache line aligned address of the most recent combining store. The hit counter is used to implement the non-blocking conditional flush operation. It counts the number of consecutive stores that have been issued by a process without conflict.

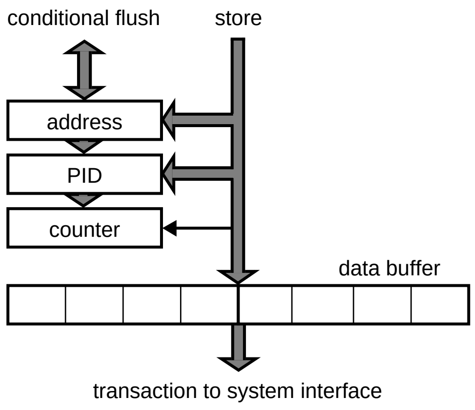

Figure 2: Conditional Store Buffer

When the buffer receives a combining store, it compares the destination address and the current process ID with the values that have been saved from the previous store instruction. On a match, it stores the data in the appropriate slot and increments the hit counter by one. If the comparison fails: the buffer is cleared, the hit counter is reset to 1, and the new data is stored. Note that combining stores can be issued in any order, since only the total number of stores is needed for conflict detection.

The conditional flush instruction communicates the expected value of the hit counter. If the counter value is equal to the value provided by the instruction, and the destination address and process ID match the value present in the CSB: the data is sent to the system interface as a single burst transaction, and the buffer and hit counter are cleared. If any of these conditions fails: the data register is cleared, the counter is reset to zero, and nothing is issued to the system interface.

Rather than introducing a new instruction, our implementation uses the SPARC swap instruction for the conditional flush. If the destination address is in uncached combining address space, the instruction is sent to the CSB. The conditional flush instruction leaves the value of its source register unchanged if the flush succeeded, otherwise it returns 0. The destination register value should then be compared with the expected value to determine if the entire sequence was issued as an atomic transaction. Software is responsible for recovery after a failed flush, usually by branching back to the beginning of the sequence of combining stores.

The following SPARC assembler code segment shows an example of how software might access the CSB.

```
.RETRY:
    set 8, %l4          ! expected value
    ! store 8 dwords in any order
    std %f0,[%o1]
    std %f10,[%o1+40]
    ! ... 5 additional dword stores
    std %f12,[%o1+8]
    swap [%o1], %l4     ! conditional flush
    cmp %l4, 8          ! compare values
    bnz .RETRY          ! retry on failure
```

Suppose, for example, that this process is interrupted before it executed the conditional flush instruction. The first combining store of the competing process will clear the buffer and reset the hit counter to 1. When the original process attempts to flush the buffer, its process ID and/or the expected counter value will not match the values stored in the buffer and the conditional flush instruction will return a 0 to signal the conflict.

This non-blocking policy removes the need to lock the CSB prior to access and competing processes do not block on a conflict [6]. The policy is optimistic in its assumption that conflicts are rare and it is more cost effective to replace heavyweight synchronization on every sequence with a software recovery mechanism on a failed attempt. Since lock-free synchronization schemes do not prevent competing processes from accessing a resource, they do not lead to problems like priority inversion or the difficulty of deadlock avoidance.

Theoretically, it is possible for two processes to be scheduled such that each continuously conflicts with the other. There are numerous simple solutions for this livelock scenario. One can limit the number of failed conditional flushes, or use an exponential backoff algorithm to reduce the likelihood of a conflict. It should be noted that uncached loads bypass the combined stores. This is reasonable because the combined stores have not yet been committed by a conditional flush.

The size of the data register has been chosen to be a single cache line, since the system interface and system bus are already optimized to handle cache line sized burst transfers. Since most system buses do not allow arbitrary-length bursts, the CSB model in this study always issues a full cache line, regardless of the number of combining store instructions. This restriction could be relaxed in a CSB design for a particular bus which permits multiple burst sizes. Unused words are padded with zeroes since the buffer will be cleared when the first conditional store arrives, thus avoiding subtle security issues.

The single line buffer described here could be easily extended with a second line buffer to increase pipelining and avoid program stalls awaiting the completion of the conditional flush instruction. After a conditional flush, the CSB would switch to the second data register while delivering the first one to the system interface. The same effect can be achieved when the system interface provides additional buffering.

Note that it is not strictly necessary to include the destination address in the conflict check. However, this allows detection of conflicts between competing threads that might run under the same process ID.

3.3 System Implications

It should be noted that the performance improvement that is made possible by the CSB also depends on the ability of the target I/O device to accept burst writes. In general, this increases the complexity of an I/O device. On the other hand, the potential performance gain (presented in section 4.3) more than justifies the increased cost. Note also that many modern I/O adapters already provide this capability.

4 Quantitative Evaluation

4.1 Architectural Model

The potential performance benefits of the CSB are evaluated through execution driven simulation of a series of microbenchmarks. The simulator is based on RSIM [11]. Originally, RSIM was developed to study distributed shared memory architectures. The interesting feature for this effort is the fairly detailed dynamically scheduled processor model. The processor model uses a unified dispatch queue that keeps track of true data dependencies and structural hazards. Instructions are issued out-of-order as soon as their operands are available and results are committed in order to facilitate precise interrupt handling. The microarchitecture is similar to the R10000 or PA8000, but is slightly more aggressive. RSIM executes SPARC V9 binaries.

The simulator has been extended to support an uncached address space. Loads and stores to this space are issued to the system interface strictly in program order and are non-speculative. Data values are not forwarded from a store to a subsequent load, since the load might have a side effect and therefore must be issued to the system bus.

In our experiments, the processor model is configured to dispatch and retire a maximum of four instructions per cycle. Instructions may issue up to two integer units and two floating point units simultaneously. Memory operations are handled in a separate queue, which speculatively performs address calculations and executes cached loads. Uncached operations are issued non-speculatively, at or after the time they are retired from the reorder buffer.

Regular uncached loads and stores are handled in an uncached buffer. In its simplest form this is a queue that buffers loads and stores until they can be sent to the system interface. To model more aggressive combining schemes, the uncached buffer can be configured to combine stores whenever possible. A store may be coalesced into an existing entry if its destination address falls into the same block and it does not bypass an earlier load or barrier instruction. The size of a buffer entry, and hence the maximum number of combined stores, will be varied from two words (16 bytes) to a 64 byte cache line. This covers the wide variety of uncached store policies found in current processors, ranging from non-combining to combining of a full cache line.

Entries in the uncached buffer are processed in FIFO order. Combining is limited by the time that an entry spends waiting in the buffer. Memory barrier instructions are prevented from graduating until the uncached buffer is empty.

The conditional store buffer follows the design outlined in the previous section. A part of the uncached address space is designated as combining. Stores to these addresses are combined in the CSB until they are committed. The CSB has only one entry, hence stores following a flush may stall until the entry has been sent to the system interface.

Results are collected using two different system bus models: *multiplexed* and *split*. Both buses are fully pipelined, and arbitration is overlapped with the current transaction. On the multiplexed bus, an address transfer takes one extra cycle, while the split bus has separate address and data paths. The width of the bus, as well as the clock frequency ratio between processor and bus can be varied. Unless otherwise noted, the simulations assume that no idle cycle is necessary between transactions that are driven by the same master, hence consecutive stores can be issued back to back. It is also assumed that the system bus supports transfer sizes ranging from 1 byte to a complete cache line in powers of two. All transactions must be naturally aligned, which restricts the ability to combine stores.

4.2 Microbenchmarks

Two microbenchmarks are used to assess the performance of a conditional store buffer. Uncached store bandwidth is measured using a tight loop of doubleword stores. The loop is unrolled so that in each iteration a complete cache line worth of data is stored. The purpose of this experiment is to put as much pressure as possible on the system bus and compare the effective bandwidths of various combining schemes and the CSB. The second microbenchmark is intended to evaluate the overhead of locking and unlocking versus an atomic access through the CSB. The lock-acquire operation is implemented using the SPARC swap instruction inside a loop body, which is repeated until the lock has been set successfully. After lock acquisition, a sequence of two to eight doubleword stores to uncached accelerated space is executed. A memory barrier instruction ensures that the lock release operation is executed only after the last uncached bus transaction has left the uncached buffer.

4.3 Results

4.3.1 Store Bandwidth

Multiplexed Bus

Figure 3 shows the effective store bandwidth for various combinations of block sizes, processor to bus frequency ratios, and bus overheads using an 8-byte wide multiplexed system bus. Each graph illustrates the number of bytes transferred per bus cycle (y-axis) vs. the transfer size (x-axis). Each group of vertical bars shows results for the non-combining scheme on the left to full cache line combining, followed by the conditional store buffer result on the right. The total amount of data transferred is varied between 16 bytes (2 doubleword stores) and 1 Kbyte. The bus is assumed to be completely idle, except for the uncached data transfers.

The results are intuitive and not particularly surprising. However, they indicate that important trends in processor and bus design can penalize uncached bus transactions, and how much benefit burst transactions have over single-beat transactions.

In general, the effective bandwidth increases for larger data transfers because combining in the uncached buffer is only effective if the buffer is not empty. Initially, the system interface is idle and the first store instructions can immediately be forwarded. Only when the buffer begins to fill up, can new stores be coalesced into existing entries. Regardless of the combining scheme, the first few transactions transfer only one or two doublewords each.

Figures 3 (a) to (c) show the impact of the processor to bus frequency ratio on store bandwidth for a range of current and probable design points. The result is that, without any combining, the bandwidth is independent of the total amount of data transferred. Each store results in a two-cycle bus transaction, thus the effective bus

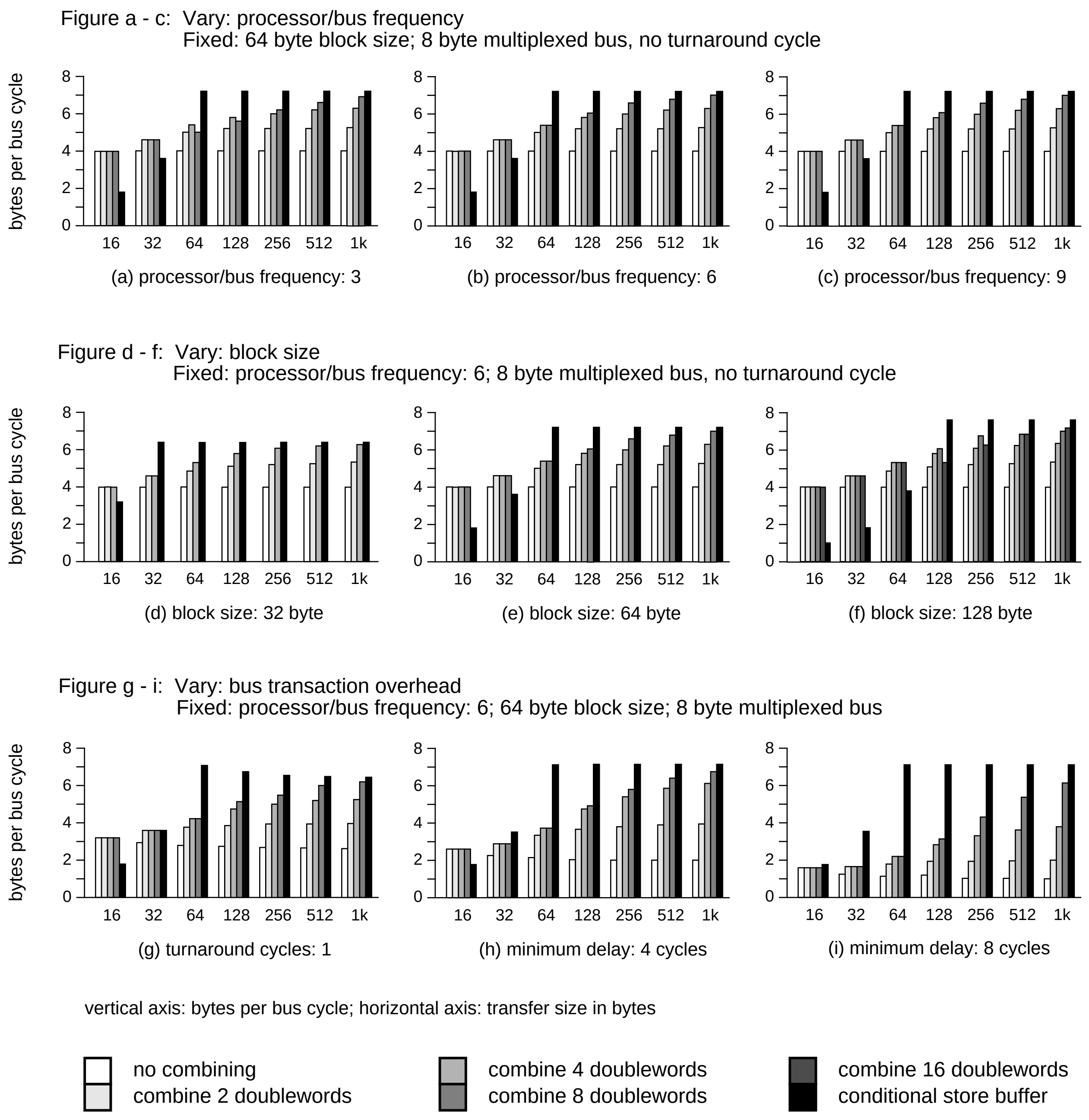

Figure 3: Uncached Store Bandwidth on a Multiplexed Bus

bandwidth is 4 bytes per bus cycle, which is half of the peak bandwidth. For small data transfers of 16 bytes, combining has no effect because the first store leaves the buffer before the second is issued. Larger data transfers benefit increasingly from combining, ultimately approaching the peak bandwidth of one cache line per 5 cycles.

The conditional store buffer clearly has the greatest advantage over all other schemes for transfer sizes of about a cache line. On the other hand, transfers that are significantly smaller than a cache line are penalized by the unnecessary long burst transactions. This penalty is easily reduced if multiple burst size transfers are supported by a particular bus design. The pressure on the uncached buffer increases with higher processor frequency, leading to better combining in the beginning of the data stream. The resulting increase in store bandwidth is relatively insignificant.

The subsequent experiments are based on a processor to bus frequency ratio of 6. This ratio is considered representative of near future microprocessors with core frequencies approaching 1 GHz and bus frequencies at exceeding 100 MHz. The results in figures 3 (d) to (f) show the impact of varying the cache line size between 32 bytes and 128 bytes. For larger cache lines, the relative performance advantage of the conditional store buffer increases because the per-transaction overhead is reduced by the longer burst. Another interesting detail is that for medium sized transfers, a smaller combining buffer can be more efficient, due to the alignment restrictions of the bus (see figures 3 (a) and (f)). Intuitively, it is sometimes better to issue a smaller transfer early, the remaining stores can then coalesce with the following instructions to form a transaction that is larger than otherwise possible. It is also obvious that increasing the cache line size pushes the crossover point between the CSB and other schemes towards larger transfers, because the conditional store buffer always issues burst transactions.

Figures 3 (g) to (i) show the effect of increasing the bus overhead. In figure 3 (g) a turnaround cycle is inserted after every transaction. This may be necessary for certain bus designs that always require an idle cycle between transactions, even when those are driven by the same master. It can also be viewed as an approximation of a heavily loaded bus with multiple masters, the effective bandwidth indicates how well a master can utilize its share of the raw bus bandwidth. For larger transfers, bandwidth decreases for the non-combining scheme and the CSB because the turnaround cycle that follows the last transaction is not included in the bandwidth calculation. The transfer is considered complete at the end of the last transaction. For instance, a doubleword transaction takes 2 cycles, two consecutive transactions take 5 cycles, three transactions take 8 cycles, and so on. This effect diminishes for large transfers and the effective bandwidth approaches the peak value. The net effect is that the CSB bandwidth surpasses all other schemes for even shorter transfers, relative to buses without a turnaround cycle.

Some buses [9] implement a selective flow control scheme. For each address transfer, the target acknowledges the transaction, indicating if it is able to process the request. The acknowledgment is usually expected a fixed number of cycles after the address cycle. This makes deadlock avoidance somewhat easier, since targets can selectively reject certain transactions while possibly accepting others. Other targets remain unaffected. However, it also introduces the possibility that transactions are reordered in the system interface. Memory transactions and acknowledgments can be easily pipelined, because transaction reordering does not create problems in modern weakly ordered memory models. However, if the system interface has to maintain strong ordering for uncached accesses, it can not issue the next uncached store before the previous one has been positively acknowledged.

Figures 3 (h) and (i) show the effect of such a minimum delay requirement. A delay of 4 cycles means that the address cycle of transactions must be 4 cycles apart. It is evident that only short transactions are affected by this, while an 8-cycle burst completely overlaps with the acknowledgment. Burst transfers yield higher bandwidth for all but the very shortest transfer (Figure 3 (h)). The high acknowledgment latency of 8 cycles penalizes short transactions. Although currently unrealistic, future high-frequency bus designs may well incur a similar penalty.

Split Address/Data Bus

Figure 4 shows results for a split bus with separate address and data paths. Examples of such systems are the Sun UPA used in UltraSPARC based systems, or the system buses of various PowerPC processors. Commonly, the data path is wider than in a multiplexed bus, ranging from 128 to 256 bits. The fact that the data path is wider than a processor word introduces a different kind of overhead, namely wasted bus width. For instance, a doubleword transaction uses only half of the bandwidth of a 128 bit wide bus.

Figures 4 (a) and (b) show this effect for a 128 bit and a 256 bit wide bus (Note the vertical axis scale difference). Data cycles can issue back-to-back without a turnaround cycle. The only overhead in this case is the wasted bus bandwidth for transfers smaller than the bus width. In particular, on a 256 bit wide bus, a burst transfer takes only two cycles, the same number of cycles as two individual doubleword stores.

Figure 4 (c) shows the impact of a mandatory turnaround cycle between transactions on a 128 bit wide bus. The effect is similar to the one found for the multiplexed bus, but the performance gap between the non-combining scheme and the CSB is greater, because of the additional overhead of wasted bus width. Introducing a minimum delay between transactions has a more dramatic impact, because transactions of the same size take only half as many cycles as on a narrower bus. For a minimum delay of 4, only the CSB can successfully hide the acknowledgment latency, while a longer delay affects all transactions. Note that this effect is dramatic only because transactions and acknowledgments can not be pipelined for strongly ordered I/O accesses.

Discussion

The bandwidth results presented in this section are not specific to the CSB design. Rather, they quantify the

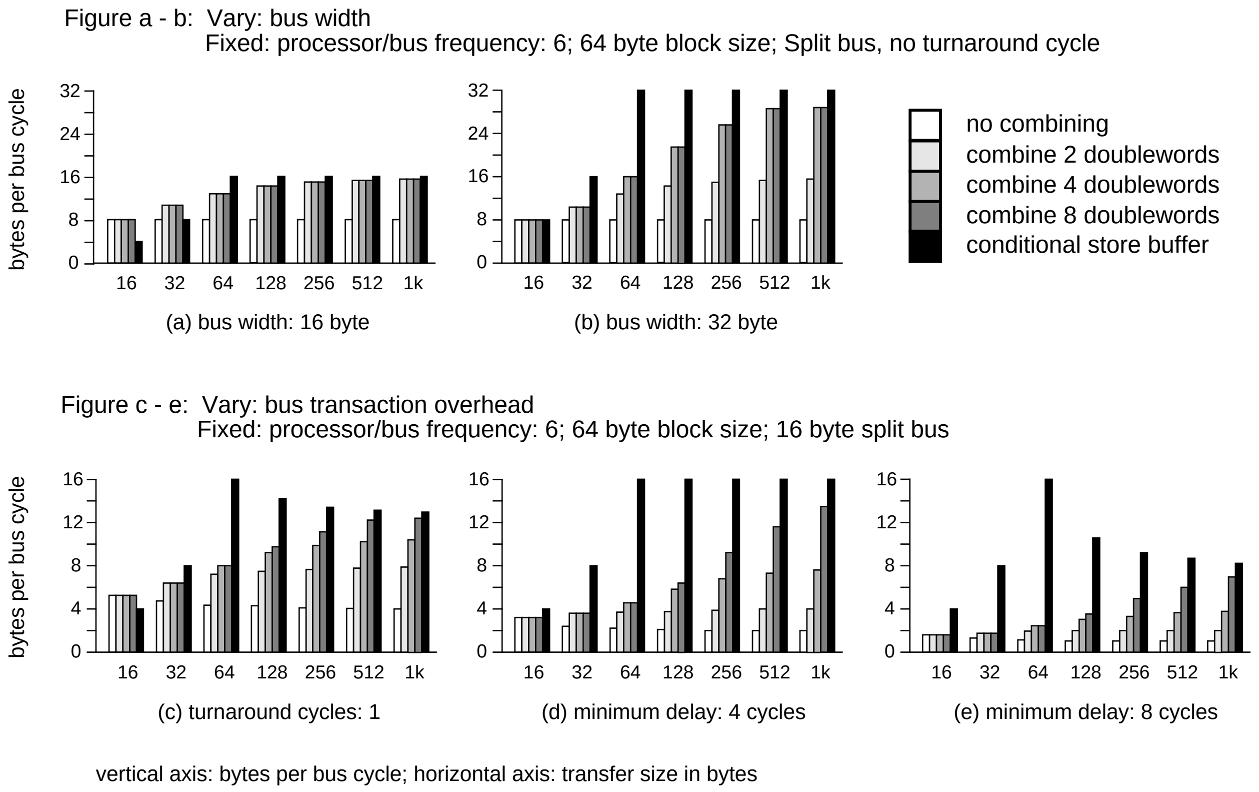

Figure 4: Uncached Store Bandwidth on a Split Address/Data Bus

intuitive importance of burst transfers to achieve high bus utilization and show the impact of several trends. The operating frequency of processors increases faster than the bus frequency, thus putting more pressure on the memory hierarchy and bus system. This increased frequency ratio makes uncached combining buffers more effective. Other trends, like increased bus width and larger cache lines have the opposite effect of penalizing sub-block transactions. Selective flow control is an example of how bus systems are optimized for burst transactions to memory.

The conditional store buffer makes more effective use of the system bus by generating burst transaction even for small data transfers, which leads to more predictable performance of uncached stores.

4.3.2 Atomic I/O Access

Figure 5 compares the number of processor cycles for a conventional lock-access-unlock sequence under various combining schemes with the uncached store buffer. Figure 5 (a) illustrates the performance if the lock access hits in the L1 cache. Two main effects are responsible for the significant advantage of the CSB. The net overhead of locking and unlocking is 8 cycles even when the lock access hits in the L1 cache, and 137 cycles for a miss. The cache miss latency is 100 cycles, which corresponds to 166 ns on a 600 MHz processor. The lock acquire and release operation consists of 8 and 3 instructions respectively. They include setting up the lock address in a register, initializing the swap destination register and checking the result of the atomic swap. Memory barrier instructions separate these operations from the uncached stores.

For the locking scheme without combining, latency ranges from 28 to 100 cycles for transfers of 2 to 8 doublewords. It increases by 12 cycles for every doubleword transferred, because the lock can only be released after the last uncached store has left the uncached buffer. On the other hand, an access through the conditional store buffer can be considered complete as soon as the conditional flush instruction succeeds. The overhead associated with the conditional flush is 5 cycles. Latency increases by 1 cycle for each transferred doubleword. Figure 5 (b) essentially shows the same effects, but with a much higher locking overhead due to the cache miss.

In general, combining decreases the latency because of the reduced number of bus transactions. The bus alignment restrictions lead to better bus utilization when going from 7 to 8 transactions, thus explaining the decreasing number of cycles.

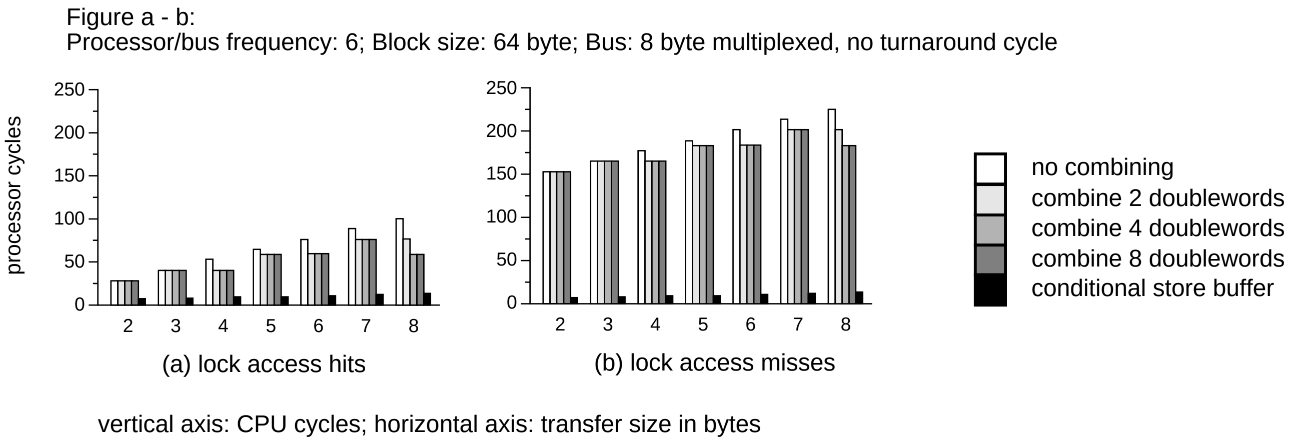

Figure 5: Comparison of Locking and Conditional Store Buffer

Discussion

The results presented in figure 5 indicate that the locking overhead is significant, especially for very small transfers or in the case of a cache miss. The absolute number of cycles will vary slightly, depending on the processor microarchitecture and the bus characteristics. Experiments with a 2-way and 8-way superscalar CPU did not change the lock overhead at all, because of the short data and control dependencies. Wider and faster buses lead to a smaller per-doubleword increase in latency. On the other hand, the processor to bus frequency ratio and cache miss penalty tends to increase, thus partially offsetting this effect. Other synchronization mechanisms, like the load-linked/store-conditional instruction pair, also affect the locking overhead. In many implementations, the store-conditional instruction results in a bus transaction even for a cache hit, which would further increase the locking overhead.

5 Qualitative Evaluation

The CSB design combines atomicity of a sequence of store instructions with high bandwidth to uncached address space. For a number of scientific applications studied in [10], the average message size ranges from 19 to 230 bytes. Other studies support these results for a wider variety of applications. It is generally accepted that DMA is too costly for small messages. Instead, NI designs like [3], [10] and [2] employ programmed I/O for message sizes below a certain threshold. Network interfaces that explicitly target fine-grain communication, like DEC Memory Channel [5] rely entirely on uncached loads and stores for message transfer.

On the other hand, scalability of a fine-grain parallel architecture depends mostly on latency, which in turn is largely determined by the per-message overhead. The CSB moves the break-even point between PIO and DMA towards bigger messages, potentially completely eliminating the need for DMA on the send side for many applications.

Moreover, the non-blocking synchronization feature opens new opportunities for the design of user-level network interfaces. Processes can be allowed to access device control registers, such as initiating a DMA transfer, without operating system involvement since atomicity is provided by the conditional store buffer.

6 Related Work

The conditional store buffer design was motivated by several existing optimizations for uncached accesses. The MIPS R10000 introduced an uncached-accelerated page attribute, which enabled the uncached buffer to detect and combine uncached stores to subsequent addresses. However, no guarantee can be made for the resulting bus transaction. The buffer stops combining when it receives a store that does not match the current access pattern. It issues a burst transaction only if an entire cache line could be combined, otherwise a series of single-beat transfers is used. This design is limited to strictly sequential access patterns.

The SPARC V9 architecture introduced block move instructions as part of VIS [13]. These instructions transfer an entire cache line between memory and the processor floating point registers, bypassing the cache hierarchy. Atomicity can be guaranteed and protection is ensured since processor registers are saved and restored on a context switch. However, floating point registers are not very well suited as a source for general I/O operations, since they don’t support bit operations and integer arithmetic. The use of precious processor registers for buffering of I/O operations increases the register pressure which might have negative performance impacts.

The non-blocking synchronization scheme is very similar to the transactional memory proposed in [6]. It is a general hardware mechanism to support lock free atomic access to shared data structures. However, since the conditional store buffer accesses only uncached address space and limits the amount of data to exactly one burst transaction, it does not require snooping of external transactions which reduces the hardware cost.

7 Summary

We have presented the design of a simple software controlled conditional store buffer which makes efficient use system bus burst transfers to improve I/O performance. The cost is a minor increase in hardware complexity of the processor and the design is consistent with current trends in microprocessor architecture. Several processors have attempted to combine stores to consecutive addresses to reduce the number of bus transactions. The CSB design just takes these optimizations one step further and gives software complete control over the combining buffer. It implements a non-blocking synchronization scheme to ensure atomicity and *exactly-once* semantics of the resulting bus transaction. The advantages are higher store bandwidth even for very small transfer sizes, and a significant reduction in synchronization costs for I/O device accesses. The next step is to evaluate the benefits of these performance advantages in terms of realistic applications, since the microbenchmarks used in this study were designed to maximize the pressure on the I/O subsystem rather than model application reality.

Acknowledgments

The author would like to thank John Carter and Wilson Hsieh for their helpful suggestions and comments in various stages of the research as well as during the preparation of the paper. Email conversation with Ulrich Bruening helped to improve the presentation of the conditional store buffer design. Kevin Van Maren provided valuable insights into the design of device drivers and I/O devices. Sally McKee's comments improved the final version of this paper.

References

[1] David Banks and Michael Prudence. A High-Performance Network Architecture for a PA-RISC Workstation. *IEEE Journal on Selected Areas in Communications*, Volume 11, No. 2, February 1993.

[2] Ulrich Bruening and Lambert Schaelicke. Atoll: A High-Performance Communication Device for Parallel Systems. In *Proceedings of 1997 Conference on Advances in Parallel and Distributed Computing*, pages 228-234, Shanghai, China, March 1997.

[3] Al Davis, Mark Swanson and Mike Parker. Efficient Communication Mechanisms for Cluster Based Parallel Computing. In *Proceedings of the First International Workshop, CANPC'97*, pages 1-15, San Antonio, Texas, February 1997.

[4] Digital Semiconductor. *Alpha 21164 Microprocessor: Hardware Reference Manual*, April 1995.

[5] Marco Fillo and Richard B. Gillett. Architecture and Implementation of Memory Channel 2. *DEC Technical Journal*, Volume 9, No. 1, January 1997.

[6] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of 20th Annual International Symposium on Computer Architecture*, pages 289-300, May 1993.

[7] Richard P. Martin, Amin M. Vahdat, David E. Culler, Thomas E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[8] MIPS Technologies. *MIPS R10000 Microprocessor User's Manual*, Version 2.0, 1996.

[9] Motorola Inc. *PowerPC Microprocessor Family: The Bus Interface for 32-Bit Microprocessors*, 1997.

[10] Shubhendu S. Mukherjee and Mark D. Hill. The Impact of Data Transfer and Buffering Alternatives on Network Interface Design. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture* (HPCA-4), pages 207-218, February 1998.

[11] Vijay S. Paj, Parthasarathy Ranganathan and Sarita V. Adve. *RSIM Reference Manual*, Version 1.0. Technical Report 9705. Department of Electrical and Computer Engineering, Rice University. August 1997.

[12] Kevin Skadron and Douglas Clark. Design Issues and Tradeoffs for Write Buffers. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture* (HPCA-3), pages 144-155, February, 1997.

[13] SPARC Technology Business. *UltraSPARC-1 User's Manual*, Revision 1.0. September 1995.