The Cache Injection/Cofetch Architecture: Initial Performance Evaluation

Veljko Milutinovic, Aleksandar Milenkovic, and Gad Sheaffer[1]
University of Belgrade, Serbia, Yugoslavia
[1]Intel Corporation, Oregon, U.S.A.
email: emilutiv@etf.bg.ac.yu

Abstract

One of the major problems in a number of SM (Shared Memory) and DSM (Distributed Shared Memory) applications is the overall cost of read misses in conditions when: (a) system latencies are relatively large, and (b) a shared data item is read relatively few times by each of the processors in the system; modern SM and DSM systems are typically based on off-the-shelf microprocessors which do not include any support for the described problem. Consequently, the major goal of our research is to come up with a new concept to be incorporated into the next generation microprocessors, so they can become more efficient in the sense described above. Existing 64-bit processors support only data prefetching (PF) as a method to fight against negative effects of the described problem. Our research introduces a new concept referred to as cache injection (CI), as well as the related cache injection/cofetch architecture (CICA). Initial performance evaluation is performed using a simulation methodology based on the set of synthetic benchmarks of interest for the research sponsor.

1. Introduction

The CICA approach has been defined as a method to minimize the probability and the negative effects of the read miss, in SM and DSM systems. This paper compares existing solutions (which can be synthesized from features supported by the present day microprocessors) and variations of the proposed solution (which can be synthesized only with the newly proposed cache injection mechanism). This general idea was first introduced in [1]; however, it has been considerably improved through the research presented here [2].

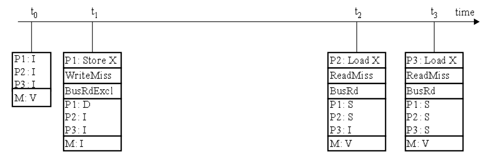
This paper represents absolutely no views of Intel and its future developments. Coauthorship of Gad Sheaffer is a consequence of academic discussions with V. Milutinovic which led to a segment of Ph.D. thesis research of A. Milenkovic.

Proceedings of the IEEE MASCOTS-97,
Haifa, Israel, January 12-15, 1997.

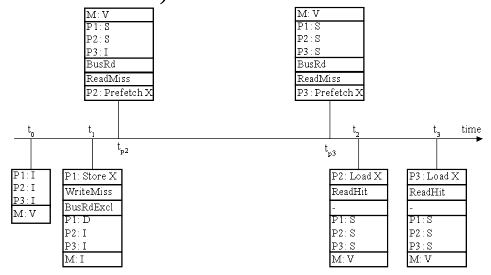
2. Cache injection/cofetch strategies

The proposed cache injection/cofetch architecture includes three scenarios, in the context when one processor node is a data producer and one or more processor nodes are potential data consumers: (a) one or more consumers express a potential need for certain data (by executing appropriate code generated at compile time), and *injection* is initiated by producer (on write-back); (b) one or more consumers express a potential need for certain data, and *injection* is initiated by a consumer (on first consumer read); (c) no consumer is responsible to express the need for certain data; however, a producer (based on its code generated at compile time) forcefully *injects* the potentially needed data into the cache memory of one or more potential consumers. The term *cofetch* stresses the fact that the above described injections can be done concurrently at various processor nodes in the system. In this paper, the first two scenarios have been further elaborated; the third one is the subject of a follow-up work.

Figure 1. includes all necessary descriptions and explanations for the existing solutions: (a) classical (processors do not include data prefetching), and (b) prefetching (PF). Figure 2. includes all necessary descriptions and explanations for the cases: (a) cache injection on producer write-back, and (b) cache injection on first consumer read. In the first case, the data producer initiates the write-back command to force the block back into the memory; at the same moment, all processors that have estimated that the related block will be needed, will inject the block into their respective caches (the IAT table contains the addresses of data estimated to be needed). In theory, estimation can be done either by the compiler or the programmer. If the injection destination is the cache - there may be implementational problems. Consequently, an intermediate buffer is needed. Later, data can be automatically transferred from this buffer into the cache, if the estimation was proved to be correct and the data are really needed. In the second case, the first of the consumers will not find the needed data in its own cache; however, other consumers will, due to the

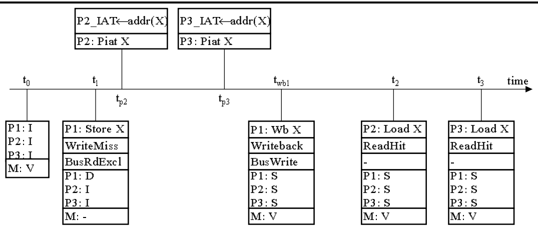


a) *classical solution*

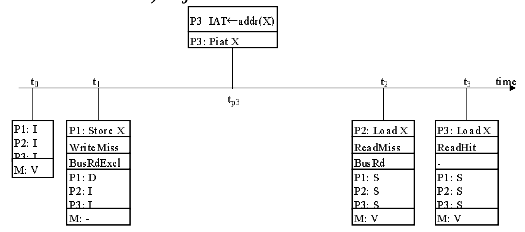


b) *prefetch solution*

Figure 1: Existing solutions. **Legend:** I-Invalid, V-Valid, D-Dirty, S-Shared, T_{WM} - Write Miss Time, T_{RM} - Read Miss Time, T_{RH} - Read Hit Time. **Description:** A sequence of instructions is shown to demonstrate a producer-consumer relationship: first P1 writes to location X; second P2 reads location X, and third P3 reads location X. **Explanation:** (a) The P1 store instruction causes a read-exclusive bus operation, to retrieve the exclusive copy of the cache line. The P2 load instruction initiates a read bus operation, P1 snoops the line and sees that it has the block dirty, and P1 flags this using the dirty line on the bus, which disables the memory and enables it to drive the data; P1 then transitions to a shared state along with P2, which reads data. The memory controller also triggers a memory write in order to update main memory. The P3 load instruction is translated into a read bus operation. Processors P1 and P2 perform snoop, but they have no dirty copy of data; they do not interact with this read; (b) The sequence of instructions which is considered above has been extended with prefetch instructions; P2 and P3 insert prefetch instructions to prefetch data which is expected to be used. **Implication:** (a) P1_wait_time=T_{WM}, P2_wait_time=T_{RM}, P3_wait_time=T_{RM}; Bus cycles: bus_read_exclusive + 2*(bus_read); (b) P1_wait_time=T_{WM}, P2_wait_time=T_{RH}, P3_wait_time=T_{RH}; Bus operations: bus_read_exclusive + 2*(bus_read).



a) *injection on write-back*



b*) injection on first consumer read*

Figure 2: Proposed solutions. **Legend:** IAT - Injection Anticipated Table. **Description:** Instead of a prefetch instruction, P2 and P3 issue the *Piat* instruction, which puts the address of the data that is expected to be used in the IAT. **Explanation:** (a) When processor P1 issues a write-back instruction, processors P2 and P3 catch the data, and put it into their caches; (b) In this case, processor P1 (producer) does not issue the write back instruction. When processor P2 (first consumer) reads data, processor P3 (second consumer) will catch the data, if the address of that data is in the IAT of the processor P3. **Implication:** (a) P1_wait_time=T_{WM}, P2_wait_time=T_{RH}, P3_wait_time=T_{RH}; Bus operations: bus_read_exclusive + bus_write; (b) P1_wait_time=T_{WM}, P2_wait_time=T_{RM}, P3_wait_time=T_{RH}; Bus operations: bus_read_exclusive + bus_read.

above described action of the first consumer. An important issue here is the state of the arriving block (it must arrive as a read-only block). This is of relevance for the first case, as well. If more consumers are involved with a right to write, only one is to obtain some type of exclusive rights, which is the subject of a follow-up research (others have to stop injecting in that case, in a predefined way).

3. Initial performance evaluation

Initial performance evaluation is performed using synthetic address traces for a bus based multiprocessor. The major goal of the simulation analysis is to compare performance of the existing (Classical, PF) and proposed (CI, CI+PF) solutions. Execution time and bus traffic are used as performance measures. Application of interest is described through a set of workload parameters. Details of workload, software, system architecture, conditions, assumptions, and related references for each solution are given in [2].

The results of simulation show that each suggested solution can contribute to performance, but the right combination of prefetch and injection mechanism is the winning solution for a number of different applications of interest. The performance benefit for the solution which combines prefetch and injection approaches is between 5% and 15%, depending on application type.

4. References

[1] V. Milutinovic, “Some Solutions for Critical Problems in the Theory and Practise of Distributed Shared Memory: New Ideas to Analyse,” School of Electrical Engineering, University of Belgrade, TR-ETF-95-037, Belgrade, Serbia, Yugoslavia, December 1995 (www.computer.org/TAB/TCCA/tcca.htm).

[2] A. Milenkovic, *Applying the Injection Mechanism to Memory Hierarchy of Reflective Memory Systems*, Ph.D. Thesis in Progress, University of Belgrade, Belgrade, Serbia, Yugoslavia, 1996 (ubbg.etf.bg.ac.yu/~emilenka/cica.zip).