University of Rhode Island
Dept. of Electrical and Computer Engineering
Kelley Hall
4 East Alumni Ave.
Kingston, RI 02881-0805, USA

Technical Report No. 0797-0002

High Performance Memory System for High ILP Microarchitectures

Augustus K. Uht
Department of Electrical and Computer Engineering
University of Rhode Island

uht@ele.uri.edu
www.ele.uri.edu/~uht

August 26, 1997

Abstract

A new memory system is proposed that greatly widens the von Neumann bottleneck for uniprocessors with high bandwidth requirements, especially those processors exhibiting large degrees of Instruction Level Parallelism (ILP). The new system has high bandwidth and low latency, and is not costly. Further, minimal ordering of memory accesses is achieved, in that only accesses to the same main memory location are sequentialized; accesses to different addresses are independent. Memory-mapped I/O accesses are also taken into consideration. The system is described in an overview, discussing its general architecture, and then the details of the communications protocol are given. Lastly, performance of the system is discussed.

1 Introduction

High performance memory systems are needed to complement emerging microarchitectures exhibiting high levels of ILP, as well as more traditional CPU models. The classic "von Neumann bottleneck" between the CPU and main memory will increasingly be a limitation on overall computer performance, unless key memory problems are solved. This paper presents a new approach to memory architecture which provides significantly improved solutions to the following memory problems:

- Bandwidth limitations to memory
- Proper ordering of simultaneous reads and writes

The new memory system was specifically designed for use by Levo[12], a CPU machine model which allows realization of ILP of factors of 20 to 30 (2,000% to 3,000%) improvement over sequential code. The microarchitecture of Levo is described in [10], a companion document to this paper. The new memory system will also be useful for other processors of more traditional design.

The key novel features of the memory system proposed are:

This is a minor revision of the similarly-titled document dated January 21, 1996.
This work was supported in part by the Intel Corporation and the NSF under grant No. MIP-9708183.

- Disjoint-address caches, reducing address comparison hardware and readily keeping cache contents coherent
- Time-tagged memory accesses, allowing a simple algorithm for ordering accesses
- A simple interconnection network between processing element buffers, caches and memory banks

Other related problems in the design of a suitable memory system are as follows. We assume a CPU of 32 general purpose functional units or processing elements (PE). The approach of having one cache per PE has these problems:

1. From a functional viewpoint, it is difficult to keep the caches coherent, in that there could be two or more PE's with memory accesses to the same location; hence, the caches might contain different versions of the same data.

   Another, related problem is that a load from one PE might need the value of a store from a different PE (different cache). Communicating the value is problematic, and might necessitate some or all of the accesses going through the main memory, thereby nullifying the desired benefits of a cache, which is reduced access time.

   Another problem is that there could be as many as 32 or more accesses to the same location in main memory at the same time, with an arbitrary mix of reads and writes. Also, the ordering of these accesses is not known by the memory system, complicating the situation.

2. From a structural viewpoint, there is also a problem. There would still need to be 32 connections (one per PE) to the main memory. The interconnection between the two might then be impractical, if not at least unwieldy and slow.

Clearly such a design does not meet the desiderata for a high ILP memory system:

1. Low-latency and high-bandwidth access to both the cache and main memory is necessary, at least in the common cases (the common case should be made fast).
2. Multiple simultaneous accesses of any mixture of reads and writes should be properly and *minimally ordered*. By minimally-ordered memory accesses, we mean that only accesses to the same main memory location are sequentialized, with full dependencies on the memory location enforced. Memory accesses to different addresses are completely independent.
3. The system should be buildable from hardware cost and complexity perspectives. Regularity and simplicity are desirable.

A memory system that achieves these goals is presented in this paper. This system (see Figure 1) is composed of per-PE access buffers, a much smaller number of interleaved main memory banks, each bank having its own cache, a simple multiple-bus system connecting the buffers and the caches, and a new snooping-type algorithm that minimizes the ordering of memory accesses, yet ensures that those accesses that need to be ordered are. Correct ordering is achieved by *time stamping* memory accesses at issue time, and examining the resultant *time tags* in the snooping algorithm. The proposed memory system can be used in Levo, in standard superscalar machines, and in other types of machines.

The purpose of this paper is to introduce the new memory system and its operation, and to show that it is buildable. Experimental evaluation of it along with that of a high ILP CPU will follow in later work.

The remainder of the paper is organized as follows. Prior work is briefly reviewed in Section 2. A system overview of the solution is given in Section 3. The details of the access ordering algorithm and its characteristics are given in Section 4. Performance is discussed in Section 5. The solution is summarized in Section 6.

2 Background and Prior Work

Memory systems are commonly interleaved[6]. Models of such systems indicate that in many systems there may be a square root law relationship[4] between the number of banks b and the increase in effective bandwidth BW of the system, allowing for bank conflicts (references to the same bank desired at the same time), i.e., $BW = \sqrt{b}$. This would seem to imply that for our target system, in which an 8-fold improvement in bandwidth is needed (see Section 5), about $BW^2 = 64$ banks would be needed. However, for a number of reasons, this is not as likely to be an issue in our computer. Various techniques reduce our needed bandwidth.

First, high ILP processors can be designed to forward stores to loads of the same address, eliminating accesses[12]. Also, stores to the same address can be combined into a single store, under certain conditions[11]. Lastly, since in our system each memory bank has its own cache, bank conflicts do not last long (in the non-cached case, bank conflicts would last for the latency of a main memory access, potentially a very long time). We discuss this issue further later in the paper.

Early multiprocessors placed caches with memory units; however, they were then separated from the processors by a complex interconnection network, introducing a large additional delay in memory accesses, and reducing performance drastically. The proposed memory system herein does not exhibit this phenomenon, as the interconnection network between the buffers and the caches is on-chip.

VLIW machines can, in principal, largely finesse the problems with memory accesses given above, via compile-time memory access scheduling. However, in practice run-time bank conflicts can occur and require complex hardware to resolve; also, performance suffers[1].

Common superscalar machines do not employ multiple paths to memory, although the single path may be wide. The new *multiscalar* model[8], however, uses an *Address Resolution Buffer*[2]. In common with our system, the ARB allows multiple memory accesses per cycle, and uses interleaved memory banks to reduce the cost of the address comparison logic. The ARB allows both speculative loads and stores, and limited memory renaming (minimal data dependencies); in [2], it is stated that renaming is supported by the ARB, but that further research is necessary for it to work smoothly. Although Levo allows speculative loads and stores, it does not currently allow any loads to be made from speculative stores. With that exception, Levo achieves this speculation and memory renaming uniformly and easily with the already extant renaming registers (*Shadow Sinks*) in the Levo CPU, greatly reducing the complexity of the memory system as compared to the ARB, although some associative search hardware is needed in the CPU of Levo that is not necessary in the Multiscalar CPU.

Dynamically unresolved loads are allowed in the ARB. This is a load made before preceding stores' addresses are known, as soon as a load's address has been computed. This is the same as Levo's *address decoupling* scheme[12]. In the ARB, stores are made as soon as their address and data are known. In Levo, stores are made when both these conditions are true and when the store leaves the CPU (in Levo parlance, when the store has reached the first column of the Instruction Queue).

Lastly, our memory system actually reduces memory accesses, both by forwarding values in the buffers to new loads, and by eliminating redundant stores. These do not appear to be done in the ARB. They may be significant factors in reducing memory bandwidth demands.

3 New Memory System Architecture

The new memory system is novel in that it achieves high bandwidth and minimally orders memory accesses. Memory-mapped I/O is also taken into account.

Referring to Figure 1, there are 4 major components of the system: the buffers (one per PE), an interleaved[1] main memory, the caches (one per main memory bank), and the interconnection buses between the buffers and the caches. For the purposes of our discussion, we assume the memory to be low-order interleaved, i.e., on the least significant bits; other types or combinations of interleaving may be desirable, but detailed simulation and study is required to establish the best configuration. Each buffer for a typical system would hold about 2-8 accesses.

Since the main memory is interleaved, and the caches are associated with the individual disjoint banks, then the contents of the caches are also disjoint, i.e., no two caches can contain data for the same memory address. Therefore there is no possibility of the caches being incoherent, by design.

[1] A b-way interleaved memory consists of b independent *banks* or sections of memory, each of which holds $1/b$th of the total memory. With a *low-ordered* interleaved memory, the banks contain locations corresponding to memory addresses in which each bank holds every bth address. Since most memory access patterns are to sequential locations in memory, usually many accesses to the memory (to the different banks) can be made at the same time.

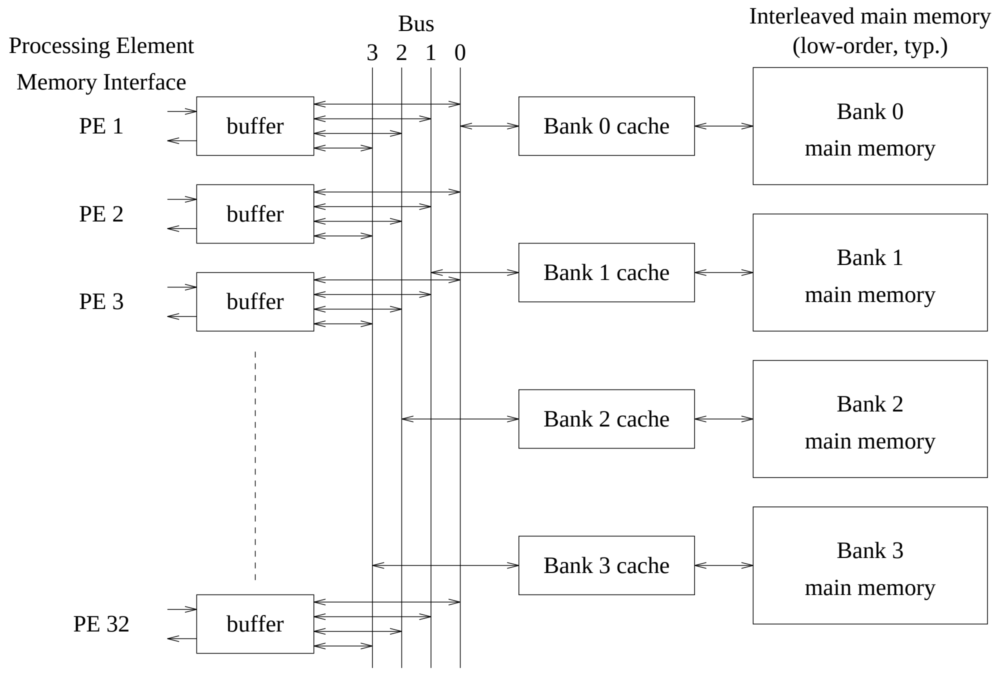

Figure 1: *Novel memory system.* For the sake of illustration, 4-way interleaving is assumed).

The number of memory banks that a high ILP computer should have is a design parameter of the particular implementation in question. The greater the number of banks, the greater the number of simultaneous memory accesses from the CPU which may be satisfied. However, of course, with more banks the cost of the system increases significantly, as well; thus, there is a tradeoff to be made.

There is one bus per cache (and hence per bank). All of the buffers connect to all of the buses.

In the common mode of operation, when a PE requests an access from a buffer, the buffer determines the proper bus to put the access on (each physical address corresponds to only one bus). If the bus is free, i.e., no other row wants to connect to that bus' associated bank, the access is made to the cache in the normal fashion. If there is a hit, the access is satisfied immediately; the cache penalty in this case may be as low as one cycle, since the bus is on-chip. If there is a miss, the access will have to be satisfied later. Multiple accesses to the same cache may be supported via commonly known standard techniques, when one or more accesses must wait for main memory to be read ("non-blocking" caches). We will not consider the multiple access case further at this time.

In rarer cases, when a PE makes an access (load or store) to an address `X` which is the same as that of accesses in other buffers, the situation, and hence operation, becomes much more complicated. Fortunately, the worse the scenario (the greater the number of simultaneous accesses to `X`), the less likely it is to occur, and the more time that can be taken to handle it. In general, all of the buffers snoop[3] (look at the activity on) all of the buses. The time tags and addresses of the pending accesses in a buffer are constantly compared with those appearing on the buses. When addresses match, the buffers take special actions to properly order the conflicting accesses, and to reduce the overall number of accesses. These *multiple access* cases and their resolution are considered in detail in the following section, in which a new access-order conserving protocol is presented and its operation described.

4 Access Ordering and Reduction Algorithm and Its Implementation

For a high performance and buildable bus system and access protocol the following desiderata exist:

- All main memory address space accesses should be correctly and minimally ordered. This includes memory-mapped I/O accesses.
- Any necessary arbitration should have low latency, and thus not contribute overly to the normal access time. Ideally, an access should be satisfied in one cycle.
- The throughput of arbitrations and accesses must be at a high rate, ideally one of each per cycle per bus.

4.1 Algorithm Outline

In this section we present an overview of the characteristics of the ordering algorithm. Details follow in the next section. For most of our discussion we assume no cache misses, and single-cycle cache accesses. These constraints and their relaxation are discussed at the end of the presentation.

Every memory access is classified at instruction issue time as being of one of three types of accesses, Load, Store or I/O. Possible I/O addresses can be hardwired into the overall computer system, or they can be part of a BIOS. Thus, there are three possible types of accesses in the system at any time. Each type of access is handled somewhat differently.

In order for multiple loads and stores to the same address to be correctly ordered (but still be as weakly ordered as possible), the memory system must have an indication as to the time of execution of the accesses. Thus, when an access instruction is issued to a PE, it is stamped (appended) with a *time tag*.

With respect to the Levo prototype, this tag does not indicate a physical or machine cycle time, but is rather a number composed of the column number of the access in the Instruction Queue (IQ) concatenated with the IQ row number of the access. The numbers are small integers, increasing with increasing column numbers and row numbers (later time). Since there are likely to be relatively few active columns (~ 8) at any given time, the time tags could be only 8 bits long.

The tag indicates the nominal dynamic order that accesses should be performed in to ensure correct program (sequential) semantics; the lower the tag value, the earlier the access is to be performed. The wraparound case can be handled as in the Metaflow machine[7]. The tag of an access stays with the access when transferred to the buffer, and onto the bus. In fact, a buffer holds all issued but not executed memory accesses sent to its corresponding PE. Multiple instructions and access instructions can be issued per cycle, as long as the time tags reflect the nominal sequential order. As with the tags used for sink result routing from the Levo pipeline, the column part of the memory time tags must be altered (decreased by one) whenever the IQ complex is left-shifted (first column retired).

One of the key challenges of the algorithm is prioritizing the pending accesses (issued but not executed) with respect to their time tags, as semantically necessary, at high speed. This is done by doing the following at each buffer for each bus:

- The number of accesses being attempted in the current cycle is classified as 0, 1, or greater than 1. (The 0 accesses case is trivial.) This classification is performed at high speed (a few gate delays) with a special wired-OR based “counter” circuit, described in the Appendix.
- If there is 1 access to be made in the current cycle, it is just made.
- If there are 2 or more accesses desired to be made in the current cycle (on the same bus, within one or more buffers on the bus), then another special circuit, a slower “prioritizer”, comes into play. (It actually overlaps the beginning of its work with the counter circuit.) In the current cycle no access takes place, but the prioritizer determines the next (earliest time tag) access to appear on the bus in the next cycle. In the next cycle, the next prioritization is made for the third cycle, and so on, pipelining the prioritizations with the data transfers, and achieving a steady-state of 1 data access per cycle.

  Thus, one cycle of data transfer is lost whenever two or more accesses are attempted after a cycle of 0 or 1 accesses. Since the norm is likely to be two or more attempted accesses per cycle, on the average few cycles will be wasted. The possible loss of bus utilization is an area for future improvement.

  The prioritizer circuit is further described in the Appendix.

Characteristics of the algorithm are as follows:

1. During accesses, the time tags of the pending accesses are compared with that of the access on the bus; the addresses of same are also compared.
2. The default is to handle accesses in time tag order; thus, deadlock cannot occur.
3. Under appropriate conditions, stores to the same address can be combined to one actual store.
4. In many cases, a load following a store to the same address grabs the store's data off of the bus at the end of the store's cycle, eliminating the load's actual access. The load request is thus satisfied faster, returning the datum a cycle earlier.
5. It is also possible for non-I/O loads to be made out-of-order on the same bus. It is alway possible to execute all non-I/O accesses out-of-order on different buses.

4.2 Details of the Algorithm

Memory-mapped I/O is handled in the new memory system. An important assumption that is made in this case is that every I/O access has potential state-modifying side effects. Therefore, all three of the classic data dependencies (flow, anti- and output; or RAW, WAR and WAW) must be enforced amongst all I/O accesses, even those with differing addresses. Additionally, with side effects, I/O reads must also be dependent (RAR). Therefore, as far as all actions except the actual data movement are concerned, I/O reads and writes are treated the same. Also, we assume that DMA (Direct Memory Access) operations may be performed by I/O units to and from the main memory. Therefore, every I/O access is a barrier. All accesses, I/O or otherwise, regardless of address, with time tags earlier than a given I/O access must execute before the I/O access executes. Similarly, all accesses with later time tags must wait for the I/O access to occur before being executed themselves. The DMA assumption may or may not require the caches to be flushed of dirty data upon an I/O access, depending on the aggressiveness of the design of the I/O system.

4.2.1 Access Nomenclature

Accesses that have been issued (and are thus in a buffer) but not executed are called *Pending Accesses* (PendAcc). An access which is on the bus in the current cycle has a B subscript for "Bus"; e.g., Ld_B - load on the bus, and St_B - store on the bus. A bus' pending accesses have a p as a subscript, e.g., Ld_p and St_p.

4.2.2 Protocol

Each buffer constantly snoops the buses, in particular comparing the addresses on the bus with the addresses of all of its pending accesses. Special operations, given below, must be performed if there are matching addresses. Time tags are also compared. Each pending access in a buffer has associated with it its own address and data registers, time tag, access classification type tag, and conditional data flag.

Unless otherwise noted, the actions below happen in all of the buffers for each of the buses concurrently.

Single Access Case: The time tag of the access on a bus (TTag_B) is compared with the time tags of the pending accesses for that bus (TTag_p). If the bus tag is less than all of the pending tags, there is no problem, and the access goes to completion.

If TTag_B is greater than TTag_p, however, and either the bus or the pending accesses is a store or an I/O access, the bus access must be squashed. This is true because either: the I/O access must be treated as a barrier, or: loads must be ordered with stores, and stores must be ordered with stores, BUT loads need NOT be ordered with loads.

Two other situations must be addressed. First, if the bus access is an I/O access, and its time tag is less than a time tag of a concurrent access on another bus, then the concurrent access is squashed. Second, if any pending access of any type on another bus has a time tag less than an I/O access on another bus, the latter I/O access must be squashed. These precautions must be enforced to ensure that every I/O access is a barrier, even to other I/O accesses. (Note that main memory accesses to DMA buffer blocks can still be concurrent with each other and with other main memory accesses.)

Multiple Access Case: This scenario occurs when two or more pending accesses try to use a bus in the same cycle; it is detected with the counter circuit, and is handled with the prioritizer circuit. The cycle currently in progress when the collision(s) is detected is used for the first prioritization. By the end of this cycle the highest priority access of the pending accesses is selected, and is executed on the bus in the next (second) cycle. In the second cycle, the second prioritization is made (not including the access currently on the bus), and the selected access is placed on the bus in the next (third) cycle, and so on. In this way, the slow prioritizations are overlapped with data transfers, and thus, after the initial one cycle latency, the bus operates at its peak transfer rate.

All pending accesses participate in the prioritization process. The accesses are prioritized with respect to their time tags; earlier issued (lower-numbered or smaller) time tagged accesses have higher priority and are performed first.

Reduction of Accesses: Assume that the access on the bus is a Store Access StAcc_B. If any pending Store Access StAcc_p has the same address as StAcc_B, then the accesses can be combined. Since the accesses are prioritized, the bus access' time tag is guaranteed to be less than the pending access. The bus access is squashed (at the end of the cycle). If there is no other pending store with the same address, the access goes to completion, and the data is stored in the cache.

If one or more pending Load Access LdAcc_p has the same address as StAcc_B, then the load can potentially use the store's data. Therefore the load saves the store's data in its data register, marking the data as Conditional, and the load no longer seeks to use the bus. This load continues to pick store data off of the bus for subsequent stores on the bus with the same address, until it is the time that the load would otherwise go on the bus, i.e., when it has priority since its time tag is the earliest (lowest numbered) of the bus' pending accesses. At this point the load uses the Conditional data, which is the result of the latest store to that address, as the result of its execution, and sends the data to its corresponding PE; the load is thus finished executing, and it is executed one cycle earlier than had combining not been used.

If the Conditional bit had not been set for that Load's data, then the Load access would use the bus normally and be executed normally once it had priority. Note that other Loads may occur out-of-order with this load. Note that I/O Accesses may not participate in combining, since they are barriers.

Other Considerations: Only write-back caches are needed. In a worst-case scenario, with I/O connected to the main memory directly (not through the caches), an I/O access causes all of the caches to be flushed and their dirty data to be written to their corresponding main memory banks. Thus, if a DMA request follows, the memory has the latest and correct data with respect to the I/O request.

Instruction fetching is handled similarly to the regular data Loads. The fetch Buffer(s) are connected to the same buses, and each instruction has a time tag set when the instruction is issued. The data and instruction time tags are constructed the same way. Instructions are held in a buffer until they are executed. Thus, self-modifying code is handled correctly; any store data to an instruction address will be picked up by the instruction, if the instruction is being fetched. Otherwise, the store will have executed before the instruction fetch is satisfied, and thus the instruction will still get the correct (modified) data.

Cache misses and multiple-cycle cache latencies are now considered. Non-blocking caches are desirable for improved performance. In such a scenario, pending accesses which were once on the bus but resulted in a cache miss continue to monitor the bus for the return of data. Continued snooping of the bus by these pending accesses allows them to stall the bus as necessary, or to grab data from a later store to the same address as previously described. Although load misses snoop the bus, they do not participate in the prioritization process.

The cache can gain control of the bus to give data to a load miss. The cache uses the load's time tag and address to participate in the prioritization process, and eventually forwards the load's data to the load over the bus. In this case, the load looks for a store (the cache's return of data) with an address AND time tag equal to its own to grab the returned data.

4.2.3 Summary of the Algorithm

The effect of the above rules is to temporally sequentialize the stores to the same location, and interleave loads to the same location with these stores. This results in the loads getting the proper values of the memory location, and in the cache being written (to that location) in the proper order.

Thus, with respect to a given memory address or location, the stores to that location are strongly ordered, and loads are temporally ordered with stores. However, for all of the loads between two stores, the loads are unordered. All accesses to different memory locations are completely unordered. These conditions are necessary and sufficient for a minimal ordering of memory accesses with correct code operation.

4.3 Summary of Benefits of the New Memory System

1. An arbitrary number of simple caches may be used in a high-access computer, esp., one containing a high-ILP CPU.

2. High memory access bandwidths are supported.

3. The caches are simple. They are single-ported (to the CPU), and need no relatively-involved bus operations when communicating with the buffers.

4. The accesses to the caches and memory are high-speed, i.e., low-latency.

5. The hardware required is simple and regular.

5 Performance Estimation

The novel memory system described above, configured with 8-way interleaving, should be sufficient to handle the requirements of a high ILP processor averaging 20 instructions per cycle. We argue this as follows.

First, we assume that 40% of dynamic instructions have data memory references; from [6] this is conservative both for CISC and RISC machines. Therefore, a memory system for our target machine should sustain an average of 0.4*20= 8 accesses per cycle. If purely random accesses are assumed, then the square root law would hold and the memory system would be insufficient. However, accesses are not purely random; many tend to be sequential, raising the effective bandwidth above $\sqrt{8}$ accesses per cycle, and closer to the peak of 8. Also, with a high ILP machine, there are likely to be many accesses pending per cycle on average, and these accesses are typically independent, due to ILP enhancements. Therefore, the banks (and buses) are likely to always be busy, supplying close to the peak of 8 accesses per cycle.

The bandwidth requirements of the memory are reduced for a couple of reasons. Those accesses that are still dependent can be combined either in the processor or the memory system, reducing the number of accesses. Also, many stores can also be combined, as described above or as shown in [11]. In the latter study, the number of dynamic stores in a program were reduced by 45%, on average, for general-purpose code.

The number of pending memory accesses can be further smoothed over time by computing store effective addresses speculatively, before their dependencies resolve. This also enhances ILP. Also, loads can be performed speculatively, also increasing the smoothing. However, in the latter case this can also lead to more accesses being generated per cycle, putting more pressure on the memory system. On average, this is not likely to be a large percentage of the overall bandwidth, assuming reasonable branch predictors.

Instruction fetching can also increase bandwidth requirements significantly. However, if suitable dynamic[5] or static[9] instruction stream techniques are used, the required instruction fetch bandwidth will dramatically decrease, reducing the pressure on the memory system.

From the above reasoning, it seems likely that an 8-way interleaved memory system as described herein will satisfy the memory requirements of high ILP machines. Such a system should be more than adequate for superscalar machines with less aggressive ILP exploitation.

6 Summary and Conclusions

A memory system suitable for use with high ILP processors is proposed. It exhibits high bandwidth and low latency, and is readily constructed. The caches within the memory system are coherent by design, due to their interleaving. A hardware algorithm that properly orders and reduces the accesses in the system has also been given. Its realization is also straightforward.

This new system substantially widens the "von Neumann" bottleneck. Memory need not limit high ILP performance.

Appendix: Counter and Prioritizer Circuits

The following signals are added to the normal signal lines on each bus; both new lines are wired-OR signals, i.e., any number of bus drivers (buffers) can assert them at the same time. The signal is "true", asserted, when one or more buffers assert it. (Alternatively, if wired-OR lines are not advisable from a device technology viewpoint, discrete gates can be used in an equivalent circuit.)

$\texttt{Any1}_i$: There are n of these lines corresponding to the size (in bits) of the time tags. Each line is asserted when any buffer's corresponding ith tag bit is a 1.

$\texttt{Any0}_i$: These are the same as the `Any1` lines, except that each is asserted when any buffer's corresponding ith tag bit is a 0.

The `Any1` and `Any0` lines are used to determine quickly (within a few gate delays) if there is more than one access being attempted on the bus in the current cycle.

$\texttt{PriorityAny0}_i$: These lines are similar to the prior defined `Any0` lines, but are used over the course of an entire bus cycle to determine the highest priority (earliest time tag) access to use the bus in the next cycle. It is low speed, but is overlapped with data transfers on the bus.

Counter Circuit: The purpose of this circuit is to determine if there are 0, 1 or more than 1 accesses trying to use a bus at the same time. The circuit's delay is only a few gates-worth. It works as follows, using the `Any0` and `Any1` lines. At the beginning of a cycle, all pending accesses put their time tag values on the `Any` lines, as described above. There are 0 accesses competing for the bus if all of the `Any` lines are not asserted (or any `Any0` and `Any1` pair of any bit are both not asserted).

1 and only 1 access desires to use the bus iff $\texttt{Any0}_i = \overline{\texttt{Any1}_i}$ for all bits i. Since all accesses competing for the bus have unique time tags, if two accesses compete for the bus they will differ in at least one bit, and the prior condition will not be true for at least one bit.

Thus, 2 or more accesses are indicated by $\texttt{Any0}_i$=$\texttt{Any1}_i$=asserted for one or more bits i.

Prioritizer: This circuit enables only the highest priority access competing for the bus in a given cycle. This access has the lowest time tag value of the competing accesses. Since this circuit is relatively slow, the access granted the bus does not use the bus until the next cycle. The `PriorityAny0` lines are used here.

Thus, there is one output `BusGrant` per possible access per bus (one for each buffer entry per bus). The delay of the circuit is linearly proportional to the number of bits in the time tags, although this is shown to be reducible later.

The circuit works as follows.

1. The most significant bit of each possible access' time tag is put on the corresponding `PriorityAny0` line. If the access' bit is a 0, then it is possible that this access gets priority, depending on the relationships with the lesser significant bits. This access also gets priority, as far as this bit position is concerned, if the access bit is a 1, and after a settling time the corresponding `PriorityAny0` line is not asserted (all of the other accesses have 1's in their most significant bits, too).

2. For any bit position, a competing access is possibly the highest priority if $\texttt{BusGrant}_\texttt{i} = \overline{\texttt{TTag}_i} + \overline{\texttt{PriorityAny0}_i}$ is true. (i indicates the bit positions of the three variables; this computation is carried out for each possible access.)

   For each bit position:

   - If this bit position indicates that the access can not possibly get priority, then this access is kept from participating in the rest of the prioritization process. This is indicated if the access' tag bit is a 1, but the `PriorityANy0` line is asserted, indicating that there is at least one other competing access with higher priority (at least one other access having a 0 in its corresponding tag bit). `BusGrant` for this access is kept not asserted.

- If this bit position indicates that the access might be the highest priority access, then this access participates in the rest of the prioritization process, i.e., looking similarly at lesser significant bits.

3. For all bit positions lower than the most significant position, $\texttt{PriorityAny0}_i$ is asserted if the following is a 1: $\texttt{BusGrant}_{i+1} \cdot \overline{\texttt{TTAg}_i}$. In other words, the `PriorityAny0` line for a bit is driven only if the next more significant bit's computation indicates that the access may be given priority.

4. The overall `BusGrant` signal for an access is then equal to $\texttt{BusGrant}_0$, e.g., of the least significant bit.

Therefore the computation as outlined has $\bigcirc(n)$ delay, which can be large. To circumvent this, the per bit computation (radix-2) can be performed more bits at a time, e.g., 2 bits at a time, or radix-4. In the latter case, there are now 4 bused lines associated with each bit position, $\texttt{PriorityAny0}_{i/2}$, $\texttt{PriorityAny1}_{i/2}$, $\texttt{PriorityAny2}_{i/2}$ and $\texttt{PriorityAny3}_{i/2}$, each corresponding to a particular combination of two tag bits. The circuitry for this scenario is straightforward, and results in a delay of $\bigcirc(\log_{\log_2 r} n)$, where r is the radix mentioned above.

References

[1] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. *IEEE Transactions on Computers*, C-37(8):967–979, Aug. 1988.

[2] M. Franklin and G. S. Sohi. A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, 45(5), May 1996.

[3] J. R. Goodman. Using Cache Memory to Reduce Processor Memory Traffic. In *Proceedings of the 10th International Symposium on Computer Architecture*, pages 124–131. IEEE-ACM, June 1983.

[4] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill Book Company, New York, N. Y., 1984.

[5] R. F. Krick and A. Dollas. The Evolution of Instruction Sequencing. *COMPUTER*, 42(4):5–15, April 1992.

[6] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, Inc., San Mateo, California, 2 edition, 1995.

[7] V. Popescu, M. Schultz, J. Spracklen, G. Gibson, B. Lightner, and D. Isaman. The Metaflow Architecture. *MICRO*, 11(3), June 1991.

[8] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. IEEE and ACM, June 1995.

[9] A. K. Uht. Concurrency Extraction via Hardware Methods Executing the Static Instruction Stream. *IEEE Transactions on Computers*, 41(7):826–841, July 1992.

[10] A. K. Uht. Overview of the Levo High-ILP Computer. Technical Report 0797-0001, Department of Electrical and Computer Engineering, University of Rhode Island, August 1997. Available via `http://www.ele.uri.edu/~uht`.

[11] A. K. Uht and D. B. Johnson. Data Path Issues in a Highly Concurrent Machine. In *Proceedings of the Twenty-Fifth International Symposium on Microarchitecture (MICRO-25)*, pages 115–118. ACM and IEEE, December 1992.

[12] A. K. Uht and V. Sindagi. Disjoint Eager Execution: An Optimal Form of Speculative Execution. In *Proceedings of the 28th International Symposium on Microarchitecture (MICRO-28)*, pages 313–325. IEEE and ACM, November/December 1995.