Memory Latency Hiding Techniques
메모리 지연을 감추는 기법들

Ando Ki (기안도)  
Computer System Department  
(컴퓨터-소프트웨어기술연구소 컴퓨터시스템연구부 선임연구원)

The obvious way to make a computer system more powerful is to make the processor as fast as possible. Furthermore, adopting a large number of such fast processors would be the next step. This multiprocessor system could be useful only if it distributes workload uniformly and if its processors are fully utilized.

To achieve a higher processor utilization, memory access latency must be reduced as much as possible and even more the remaining latency must be hidden. The actual latency can be reduced by using fast logic and the effective latency can be reduced by using cache. This article discusses what the memory latency problem is, how serious it is by presenting analytical and simulation results, and existing techniques for coping with it; such as write-buffer, relaxed consistency model, multi-threading, data locality optimization, data forwarding, and data prefetching.

I. Memory Latency Problem

The latency of memory accesses is the time spent waiting for the data to arrive at the processor, and it is a well-known obstacle in shared-memory computers because processors stall for the duration of memory requests.

The speed gap between processor and memory has widened in the last few years, and the technology trends indicate that this gap is not likely to decrease in the future [11]. The tendency of computer architecture in which processors and memories become more physically distributed [20], makes memory access latency increase. The inter-

connection network of multiprocessors becomes complex, resulting in network delay which contributes to memory access latency. The characteristics of increasing network traffic in parallel workloads result in network and memory contention which increase the memory access latency still further. As the memory access latency becomes larger, high-performance computers become more sensitive to stalls for memory accesses.

The most fundamental approach to tackling the latency problem is adopting a multilevel memory hierarchy. The goal of the memory hierarchy is to bridge the speed gap
between high speed processors and relatively low speed memories by introducing faster cache storages between them. The cache is a small but fast memory whose purpose is to hold currently active data, thereby reducing the number of accesses to memory [21]. The memory hierarchy works effectively because of the principle of locality: during an execution of a specific part of program, the program refers to only a small portion of its address space. Therefore, by keeping the active fraction of the address space in fast storage, the program runs as if the whole address space is fast. Even though the memory hierarchy works well, the processor must wait until the required data arrives if the data is not found in the nearest storage. Therefore, although caches reduce the effective memory access time, cache misses still suffer from large memory access latency.

We will show that the latency of memory accesses is a serious problem by presenting two examples that are an analytical model and a simulation result.

1. A Simple Analytical Model

We assume a simple memory hierarchy model that consists of one-level cache and an ideal processor that executes one instruction per cycle. We take a typical program in which the number of instructions executed is $N_I$ that can be divided into $N_C$, $N_S$, and $N_L$, where $N_C$ is the number of instructions that do not refer to memory, $N_S$ is the number of stores, and $N_L$ is the number of loads. In this simple model we ignore synchronization and store stall times. In practice, store stall time can be removed by using a write-buffer and relaxed memory consistency model [7, 13].

The program execution time ($t_{exe}$) with cache miss rate ($m$) will be

$$N_C + N_S + N_L + N_LT_C + mN_LT_M,$$

where $T_C$ is cache access time and $T_M$ memory access time. Load stall time ($t_{stall}$) due to memory reads will be

$$N_LT_C + mN_LT_M.$$ 

Therefore, the load stall time overhead ($R_S$) that is the ratio\(^1\) of load stall time ($t_{stall}$) to the program execution time ($t_{exe}$) will be

$$\frac{IT_C + mIT_M}{1 + IT_C + mIT_M},$$

where $l$ is the ratio\(^2\) of the number of loads ($N_L$) to the number of instructions ($N_I$).

For simplicity, we only consider global loads, and, according to SPLASH-2 characteristics [26], $l$ ranges between 0.1 to 0.2 and $m$ is up to 0.02 (2%) with a reasonable cache configuration (direct mapped, 1-MByte size).

Figure 1 shows load stall time overhead ($R_S$) versus miss rate ($m$) with load rate $l = \frac{N_L}{N_I}$.

---

\(^1\) $R_S = \frac{t_{stall}}{t_{exe}} = \frac{N_LT_C + mN_LT_M}{N_C + N_S + N_LT_C + mN_LT_M}$

\(^2\) $l = \frac{N_L}{N_I}$
0.15 and different values for cache access delay ($T_C$) and memory access delay ($T_M$). The results shown in Figure 1 imply that cache hit time should be faster to reduce load stall time because miss rate 0 biases load stall times. This is why a multi-level cache hierarchy is needed, because a faster cache near the processor effectively reduces the cache hit time. Figure 1 also reveals that although the cache hit rate is high enough (miss rate is less than 1%), the program suffers large amount of memory access stall time that is spent in waiting for data. As the results with $T_C = 1$ show, $R_S$ is more sensitive to latency of memory access when the cache access time is small.

2. A Simulation Result

For the sake of reality, we conduct a simulation of a set of parallel applications on a shared-memory architecture, to show the impact of memory access latency.

The programs are taken from the SPLASH-2 benchmark suite [26]. The architecture we modeled is a typical shared-memory multiprocessor that consists of 8-processors and 8-way low-order interleaved memories connected via an interconnection network. The memory hierarchy is composed of a two-level cache hierarchy (4-Kword cache size and 4-Word cache line size for the primary cache with direct mapped write-through, and 16-KWord cache size and 8-Word cache line size for the secondary cache with direct mapped write-back). We used no-wait reply for the primary cache, 10-cycle access time for the secondary cache, 50-cycle latency for network, and 50-cycle delay for memory.

Figure 2 shows a simulation result that breaks down the total execution times into instruction execution, stalls due to memory accesses, and synchronization. As we can see, memory access stalls (store stall and load stall sections), waiting for memory accesses to complete, are significant.

From these two demonstrations of the analytical and simulation results, it is now clear

---

3) In practice, a word means 4-byte, so that 4-KWord is equal to 16-KByte.
4) When an access hits on the primary cache, the processor accesses the data without any delay.
5) If we assume an aggressive architectural model in which the processor runs 250-MHz processor cycle and executes one instruction per cycle, then 10-cycle cache delay results in 40-nsec, 50-cycle means 200-nsec delay.
that a considerable amount of execution time is wasted in waiting for data and this causes the processor to be under-utilized. Hence, we conclude that reducing the stall time is a challenging approach to get higher computer performance.

II. Coping with Memory Latency

Many solutions have been proposed to deal with the memory access latency problem. The first two approaches are mainly concerned with hiding write latency. The middle one approach exploits overlapping opportunity between computations and memory accesses. The last three approaches focus on maximizing cache hit rate.

- **Write-buffers** [5, 6, 7, 13] can hide write latency by buffering write requests and re-leasing the processor from waiting for the write to complete.
- **Relaxed consistency models** [7, 8] allow buffering and pipelining of memory requests by relaxing the order of requests. Buffering writes using a write-buffer and a relaxed consistency model can remove almost all write stall times and is complementary to data prefetching since each tackles different parts of stall times [7, 13].
- **Multi-threading** [1, 2, 19, 22] allows processors to hide memory access latency by switching from one thread to another when a long latency reference is encountered, and proceeding with other useful work while the miss is being serviced. Multi-threading may not be beneficial when the context switching cost is larger than the memory access penalty.
- **Data locality optimization** [3, 24] optimizes data partitioning or layout to avoid unnecessary memory references. Many techniques such as blocking [15] and loop transformation [25], are devised mainly for numerical applications. The benefits depend on programs and data sets and this optimization approach makes the compiler complicated and compilation time long.
- **Data forwarding** [14, 23] can hide the memory latency of sharing accesses by sending data to destination processors’ caches or local memories when the data
is available. This technique is only applicable to shared data.

- **Data prefetching** [4, 10, 12, 18] hides the memory latency by bringing data close to the processor before the data is actually used.

These techniques are discussed in detail in the following sections.

### III. Write-Buffer and Relaxed Consistency Model

Load instructions usually make the processor stall until the required data arrives. However, store instructions do not need to make the processor wait unless the write request is lost. In other words, a write request generated by a store instruction can release the processor from waiting for the write to complete, and the write request reaches the memory at a later time. Write-buffers use the above idea. The write-buffer enables the processor to continue its execution without stalling by buffering write requests.

There are two possible mechanisms to manage the write-buffer; **no-bypass scheme** in which write requests are buffered and release the processor from waiting for the write to complete and read requests cannot bypass pending writes; **bypass scheme** in which write requests are treated as the no-bypass scheme does, but read requests can overtake pending writes. It is not difficult to imagine the no-bypass scheme cannot get much benefit, because the benefit of cycle saving is limited by the distance between a write request and the next read request which must wait all write requests in the write-buffer finish. The bypass scheme has the potential to remove stall times due to write requests, since read requests can be issued in spite of pending writes in the write-buffer.

When reads are allowed to bypass writes as in the case of bypass scheme, memory access order differs from the original program order. This can result in an incorrect result if a program depends on the order of memory accesses. Therefore, the programmer should be aware of a certain rule which specifies the implication of memory accesses and should code programs using explicit synchronizations including lock, barrier, and memory-barrier (or fence) if necessitated.

The **strictest ordering** model does not allow any relaxation: *i.e.* each load and store makes the processor wait for each of them to complete. The **sequential consistency** model [16] allows write requests to be buffered, but not to be bypassed by reads because all reads and writes must be observed in the same order by all processors. The **processor consistency** model [9] allows write requests to be buffered and reads to bypass the buffered writes because write issued from different processors can be seen in different order by different processors. There are several more relaxed consistency models such as **weak consistency** [5]

No matter how relaxed the consistency model is, there are only two schemes for the write-buffer as discussed at the beginning of this section. From the viewpoint of write-buffer, the strictest consistency model must not use write-buffer, the sequential consistency model can use no-bypass write-buffer, and the other more relaxed consistency models including the processor, the weak, and the release consistency models, can use bypass write-buffer.

**IV. Multithreading**

Figure 3 illustrates how multithreading hides memory access latency. When thread 1 encounters a cache miss caused by loading location, it is swapped out and the processor begins to execute thread 2. Hopefully by the time the thread 2 needs to be swapped out when it encounters a cache miss, the memory fetch for the thread 1 has completed and, therefore, the thread 1 is ready to run again.

Multithreading has the following advantages. It can handle arbitrarily complicated access patterns, and it can be applicable to existing executable without recompilations. However, there are several disadvantages. It has the overhead of context switching, and it cannot improve execution time of a single application unless there are sufficient concurrent threads of the application.

**V. Locality Optimization**

Locality optimizations attempt to make caches more effective by restructuring computation to enhance data locality, so that it could avoid unnecessary memory reference by optimizing data partitioning or layout.

One particular example is blocking (or tiling) algorithm which operates on submatrices (called block) rather than operating on entire rows or columns of an array, so that data loaded into the caches are reused.

While locality optimization approaches are quite useful in reducing accesses to memory, their applicability is somewhat limited since extensive data dependency analysis and re-programming are required.
VI. Data Prefetching and Data Forwarding

The basic idea of data prefetching is to generate data fetch requests prior to the actual use of the data. Ideally, the memory access latency time could be hidden completely if all data are brought close to the processor before they are used by it. From the viewpoint of the processor, data prefetching can improve processor utilization by overlapping computation and memory accesses.

Figure 4(a) shows a simple example of how demand fetches make the processor wait. The demand fetches for A and B that are not present in the cache, experience memory access delay. If somehow prefetches for A and B can be issued, then A and B will be available before the demand fetches are issued. Figure 4(b) shows how prefetch hides memory access delay. For this example, we assume that prefetch can be overlapped with computation and other memory accesses.

Prefetching schemes differ in where the prefetching data is located, how prefetching addresses are determined, and when prefetchings are initiated. Prefetching data is usually located into cache, and it remains visible to the memory coherency activity, and it is kept consistent until the processor actually uses the value. Prefetching addresses can be determined by software, hardware, or combination of these two. In hardware (controlled) prefetching, the hardware alone decides what data to prefetch and when and where to prefetch the data. Therefore, prefetchings are handled dynamically without intervention of compiler or programmer, and there is no need to change code. With software (directed) prefetching, the compiler or programmer directs prefetching by inserting prefetching instructions into the code. The software prefetching needs the non-blocking load or prefetching load instruction, and special hardware to treat this instruction. The software prefetching increases compilation time, runtime overhead, and code size because additional instructions are needed to calculate the
prefetching address and issue the prefetching loads. Integrated prefetching or hybrid prefetching is a technique combining hardware and software.

There are two main hardware approaches; one is sequential prefetch whose success depends on spatial locality and the other is stride prefetch that is based on stride calculation. In the sequential prefetch method, access to cache line causes prefetching for successive line or lines. If there is sufficient spatial locality, this approach can work well. In a stride prefetch method, the instruction and operand address of each load is stored in a hardware lookup table and the instruction address enables the system to find the access stream, which belongs to the same instruction. The operand address of each load is compared with the previous one from the same instruction address and the difference of these two operand addresses becomes the stride. The stride is used to determine the possible next access by adding the stride and the current operand address.

Whereas data prefetching initiates data movement operation by consumer, data forwarding initiates it by producer. When a processor updates a word in its cache, data forwarding also propagates the update to the caches of the processors that will use the word in the near future. Hopefully, when these consumer processors refer to the word later on, they will find the updated data in their caches and, therefore, proceed without stalling.

Data forwarding needs a special instruction, which combines a write and a sender-initiated forwarding operation, and requires an extensive compiler support.

The DASH deliver operation [17] transfers a copy of a cache block to one or more explicitly specified cluster caches. The KSR poststore instruction [23] enables a processor to forward data to other processors.

VII. Summary

As processor speed is increasing at a much faster rate than memory speed and computer system scales up in the number of processors, latencies for memory accesses increase. Therefore, latency hiding techniques for computer systems are critical for achieving a higher performance. This article has attempted to address latency hiding techniques: write-buffer, relaxed consistency model, data locality optimization, data prefetching, and data forwarding.

In this section, the article is concluded by summarizing the possible benefits of the latency hiding techniques. Data prefetching is very useful in reducing the stalls due to read latencies. Write-buffer in association with relaxed consistency model can remove almost all stalls due to write latencies. Data forwarding can be useful in reducing latencies for only cacheable and shared data.

To achieve overall benefits, first, the la-
tency should be reduced as much as possible through locality optimization. After that, the remaining latency should be reduced by using write-buffer, data prefetching, data forwarding, and multi-threading. While combining more than one technique such as write-buffer, data prefetching, and data forwarding can result in a better performance, multi-threading in association with data prefetching can make the performance worse, since switching context between different threads could pollute the cache in case where there is substantial cache interference.

**References**


