CS203A FALL 2003 PROJECT

Term paper on micro-architecture

Asst. Prof. Jun Yang
Department of Computer Science
University of California, Riverside

PREFETCHING

Keri Nishimoto
Abhishek Mitra
Som C. Neema

Wrong-Path Instruction Prefetching [1]
Summary by Keri Nishimoto

To improve the performance of superscalar processors and high speed sequential machines, Jim Pierce and Trevor Mudge propose a scheme called *Wrong-Path Instruction Prefetching* to fetch instructions from memory into the instruction cache before the instructions are needed

Wrong-path prefetching is a combination of next-line prefetching and target-always prefetching. Next-line prefetching fetches the next sequential cache line(s) whenever it is not resident in the cache. The target prefetching used in wrong-path prefetching fetches the line containing the target of a conditional branch. It makes no attempt to predict whether or not the branch will be taken. It fetches the branch target regardless. Thus, for conditional branches both paths are always prefetched: the fall-through direction by next-line prefetching, and the target line by target prefetching.

Note that this target prefetching used by wrong-path prefectching is different from table-based target-line prefetching, which uses a target prefetch table containing current line and successor line pairs. Target-line prefetching uses the table to prefetch the line that was requested next the last time the current line was executed. The table and the associated logic used in this scheme requires significant additional hardware that is not needed in wrong-path prefetching.

Target prefetching in wrong-path prefetching cannot initiate a fetch until the branch target is computed. In simple implementations, this does not happen until the decode stage. Because of the lateness of the fetch, prefetching the target line when the branch is taken is unproductive. The prefetch request would be generated at the same time as the cache miss. For this reason, there are no target prefetches for unconditional jumps and subroutine calls. One possible improvement would be to add a prefetch buffer which has the ability to partially decode the instructions that are stored there. This would allow target prefetches to be initiated earlier. Unfortunately this addition caused a slight performance gain only when the cache was large.

The idea behind wrong-path prefetching seems counterintuitive. Next-line and target-line prefetching try to predict the correct execution path and only prefetch down the predicted path since fetching down wrong paths would likely increase memory traffic and cause cache pollution. Wrong-path instruction prefetching, however, relies on the tendency that instructions accessed on mispredicted paths will later be accessed during correct path execution. The authors accept the increased memory traffic as a necessary cost and conclude that the benefits of prefetching overcome the added cache pollution.

In simulations carried out by the authors with various cache parameters, wrong-path prefetching performed up to 64% better in terms of CPU cycles over no prefetching. It also outperformed next-line prefetching, table-based target-line prefetching, and a hybrid scheme that combines next-line and table-based target-line prefetching.

Instruction Prefetching Methods
Keri Nishimoto

Prefetch algorithms reduce instruction cache misses by prefetching instruction lines into the cache. Currently, I-cache misses are a significant source of performance loss, especially in integer and database codes. The performance penalty due to these misses is predicted to increase in the future due to (1) the increasing performance gap between processors and memory and (2) increasing instruction level parallelism [4].

Many variations of instruction prefetching have been proposed. Software-based methods rely on the compiler to specify when and what to prefetch by inserting explicit prefetch instructions into the executable. While very little hardware needs to be introduced to take advantage of software prefetching, there are numerous disadvantages. These include (1) the additional overhead of the prefetch instructions, (2) the fact that not all compilers on a machine may implement prefetching, (3) old binaries are

not optimized, (4) the limitations in predicting behavior at compile time, and (5) the portability difficulties over different implementations of the ISA.

Hardware based prefetching does not require software support and can be divided into two categories: correlated prefetching and non-correlated prefetching. Correlated prefetching correlates previous cache misses with other events, such as old misses (as in Markov prefetching [5]) or previous instructions (as in branch-history-guided [6] and execution-history-guided prefetching [4]). This information is usually stored in a dedicated table. In non-correlated prefetching, hardware predicts which instructions will be executed in the near future and prefetches them. Methods that fall in this category are fetch directed instruction prefetching [7], next-n-line prefetching, and wrong-path prefetching [1].

[Summarized by Abhishek Mitra]

Tango [2] [14], an on-chip dynamic data prefetching mechanism which does not overload cache ports, skips through non memory reference instructions and builds on the accuracy of the branch predictor to implement data prefetching for superscalar processors. The prefetch requests are scheduled to tango (sic.) with the requests from the core to the cache during free time slots.

The three components of Tango are the Program Progress Graph (PPG), SRPT (Reference Prediction Table for Superscalar processors), and the PRC (a Prefetch Requests Controller). The superscalar processor may issue at most two memory reference instructions and one branch instruction per cycle.

The PPG works as an extension to the Branch Target Buffer (BTB), and is a directed graph, with nodes labeled as the entry number of a particular branch instruction in the BTB/PPG and each edge directs to the next branch labeled as <T/NT>, <Num> where <T/NT> signifies a Taken / Not Taken branch and <Num> is the number of instructions between the two branches. The data structure for the PPG appends to the regular BTB, with entries for taken and not–taken branches and the number of instructions spaced in between. The PPG utilizes additional 32 bits per entry and is generated from the current instruction window of the processor.

SRPT is a cache, indexing the memory reference instructions by two tags, pc-tag (address) and pre-pc-tag (BTB index, T/NT, mem-ref-num). The mem-ref-num refers to the position of the instruction within the basic block with respect to other memory reference instructions. Prediction is generated by using information from previous accesses i.e. the ‘stride’(difference between last two addresses), ‘times’(number of iterations already accessed by prePC), ‘last-ea’ (effective address) fields and an fsm (a finite state machine which disables prefetching only for the while when the last two strides differ from each other). When the PC accesses a memory reference instruction the SRPT is updated for that entry.

A look-ahead mechanism is implemented by a prePC which advances through the PPG to predicted branches. Simultaneously, the corresponding SRPT entries are matched using the pre-pc-tag for their branch indexes and direction to generate prefetch requests for up to two memory reference instructions (dual ported memory) per clock cycle. As the prefetch requests have a lower priority, the distance between prePC and PC is controlled by tango to account for fetch latency and other delays.

PRC is a prefetch requests controller scheduler which works in tandem with four fully associative queues and an LRU touch mechanism.

The first queue, Req-Q, is a FIFO that stores upto four prefetch requests to the data cache. Redundant requests are avoided by a search and exclusion. Wait-Q, is a buffer, which stores missed prefetch requests to the main memory along with devoting two priority entries for core requests. The prefetch requests to the main memory are also duplicated on the Track-Q and removed therefrom when the memory services the requests.

A wrong prediction leads to flushing of prefetch requests from the Wait-Q and Req-Q. A full Req-Q or a Track-Q stops the prePC from advancing.

Filter-cache, is an unique FIFO buffer to store prefetch requests for data already present in the cache. Every entry has a decrementing counter initialized to 'fetch latency + fetch spacing'. It eliminates redundant requests which arrive as a principle of locality and assists superscalar processors by virtue of removing redundant requests and reducing memory traffic.

Finally the prefetch requests exclusive of Req-Q, Wait-Q, Track-Q and Filter-cache are sent to the Req-Q buffer.

LRU touch is done on a block, whenever a prefetch request results in a cache hit on the block because the block would be accessed sooner or later and it would not be a nice idea to get it evicted.

A very aggressive prediction scheme which doesn't wait for stride stabilization, coupled with highly efficient cache port bandwidth usage make tango a high-performance pre-fetch mechanism for superscalars so much so that a reduction in the number of memory transactions compared to a base system and an overall performance improvement of 1.36 (as an improvement in CPI) is claimed by the authors.

Hardware-Based Data Prefetching [15]

Hardware based data pre-fetching avoid additional software / instruction overheads, and dynamically prefetch data from the main memory and into the data cache. The tango scheme is based on previous work documented in [ ]. The authors have cited three types of memory access patterns viz. scalar, zero stride and constant stride, and the third one is the case which is benefited by prefetching.

The authors have classified hardware based prefetching into three schemes namely, Basic Reference Prediction, Lookahead Reference Prediction and Correlated Reference Prediction. The tango mechanism builds on the first two prediction methods and hence they are touched upon below.

Basic reference prediction utilizes a Reference prediction table (SRPT builds on this idea) tagged with the PC of a memory reference instruction and stores the previous address, stride and the fsm (as in tango). Prefetching is done when stride stabilizes. There is a possibility of timing mismatch between arrival of prefetched data and its use in an instruction, especially if the instruction is referenced again before a time less than or equal to the memory latency.

Lookahead reference prediction is advancement over the above mentioned scheme and it implements a Look Ahead Program Counter (the prePC builds on this idea) that accesses the BTB to move onto the next predicted iterations or basic blocks and start prefetching early enough. When the PC catches up with addresses visited by LA-PC, the data is already in the cache. The ideal lookahead distance to be maintained by this scheme should be equal to the latency of the next memory hierarchy.

Programming Inference: Documented on Page 8.
[End of Abhishek Mitra's Work]

Distributed Prefetch-buffer/Cache Design for High Performance Memory Systems[3]
Summary by Som C. Neema

The authors have addressed the issue of disproportionate growth rates between the CPU speed and Main Memory speed.

They propose a hardware data prefetching technique and the associated memory system architecture that performs much better than existing methods for data read requests that do not have temporal or spatial localities of accesses but which repeat over time.

The essential scheme is to predict future CPU read requests to main memory and prefetch the data into small SRAM prefetch-buffers that are integrated on each DRAM IC. By this mechanism the access latency for the DRAM is improved and the CPU-Memory bandwidth is conserved, thus reducing the wrong prediction penalty. A correlation based prefetching scheme is used to detect patterns across loop levels. The

solution performs better than the alternative of predicting using one block look-ahead (OBL) or stream buffers in large L2 caches that are ineffective for programs that traverse large data structures and have complex data references.

The read request patterns are stored in a prediction cache by storing the physical addresses, clustering the addresses into blocks, to reduce the number of unique addresses. A table update mechanism tracks the program execution and uses the past address request patterns to predict future requests. Large prefetch buffers reduce the number of address bits for each prediction.

Prediction is done using a table lookup method that simultaneously prefetches all the guessed block addresses of the next reference based on the present entry into the SRAM prefetch buffers. Storing the memory access that follows a memory-read-access does table update.

The authors present a design example using 64 MB EDRAM ICs with integrated 32 KB SRAM buffers. The memory is 4-way interleaved but there are no concurrent accesses to banks. The address cluster size is 256 bytes. The prefetch block size is 2 KB and hence the number of guesses is 4 for each entry.

Simulations were done on nine programs including 7 from SPEC95, for 2 billion instructions using SHADE, (simulator of SPARC instruction set) to extract data references and cycle timing, and *ldvsim*, to simulate the memory system. Calculations of CPI and Cycles per memory reference show that the technique performs marginally better compared to the traditional L2 cache for 7 programs that perform well with L2, but there is significant speedup for 2 programs in the benchmark suite.

The authors conclude that their scheme is able to accurately predict CPU read requests to main memory, has a very low prediction miss penalty and conserves bus resources. They also contend that the proposed memory system architecture is compatible with conventional microprocessors and it could be possible to further improve prediction consistency by improved prediction techniques.

RELATED WORK **(thorough part)**

Som C. Neema

Hardware data prefetching techniques can be sequential or a variant: basic, lookahead and correlated (the scheme in [3]). The basis for all the three variations is a Reference Prediction Table (RPT) that holds data memory access patterns[8].

Linked Data Structures (LDS) like trees and lists have complex and frequent memory access patterns with data dependencies that require pointer indirection requiring specialized prefetching techniques.

Prefetching schemes could be *processor side prefetching* or *memory side prefetching* depending on where the prefetches are initiated[9].

Yang and Lebeck[10] present a memory side-prefetching scheme with a prefetch engine at each level of the memory hierarchy that executes the load instructions that traverse the LDS, independent of the CPU and sends the data to the L1 cache or a prefetch buffer.

Hughes and Adve[9] also propose a memory side scheme scheme in which the processor sends prefetch commands encoding the traversals to the prefetch engine that runs ahead of the processor and pipelines data to the CPU..

Roth et al[11] presents a dependence based prefetching technique, which uses the relationship between loads that produce addresses and those that use them. To hide the load latencies of LDS traversals it uses a small prefetch engine that dynamically determines the program code that computes addresses and then speculatively executes it parallel to the program.

M. Karlsson et al [12] discusses use of jump pointers to access nodes down a linked list and prefetch in parallel nodes that will be accessed in the future into prefetch arrays similar to prefetch buffer/cache. This technique is effective also for short lists and for small computations per node.

Data prefetching may demonstrate side effects of cache pollution and increased

consumption of processor memory bandwidth and congestion[1]

A Comparison of the Papers [1], [2], [14] and [3]

A high misprediction penalty caused by a cache miss limits the performance of the processors. The three papers discuss different prefetching techniques to overcome this problem. [1] discusses a scheme for instruction prefetching while [2], [14] and [3] propose data prefetching techniques. While instructions exhibit a large locality of reference it is not so for data and sequential techniques do not give the best results.

Tango uses branch prediction to prefetch the data whereas wrong path prefetching does not use any predictors but always fetches the target and fall through instructions in conditional branches to L1(by a combination of *next-line* and *target-always* schemes) . Distributed prefetch-buffer /cache design [3] uses a correlation scheme to lookup the predicted block address from the predictor table based on the current address.

The scheme in [1] requires minimal additional hardware whereas Tango[2] uses an on chip scheme that requires additional hardware or modifications to implement its PPG (Program Progress Graph), SRPT (Cache), and the PRC (a Prefetch Requests Controller) (the size of a 4KB dual port memory). Distributed prefetch-buffer/cache design[3] also requires a distributed cache with an integrated SRAM prefetch buffer on the DRAM chip and a prediction - prefetch controller. Hence while Wrong-path instruction prefetching [1] tries to avoid misses in the instruction cache by inserting extra fetch instructions, Distributed prefetch-buffer /cache design [3] tries to reduce the memory access latency and Tango tries to make good use of the slack time and hardware resources not being used for main computation.

Wrong path instruction prefetching increases cache pollution because it prefetches instructions that might not be required. Tango uses a LRU touch mechanism to promote cache data so that it is not replaced but like the distributed cache mechanism it does not prefetch data to the L1 cache. The distributed cache mechanism tries to conserve the memory CPU bandwidth, but prefetch requests may generate excess traffic for data which may not be utilized later.

Tango employs a complex scheme to search for memory reference instructions and modifying the prediction table to generate prefetch requests. Distributed buffer uses a correlation scheme to look up a table to predict and update.

Target prefetching in [1] initiates a fetch only after the target is computed (ID stage) whereas [3] initiates as soon as the current address is available (IF stage). For [3] performance improvement happens best for programs that have no localities of accesses but have time repeated data read requests. Tango [2] is most effective when data is accessed in constant strides. [End of 'Group effort']

References

[1] J. Pierce and T.N. Mudge. Wrong-path instruction prefetching. In *International Symposium on* Microarchitecture, pages 165-175, 1996.

[2] S. Pintar, A. Yoaz. Tango: a hardware-based data prefetching technique for superscalar processors. In *Proceedings of the 29th Annual International Symposium on Microarchitecture,* pages 214-225. December, 1996.

[3] T. Alexander, G. Kedem. Distributed prefetch-buffer/cache design for high performance memory systems. *Proceedings of the Second International Symposium on High-Performance Computer Architecture,* San Jose, CA, USA, February 1996.

[4] Y. Zhang, S. Haga, and R. Barua. Execution history guided prefetching. In *Proceedings of the 16th international conference on Supercomputing,* pages 199-208, New York, New York, USA, 2002.

[5] D. Joseph and D. Grunwald. Prefetching using markov predictors. *IEEE Transactions on Computers*, 48(2):121-133, 1999.

[6] V, Srinivasan, E.S. Davidson, G.S. Tyson. Branch History Guided Instruction Prefetching. In *proceedings of the 7th International Conference*

on High Performance Computer Architecture (HPCA), pages 291-300, Monterrey, Mexico, January 2001.

[7] G. Reinman, B. Calder and T. Austin. Fetch Directed Instruction Prefetching. In *Proceedings of the 32nd Annual ACM/IEEE international symposium on microarchitecture on MICRO-32*, pages 16-27, Haifa, Israel, November 1999.

[8] J-L. Baer and T-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, 1991.

[9] C. J. Hughes and S. V. Adve. Memory Side Prefetching for Linked Data Structures, Provisionally accepted (subject to minor revisions) for the Journal of Parallel and Distributed Computing (JPDC). Available as Department of Computer Science Technical Report UIUCDCS-R-2001-2221, University of Illinois at Urbana-Champaign, May, 2001.

[10] C-L Yang and A. R. Lebeck, Push vs. Pull: Data Movement for Linked Data Structures, *International Conference on Supercomputing 2000 (ICS '00),* May 2000.

[11] A. Roth, A. Moshovos and G. S. Sohi**.** Dependence Based Prefetching for Linked Data Structures *In proc. of ASPLOS-8*,Oct.4-7, 1998

[12] M. Karlsson, F. Dahlgren and P. Stenström. A Prefetching Technique for Irregular Accesses to Linked Data Structures. *In the Proceedings of the 6th International Conference on High Performance Computer Architecture (HPCA'6)*, pp. 206-217, January 2000.

[13] S.P. VanderWiel, D.J. Lilja. Data Prefetch Mechanisms *ACM Computing Surveys (CSUR)* Volume 32 , Issue 2 Pages: 174 - 199 (June 2000)

[14] S.S. Pinter and A. Yoaz. Tango: a hardware-based data prefetching technique for superscalar processors. Technical Report TR88.371, IBM Research Center, Haifa, Israel, September 1996.

[15] T.Chen and J.Baer. Effective hardware-based data prefetching for high-performance processors. IEEE Transactions on Computers, 44(5):609 – 623, May. 1995.

Cover page Illustration: courtesy "Fetch Softworks"

Programming Inference (Abhishek Mitra)

We implement a small program which multiplies two matrices B, C of sizes 200x200 and stores into matrix A of same size. This is a good example when data is accessed in constant strides for matrices B,C.
Pentium 4 processor makes use of hardware pre-fetching (as documented by Intel) and its efficacy could be assessed with this small program. We use Intel VTune Performance Analyzer v6.1 which collects performance data with respect to an application using counters present on the processor.
We measure the number of Loads generated by this program, and the L1 cache miss rate. The 1st-level cache on the Intel Pentium 4 processor contains data only, not instructions. If the data is not found in the 1st-level cache, the processor will look in the 2nd-level cache. “Pentium 4 reference, Intel VTune Perf. Analyzer v6.1”

```
int _tmain(int argc, _TCHAR* argv[])
{
	float a[200][200],b[200][200],c[200][200];//each element = 32bits
	int i,j,k;
	for (i=1;i<200;i++)
		for (j=1;j<200;j++)
			b[i][j]=c[j][i]=rand();//fill in matrices with random
data

	for (i=1;i<200;i++) /* Loop to calculate matrix multiplication*/
		for (j=1;j<200;j++)
			for (k=1;k<200;k++)
				a[i][j]+=b[i][k] * c[k][j];

	printf("done  ");//scanf("%d",&a);
	return 0;
}
```

Total Loads to L1 cache: 102,883,050
Total Stores in L1 cache: 16,318,214
L1 Data cache Load miss: 8,534,725
L1 Load miss rate = 0.0829
L1 Load hit rate = 0.917

We could not analyze L1 data cache store misses as they were not available in the software.
Conclusion: Considering that stores are a small percentage of the total accesses and the resulting error due to store misses would be negligible, the hit rate of Pentium 4 is lower than what tango has to offer for all cases except for spice2g6 simulation [14].
The block size for the cache in Pentium 4 is 64 bytes and its size is 8Kbyte resulting in 128 blocks versus the 32KByte, 32byte block size for the tango simulations. In this case many cache blocks are in use by other applications such as the O.S., VTune, etc. It may be possible that this processor may benefit from a larger L1 cache, but its ramification on the clock rate needs to be investigated.