The Data Transfer Engine: Towards a Software Controlled Memory Hierarchy

Victor Garcia[*,1], Alejandro Rico[*],
Carlos Villavieja[†], Nacho Navarro[*],
Alex Ramirez[*]

[*] *Barcelona Supercomputing Center - Centro Nacional de Supercomputacion*
[*] *Universitat Politecnica de Catalunya, Barcelona, Spain*
[†] *University of Texas at Austin, Texas, USA*

ABSTRACT

In today's computer architectures, many scientific applications are considered to be memory bound. The memory wall, i.e. the large disparity between a processor's speed and the required time to access off-chip memory, is a yet-to-be-solved problem that can greatly reduce performance and make us underutilise the processors capabilities. Many different approaches have been proposed to tackle this problem, such as the addition of a large cache hierarchy, multithreading or speculative data prefetching. Most of these approaches rely on the prediction of the application's future behaviour, something that should not be necessary as this information is known by the programmer and is located in the application itself. Instead of designing hardware that attempts to guess the future, the goal should be to provide the programmer with the hardware support required to decide when the data is transferred and where is it transferred to. With this goal in mind, we introduce the Data Transfer Engine, a runtime-assisted, software prefetcher that exploits the information provided by the programmer in order to place data in the cache hierarchy close to the processor that will make use of it. The DTE can not only significantly reduce stall time due to cache misses but, more importantly, it allows us to design new computer architectures that are able to tolerate very high memory latencies.

1 Introduction

For more than 20 years the speed disparity between processors and off-chip DRAM memories has been consistently increasing [WM95], a problem that has not yet been solved in the multi-core era. Although we are no longer increasing processor's clock frequencies, the off-chip memory must now serve many cores that may be simultaneously requesting data.

[1]E-mail: victor.garcia@bsc.es

Many solutions have been proposed in order to alleviate this problem. One of the most frequently used is data prefetching. Prefetchers request data from off-chip memories and bring it close to the processor that will supposedly request it in the future, usually by moving the data to some level of the cache hierarchy. While prefetching can be greatly beneficial in some situations, it may not work very well in others, and can even degrade performance if it is not used correctly [PHES05].

Prefetching can be done by means of hardware or software mechanisms. Hardware prefetchers have logic that dynamically detects the pattern of memory requests and attempts to predict future ones. Software prefetching is implemented traditionally with prefetch instructions available to programmers and compilers.

While hardware prefetching is quite common nowadays, with most commercial microprocessors having one, hardware prefetching has some important limitations. It is reactive and speculative, i.e., it takes a number of accesses to start predicting the next ones. This behaviour is very successful when the memory access pattern is predictable. On the other hand when the memory access pattern is irregular or aleatory, hardware prefetchers can hardly do correct predictions and in most cases it can be counterproductive by producing unnecessary network traffic and cache pollution [PHES05]. Software prefetchers are not speculative, but rely on either the compiler or the programmer inserting the prefetch instructions at the right point in time, which is hardware-dependent and can vary from execution to execution.

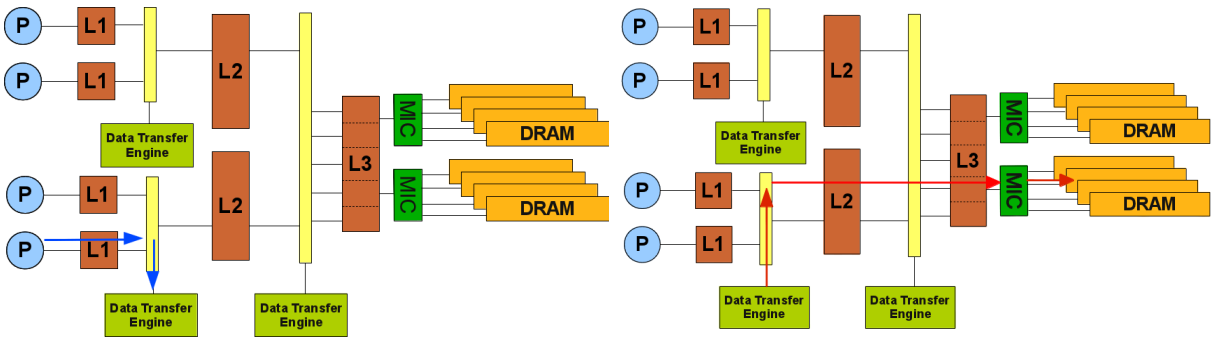
Data movement is then a key aspect in order to obtain high performance, but the speculative nature of prefetching mechanisms cannot take advantage of an important fact: the programmer already knows how and where the data will be used. This information should be made available to the hardware in order to efficiently bring data on-chip.

In this work we propose the Data Transfer Engine (DTE), a software block prefetcher that takes advantage of the information given by the programmer in order to decide where and when to move the data that each processor will use. This information can be given by the programmer in the form of code annotations or pragmas, and is used by the runtime system to program the DTE to effectively prefetch data, avoiding some of the problems of traditional prefetching techniques such as low accuracy or bad timeliness.

If the runtime system is able to fetch the required data in time, the matter of the memory latency is then moot, as longer latencies only mean earlier movement of data. This means we could re-think our system architecture design and make it not so dependent on low memory latencies.

2 Data Transfer Engine

The Data Transfer Engine is a block-based software prefetcher that takes the commands that are dynamically created by the runtime system in order to fetch large chunks of data. The information made available by the programmer through pragmas is used to create "prefetch commands" (PC). A PC is a request for a chunk of data composed of a virtual address and a size. On arrival to the DTE it is translated into one or many cache-line-sized requests These prefetch commands are scheduled to be executed before the computation that will require the data, as in this way it is likely that the data will arrive by the time the processor demands it. Since the data requested can be of greater size than a page, the virtual address must be translated at the DTE and not by the processor's memory management unit. This means the DTE must have a local MMU and a TLB where it catches the translations needed in order



(a) PC is sent to DTE (b) PC is split on mem requests going to DRAM

Figure 1: Architectural design of a system with a Data Transfer Engine

to create the physically addressed line-size requests. The addition of another TLB that has to be kept coherent with the CPUs' TLBs may cause additional overheads due to TLB shoot downs, but this overhead can be reduced by eliminating the Inter-Processor Interrupts used by the OS in the TLB shoot downs [VKV+11].

Figure 1 shows the architectural design of a system with a Data Transfer Engine. The DTE can be placed next to the L2 cache level, or L3 if it exists, and is connected to the same bus as the cache memories. In the first case, a prefetch command executed by the processor reaches the DTE right after passing through the L1, and the memory requests generated there are directed towards the cache level above (L2 in this case). If the DTE were placed next to the L3 cache level, the PCs would travel through the L1 and L2, and the resulting memory requests would place the data in the L3 level. Although previous works on software prefetching have concluded that it is more beneficial to prefetch into the primary cache level due to shorter latency penalties on miss rates [MG91], these prefetching schemes worked at a cache line granularity. Our DTE can prefetch large blocks of data, causing additional problems due to an increased number of invalidations and evictions that can cause performance degradation, thus we limit the placement of the DTE to the L2 and L3 cache levels. In our future work we plan to explore the possibility of combining the DTE with a hardware prefetcher that would be in charge of moving data to the L1 from the cache levels above.

The DTE can create a large number of memory requests in a short period of time, straining the interconnection network and delaying the on-demand loads that are in the critical path, so we must prioritise them over the less important prefetch requests. Our initial approach in order to do this has been to treat them differently. Normal on-demand loads that miss on a cache level are stored in the MSHR to be requested to the next cache level or to main memory. On the other hand prefetch requests originated from PCs are stored in a separated prefetch queue. Every cycle both queues are checked, and a decision is taken on which request is sent. We are still experimenting with different prioritization policies, with the simplest one being to always allow MSHR request to take precedence over prefetch requests. In order not to have old requests hanging on the prefetch queue, a maximum life time can be set for these requests. If after a number of cycles N a request has not left the prefetch queue, it will be removed from the queue to leave newer requests to take place.

3 Conclusions

We have presented the Data Transfer Engine, a runtime-assisted, block software prefetcher, that takes advantage of the knowledge the programmer has of an application's data usage in order to effectively anticipate data movement. The DTE does not rely on speculation or smart compilers in order to predict which memory accesses will be needed in the future. The large off-chip memory latencies found on today's system architectures are one of the main causes of performance loss, and have been the driving force behind some of the design decisions in current microarchitectures. By giving the programmer the hardware required to use his or her knowledge in order to plan data movement, we believe we could completely avoid the memory wall. If we know when and what data will be required at each moment, the latency we must pay in order to fetch it is irrelevant, as larger latencies only mean earlier prefetching.

4 Acknowledgements

This research has been supported by an FPI grant from the Universitat Politecnica de Catalunya, the Consolider program (Contract No. TIN2007-60625) and CICYT (contract No.TIC 2001-0995-C0201, TEC2004-03289) from the Ministry of Science and Innovation of Spain, the ENCORE project (ICT-FP7-248647), the TERAFLUX project (ICT-FP7-249013), and the European Network of Excellence HIPEAC-2 (ICT-FP7-217068).

References

[MG91] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12:87–106, 1991.

[PHES05] Thomas R. Puzak, A. Hartstein, P. G. Emma, and V. Srinivasan. When prefetching improves/degrades performance. In *Proceedings of the 2nd conference on Computing frontiers*, CF '05, pages 342–352, New York, NY, USA, 2005. ACM.

[VKV+11] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S. Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 340–349, Washington, DC, USA, 2011. IEEE Computer Society.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.