# HARDWARE BASED PREFETCHING METHODS

**Master's Project**
**Department of Electrical and Computer Engineering**

**Angeline Ebenezer**
**December 19, 1998**

# TABLE OF CONTENTS

# 1. INTRODUCTION

It is a well−known fact that the gap between microprocessor speed and memory speed is widening over time. Unless, memory latency is significantly reduced, it poses a threat to overall system performance. In multiple issue pipelined processors, the memory latency is the major contributor to the length of each CPU cycle[25]. Hence, by reducing the memory latency, the overall cycle time is reduced.

One brute−force solution is to increase the primary cache size or its associativity. However, this may not always be possible or desirable for an on−chip cache because the cycle time of the cache is limited by its size and is a major factor in determining the overall CPU clock speed.

Prefetching information from main memory into cache can reduce memory latency. In other words, the cache could be primed if you could accurately predict what information the processor would need after the current instruction has been executed. For this reason, active research is going on in both the areas of instruction and data cache prefetching. Although data cache prefetching is more recent than instruction prefetching, both are of considerable value in reducing memory latency.

The prefetching strategies must accomplish two things to be effective. First, they must be able to predict which cache lines will soon be referenced and second, they must initiate the prefetch requests far enough in advance so that the miss latencies are significantly reduced or eliminated entirely.

The different prefetching schemes can be broadly classified into two categories; *software* and *hardware−based* prefetching techniques. This paper is intended to address the hardware based technique and will address both instruction and data prefetching. One major advantage of the *hardware−based* techniques is that they need no help from the programmer or the compiler. A drawback, however, is that (depending on the technique used) the *hardware−based* techniques could use up expensive hardware resources.

Different prefetching techniques can be useful for different applications. This paper presents various prefetching techniques for both instruction and data caches. The intent is to introduce the reader to these different techniques so that he can select the ones that best suit his needs. The reader is assumed to have some knowledge of computer architecture.

This paper is organized in three sections. This first one is "Instruction Prefetching methods", the second is "Data Prefetching methods", and the third is "Stream Buffers". The "Stream Buffers" section presents papers that apply to both instruction and data cache prefetching. Each section introduces the papers that are presented in that section and then each paper is described in detail.

## 2.0 INSTRUCTION PREFETCHING METHODS

Instruction prefetching, as the name indicates, is when instructions that will be used in the future are fetched into the instruction cache before they are referenced in order to decrease the memory latency. Seven different instruction prefetching methods are presented here. The first four papers can be categorized as *branch−prediction based instruction prefetching*, because the branch prediction information that is already provided by the existing hardware is utilized to decide which instructions should be prefetched. The next two papers introduce and compare two simple yet effective prefetching methods. The last one is a paper that studies prefetching along both paths of a branch instruction.

## 2.1 Branch Prediction based Instruction Prefetching

### 2.1.1 Instruction Prefetching using Branch Prediction information [19]

Instruction Prefetching can effectively reduce I–cache misses and thus improve performance. In this paper, Mudge et al. employ a branch predictor and a small autonomous fetching unit which runs ahead of the execution unit to predict potentially useful instructions. They use a 4–issue machine with a Branch Predictor, which is shown to have better in performance than a plain cache four times the size, and other hardware schemes such as Look–Ahead PC, next–n line, and wrong path. This scheme makes use of existing advanced branch prediction mechanisms that are already part of the architecture. The Branch Predictor speculatively runs ahead of the execution unit at a rate close to one basic block per cycle.

In *Branch Prediction based prefetching*, the prefetching unit is autonomous and runs down the instruction stream as fast as possible. It decodes the instruction and if branch is encountered, it checks the branch predictor, records the prediction in a log and continues. If the prediction is wrong when checked by the execution unit, the prefetching PC is reset and the history log is flushed. So are the history register and the return address stack. If it is a subroutine branch, it uses the return stack. If it is a conditional branch, it uses a two–level Branch Predictor to predict direction. Unconditional branches are always taken and hence just decoded.

Hardware necessary for this is a Program counter, a branch history register, an adder to compute branch targets, and a return address stack.

Because prefetching is speculative, trace−driven simulation is not enough because it tracks the actual execution path. Therefore, an instruction fetching engine was added, which fetches all the instructions and stores them in an internal data structure. Compared to next−2 line prefetching, this technique showed a 17−32% improvement and a 34−44% improvement over wrong−path instruction prefetching. *Branch Prediction based prefetching* does not impact bus traffic by being selective and accurate and it was shown to be faster than a plain cache four times the size.

Three benefits of *Branch Prediction based prefetching* were pointed out in the paper. It is better than *table based* prediction techniques because of the independent adder to compute target address and also because the prediction is more accurate. Branch Prediction based prefetching is also better than *wrong−path* [18] instruction prefetching which calculates target address at the instruction decode stage.

### 2.1.2. An intelligent instruction cache prefetch mechanism [21]

This paper's contribution is to examine ways to reduce the significant negative impact of branches on multiple issue processors. The authors, Young and Shekita, note that an "intelligent non−sequential prefetch mechanism" is needed to accomplish this.

The authors explain that existing branch prediction techniques are not able to hide the memory latency of an I−cache miss resulting from a taken branch because prediction is typically done at the instruction decode stage which is too late to hide the memory latency of an I−cache miss.

To get around the above problem, they propose basing branch prediction on program flow. The proposal is to do branch prediction on entire blocks of instructions rather than individual instructions. The expectation is that this will be more effective at hiding the memory latency because prediction can be carried out as soon as the first instruction of a flow block is accessed. This proposed mechanism handles both sequential as well as non–sequential prefetch of I–cache lines.

The size of a flow block is chosen such that there is a high probability that each flow block contains at most one branch instruction. In practice, the flow block size would probably be a power of two to facilitate fast lookup. As the flow block size increases, the prediction accuracy worsens because multiple branch predictions can appear in each flow block.

The pieces of information kept in each entry of a flow–block prediction table are the Flow–Block Address Tag, Last Branch Target Address, History and Indirect Branch Bit. The accuracy of flow–block prediction improves with smaller flow blocks but larger flow blocks are more effective at hiding memory latency. Two ways of improving the accuracy of larger flow blocks are discussed in this paper. One way is to store two target addresses in each prediction table entry while the other is to flag branch instructions with a bit that indicates whether the instruction is a branch.

Four commercial relational database management systems were run on an IBM RISC System/6000. Time to execute the benchmarks was presented and it was shown that flow block prefetch performs up to 60% faster than the current RISC System/6000 implementation and up to 20% faster than sequential prefetch. The authors conclude that the relative performance of this mechanism should improve over time as the gap between processor clock rates and memory latency grows and the number of cycles per instruction decreases.

It is noted that flow–block prediction is compatible with many existing branch prediction mechanisms.

### 2.1.3. Instruction cache prefetching with Extended BTB [20]

Chi et al. call the method presented in this paper, Branch Instruction Based (BIB) prefetching. The prefetch information is saved in an extended branch target buffer (eBTB). The eBTB is associated with the instruction fetch stage to determine if the instruction fetched is a branch instruction, and if so, to predict the next instruction address.

A cache structure consisting of an instruction cache and a prefetch buffer is proposed. Each entry in the buffer contains three fields: a tag field, a line field, and a valid bit. The instruction cache and the prefetch buffer are searched in parallel on each instruction fetch. If there is a hit in the prefetch buffer while the cache misses, the requested line can be supplied by the prefetch buffer without any delay.

The BIB prefetching is similar to other target prefetching methods in the way it uses the history information of previous branch targets. It makes effective use of the eBTB mechanism to store prefetching information and to make the prefetching more accurate. The organization of the eBTB is very similar to a conventional branch target buffer except that it has an additional field, the prefetch line address field, to record the prefetching information, which is based on the history of the previous execution.

In this paper, a modified look–ahead prefetching mechanism[9] is also simulated. They call it a lookahead basic–block prefetching mechanism, which prefetches the first line of the basic block previously executed following the eBTB predicted basic block.

Simulations were done on a DECstation 5133 to evaluate performance and the traces were generated using pixie. MCPI is the metric used, which is cycles per instruction contributed by memory accesses. The benchmarks used were go, gcc, compress, li, ijpeg and vortex.

Considering the average MCPI of the six bench marks used, BIB prefetching outperforms the sequential prefetching by 7% and hybrid prefetching by 17% on average (Hybrid prefetching prefetches both the next sequential line and the target line).

The memory traffic generated by BIB prefetching is about 28% higher than by a normal 2KB direct mapped cache. However, they point out that with higher prefetching accuracies, the memory overhead will be lowered or even completely eliminated. They also point out that there is no need to prefetch if the on–chip instruction cache is large and the associativity is high. However, the BIB prefetching method with a small cache size can achieve performance equal to a larger cache. And, it is claimed that it requires a smaller amount of chip area to achieve better performance when compared to other prefetching methods.

### 2.1.4. Instruction Prefetching using multi–level branch prediction[22]

By storing branch history and branch target addresses, this scheme predicts a future PC several branches past the current one. Alexander Veidenbaum, in this paper, describes a possible prefetching architecture and evaluates its accuracy, the impact of the instruction prefetching on performance, and its interaction with sequential prefetching. In this paper, the author shows that their prefetching unit can be more effective than sequential prefetching.

Sequential prefetching typically starts with an I–cache miss address and stops when a branch instruction or an I–cache hit are encountered.  This paper presents an I–cache prefetch mechanism with a longer look–ahead.  "Prefetching" in this paper refers to fetching from a second–level cache to the I–cache ahead of the current program counter (PC).  Sequential prefetching is also studied and its effectiveness and relationship with branch target prefetching is explored.

Their implementation uses a two–level branch predictor because of its natural balance between branch frequencies, average number of consecutive instructions between branches and prediction accuracy. However, they have investigated the 1– and 3– level prediction.  The predictor consists of a Branch History Register, a Predictor Table, and associated control logic.

They compare a baseline architecture with no prefetching to 1)baseline with instruction stream buffer, 2) baseline plus multi–level branch predictor based prefetch unit and 3) baseline plus 1) and 3).  Five benchmarks from the SPEC92 suite are used along with dbm, a widely used UNIX data base manager.

Results indicate that the branch prediction accuracy reaches 90% and above for all benchmarks once a table size of around 1K entries is reached.  For doduc, fpppp, and gcc, the stream buffer is more efficient than multilevel branch prediction in removing cache misses.  For sc and xlisp, the effect of the two organizations is quite close.  The stream buffer sequential prefetching reduces the miss rate by 80–97% for 3 benchmarks, but for the other three it only removes 40% or fewer misses.  Branch target prefetching removes close to an additional 40% of the misses in the latter two benchmarks, while giving 3–11% improvement in the former three benchmarks.  When the two prefetching methods are used together, the effect is very close to additive.  The range of improvement of execution time from stream buffer prefetching is 1.7–10.3%, 1–8% from multi–level branch prediction based prefetching and 5.8–16% for combined prefetching.  The effect is smaller than one might expect from the miss rate reduction as there is a significant stall component in stream buffer prefetching indicating it may not be started early enough and may need more bandwidth.

In some programs sequential prefetching works well and the additional prefetching hardware provides only a small improvement. In other benchmarks, the MLBP–based prefetching produces a larger effect than sequential prefetching. Its success is due in part to the lookahead afforded by the multi–level prediction. The two approaches can be used together to effectively remove both sequential and branch target instruction cache misses.

## 2.2 A performance study of I–cache prefetching methods [17]

This paper is very similar to other papers written on comparing different data cache prefetching methods such as sequential and stride data prefetching [4,5].

This paper studies two instruction prefetching methods called fall–through and target prefetching methods. It also measures the performance of combining these two methods.

The fall–through prefetching strategy fetches the next consecutive line at some time before the end of the current line. This is also referred to as "one block lookahead" in other papers. Target prefetching is directed as branches to nonconsecutive lines, which would be the case with procedure call/returns and branches closing large loops.

The primary performance measure used is the miss ratio and memory traffic is used as a secondary measure. Ten scalar benchmarks are used in their trace simulation.

With fall−through prefetching, miss rates are improved significantly versus not prefetching. For both direct mapped and associative caches, the number of misses is reduced by a third. In contrast to no prefetching, larger line sizes are no longer an advantage. They find that the memory traffic is actually less than that of larger line size caches that perform best without prefetching. They also find that high efficiencies correspond to low miss rates.

With target prefetching, a next−line prediction table is constructed. Each entry in the table consists of a pair – current line address and target line address. When the program counter changes to a new line, the prefetch unit uses the program counter's current line address to search the prediction table. If there is a match, then the target line address from the table is a candidate for prefetching. If the line is not already is the cache, it is prefetched.

Overall performance of target prefetching is much better than no prefetch, and roughly the same as fall−through prefetch depending on the prediction table used. Optimal line sizes tend to be larger than with fall−through prefetch, although there are some exceptions. For smaller caches, the efficiencies are higher than fall−through prefetch. They also find that larger prediction tables are better (as expected).

As mentioned above, the performance of the combination of both fall−through and target prefetching methods are measured. Prefetching on both paths, sequential and target address, is much better than fall−through and target alone. Improvements are found to be almost as good as the sum of the two individual methods.

### 2.3 An improved look−ahead instruction prefetching [16]

Two methods to improve the cache miss ratio and the average access time for successfully prefetched blocks are proposed in this paper by Park et al. First, a method to fetch multiple blocks for a prefetch request is presented. Second, a technique called "prefetch on miss" is adapted to the lookahead prefetching to alleviate the drawbacks of lookahead prefetching. "Prefetch on miss" is when the next sequential block is prefetched together with the block that causes a cache miss when a cache miss is occurred. It is shown that these proposed prefetching methods reduce the memory access delay time by 32 – 56% compared to a cache system with no prefetching.

## 2.4 Wrong path Instruction prefetching [18]

Jim Pierce, one of the two authors of this paper, showed in his dissertation that executing instructions down mispredicted paths can actually reduce the number of cache misses occurring during correct path execution. The name *wrong path prefetching* is given because prefetching is done along mispredicted paths. Wrong path prefetching is similar to the hybrid scheme which combines next–line and target prefetching. The next–line is prefetched whenever instructions are accessed inside the fetchahead distance. However, unlike target prefetching, only target line addresses are saved and no attempt is made to prefetch only the correct execution path. Instead, the line containing the target of a conditional branch is prefetched immediately after the branch instruction is recognized in the decode stage. Thus, both paths of conditional branches are always prefetched: the fall through direction with next–line prefetching and the target path with target prefetching.

Unfortunately, because the target is computed at such a late stage, prefetching the target line when the branch is taken is unproductive. In this case, if the target address is not in the cache, a fetch miss and a prefetch request of the same line will be generated simultaneously. Similarly, prefetching unconditional jump or subroutine call targets is useless since the targets are always taken and the prefetch address would be produced too late.

The hardware requirements for wrong–path prefetching are roughly equivalent to what is required for next–line prefetching since the target prefetch addresses are generated by the existing decoder and no target addresses are saved. The obvious advantage of wrong–path prefetching over the hybrid algorithm is that there is a lower hardware cost.

The authors understand that prefetching down not–taken paths will put some lines into the cache that are never accessed. This will increase both memory traffic and cache pollution. For the algorithm to be successful, the benefits of prefetching must overcome the added misses due to pollution.

A single issue pipelined processor with RISC–like properties was chosen as the base model and the statistics used for comparison are total CPU cycles and miss cycles. Wrong path prefetching performed the best in terms of CPU cycles for all cache sizes, producing up to a 16% speedup over that of no prefetching and 4% over next–line prefetching. They also found that wrong–path prefetching reduces miss cycles by almost 40% in an 8K cache. In a machine with an issue rate of 4, wrong path prefetching improves performance by up to 21%.

## 3.0 DATA CACHE PREFETCHING METHODS

This section presents various data prefetching methods that are described in thirteen different papers. Nine out of these thirteen papers can be categorized broadly into three categories.

Sequential and Stride Prefetching

The first category includes four papers that study sequential and stride prefetching methods. Sequential prefetching is the simplest form of prefetching, where the next consecutive block is brought into the cache. But since data access patterns are not always sequential, the stride prefetching method was developed that supports non–sequential patterns. In addition to two papers dedicated to sequential and stride prefetching each, two other papers that deal with the effectiveness of these papers are also presented.

Branch Prediction based Prefetching

The second category consists of two papers that are based on branch prediction strategies. This is similar to a few papers that were done for instruction cache prefetching. The main difference here is that branch prediction results from the existing system resources are used to decide which pieces of data should be prefetched.

Loops

The third category deals exclusively with data prefetching techniques that are aimed at loops. These methods will improve the performance of applications that are loop oriented (like many scientific applications).

Four other papers that do not fit this categorization are also presented; two of which are about prefetching methods and two, which are intended to provide insight into prefetching. One of the prefetching papers is based on building a markov model to predict which information will be used next, while the other paper proposes a new hardware construct called the Program Progress Graph which is an extension to the Branch Target buffer. Of the other two insight papers, one is a study of the effect of various parameters such as latency, degree of prefetching and network contention of different prefetching methods while the other paper focuses on trying to cache only information that has a high probability of being used.

## 3.1 Sequential and Stride Prefetching

### 3.1.1 Sequential Prefetching [8]

Sequential prefetching had been studied for uniprocessors before, but this paper is aimed at multiprocessors. Also, only sequential prefetching on read misses is considered.

A simple hardware based prefetching technique is to prefetch consecutive blocks of data from main memory into the cache following the block that misses in the cache. Sequential prefetching in its simplest form prefetches a fixed number of blocks on every miss. The number of blocks to be prefetched is called the *degree of prefetching* referred to as K. This is called fixed sequential prefetching. However, in this paper, Dahlgren et al. propose an adaptive scheme that determines the number of blocks to be prefetched based on a dynamic measure of prefetching effectiveness. The main theme of this paper is to evaluate fixed and adaptive sequential prefetching schemes that have low implementation cost.

In sequential prefetching, the amount of spatial locality affects the effectiveness of prefetching. The effects of cache block size variations on miss rate, traffic, and write penalty in multiprocessors is studied. It is obvious that as the block size increases the miss rate will decrease. However, it is pointed out that this is true only for cold misses and true−sharing misses. The number of false sharing misses increase as the block size increases. The overall traffic and the write penalty as a function of the block size follow a U–shaped curve.  A good prefetching scheme should reduce the number of cold and true sharing misses without increasing the number of false sharing misses.

The architecture used in this study includes a processor, a first and second level cache, a first and second level write buffer, a local bus, a network interface controller, and a memory module. Prefetching was restricted to misses in the second–level cache only.

Despite its simplicity, the special case of K=1, sequential prefetching is very effective for applications where the cold and true sharing read misses exhibit a high spatial locality.  However, the invalidation traffic may actually increase, and consequently, the write penalty under sequential consistency may increase.

The optimum degree of prefetching is dictated by the spatial locality of the application and by the block size.  Depending on how far the block size B is from the minimum of the U–shaped curve for the miss rate, a smaller or higher value for K should be used.  If the prefetching degree is too high, the cache is loaded with useless data, resulting in excessive memory traffic.  However, the best value of K varies dynamically during the execution of most benchmarks.  Ideally, K should be adjusted to minimize the true sharing miss component while retaining the same false sharing miss rate.  This is where Adaptive Sequential Prefetching comes in to play.

In the adaptive approach, a register called the Lookahead Counter controls the degree of prefetching. The basic mechanisms used to measure the prefetch efficiency consist of two counters (the PrefetchCounter and the UsefulCounter) and a PrefetchBit per cache line, which are all, cleared initially.  There are two preset thresholds to determine if the Lookahead Counter should be changed. If the UsefulCounter exceeds the upper threshold, the program is in a phase of execution where it could benefit from a higher degree of prefetching and therefore the Look−ahead Counter is incremented. If the Useful Counter is lower than the lower preset threshold, the amount of prefetching is too high and the Lookahead Counter is decremented. When the Lookahead counter reaches zero, prefetching is turned off.

The benchmarks used were MP3D, Water, Cholesky, PTHOR, LU and the multi−grid version of Ocean. Scheduling policies were not studied and the results are derived using the scheduling policies built into the applications.

Fixed Sequential Prefetch of the next consecutive block was observed to exploit the spatial locality of reads as does doubling the block. However, by contrast to a larger block, false sharing miss rate is essentially unaffected by fixed prefetch. Fixed prefetch can also result in less read traffic compared to doubling the block size. The write traffic is comparable between sequential prefetch and doubling the block size.

Applications such as LU, Cholesky, and Water show a high prefetch efficiency. As a result, the number of issued prefetches is much higher for adaptive prefetch. Because of the larger number of issued prefetches, the prefetch efficiency is decreased for the adaptive scheme compared to fixed prefetch. Applications with low spatial locality of misses (such as Ocean and PTHOR) were shown to have low prefetch efficiency, and consequently the number of prefetches issued under adaptive prefetch was much lower compared to fixed prefetch. The total traffic and the number of read misses were shown to be inversely proportional. A decrease (increase) in the number of read misses leads to increased (decreased) traffic.

The simulations show a considerable reduction of read misses and of the read stall time for those applications where the spatial locality of read misses is high and show little added traffic for those applications where the spatial locality of misses is small.

For some applications, the total execution time is reduced up to 25% with fixed and adaptive prefetch. Both fixed and adaptive prefetch methods show a significant reduction in the number of replacement misses for most applications.

They conclude that considering the substantial performance improvement provided by adaptive sequential prefetching, the extra hardware complexity needed seems to be justified.

### 3.1.2 Stride Prefetching for secondary caches [10]

This paper shows how existing stride−directed methods for L1 caches do not work as well in L2 caches because of the difference in characteristics of L1 and L2 caches. Veidenbaum et al. propose a new stride−detection mechanism for L2 prefetching and combine it with stream buffers[15]. The authors show that this new prefetching scheme is more effective than using stream buffers for prefetching, particularly for application with long−stride accesses. They also evaluate an L2 cache prefetching organization that combines a small L2 cache with their stride−directed prefetching scheme. Their results show that this system performs significantly better than stream buffer prefetching or a larger non−prefetching L2 cache without suffering from a significant increase in the memory traffic.

Scientific applications often have high miss rates and result in poor cache performance due to longer access times. Though several hardware and software solutions have been proposed and implemented to tolerate these long latencies, this paper focuses on hardware based data cache prefetching for secondary caches only. Kim and Veidenbaum indicate that stride–directed prefetching is much more appropriate for scientific applications. They point out that there are several reasons for prefetching only at the secondary level. First, system designers have an opportunity to add prefetching hardware to an off–chip secondary cache which may not be possible in an on–chip primary cache. In addition, prefetching logic added to a primary cache increases its complexity and may increase processor cycle time. Finally, a larger L2 cache can tolerate more cache pollution caused by prefetching than a small primary cache can.

A new stride detection method for L2 caches called loop–based prefetching (LBP) is proposed. This scheme does not need instruction addresses to identify a memory reference stream and requires only modest hardware. In fact, the goal of LBP is to not use instruction addresses to detect memory references. Stride detection and prediction is performed when LBP is used in conjunction with stride calculation units. LBP incurs only a modest hardware increase compared to other proposed methods.

Memory references with a constant stride are usually generated by load/store instructions that are repeatedly executed in a loop. If a start of an iteration of the loop can be detected, it is possible to identify a memory reference stream by its appearance relative to the start of an iteration. The Reference Prediction table (RPT) is used to hold the load instruction address, the stride that has been calculated for that instruction, and the prefetch address. A counter is used to identify a stream. The implementation is handled using a stride detection unit consisting of a RPT index counter, a comparator, an address adder, and a stride calculator. Trace–driven simulation was done for performance evaluation. Benchmarks used were APPBT, APPSP, ARC3D, CG, DYFESM, FLO52, MDG, MGRID, OCEAN, and TRFD. All are computationally intensive and usually exhibit large data set sizes.

Loop–based prefetching is compared to stream buffers and to Instruction–address prefetching. Stream buffers and a miss table unit was chosen to be the baseline architecture and is called MTP. It was chosen because it is simple, does not use memory reference stream and stride detection, and has been shown to be effective in L2 prefetching in prior work. The loop–based detection unit is added to MTP to detect long stride memory accesses. The detection unit only sees L1 misses that miss in both stream buffers and the miss table unit. This is referred to as the Loop–based prefetching (LBP) architecture. Finally, the instruction–address–based prefetching (IAP) architecture uses instruction addresses that are sent to the Reference Prediction Table[1] for stride detection.

Results indicate that stride–directed prefetching may not be as effective at L2 as at L1, even if memory reference streams are accurately detected. The effect of the number of stream buffers on performance for the different methods was also studied. As the number of stream buffers increased, the stream buffer miss rate steadily decreased. However, increasing the number of stream buffers is beneficial because it provides more storage for prefetched data and thus the opportunity for reuse. In addition, it allows more opportunities to initiate prefetches. It is pointed out that prefetching methods, which can take advantage of good cross–stream spatial locality, should be used for L2 prefetching.

A small cache was added to prefetching hardware and was compared to a traditional cache, stream buffer prefetching without a cache, and prefetching with a small cache. It was found that the addition of even a small L2 cache improves the performance of all benchmarks significantly in more than half the cases.

It was also found for most benchmarks that L2 caches need to be two to four times larger than the cache used with LBP to match the hit rate. The one benchmark where this was not true was because

LBP suffers from L2 cache conflict misses. This, however, was improved when the associativity of the L2 cache was increased.

Lower miss rates and a relatively small memory traffic increase makes LBP with a small cache the most efficient architecture among the various L2 memory organizations they have studied.

### 3.1.3 An effective programmable prefetch engine for on−chip caches (Hare)[1]

This paper proposes a prefetching scheme that exploits both software and hardware techniques. A new on−chip prefetching scheme called Hare, which is a programmable engine controlled by the user instructions, is introduced. Prefetching is initiated by the hardware on a match with the PC of the processor at run−time. The engine always launches prefetches ahead of current execution, which is controlled by the PC. The engine is only a support unit to the data cache and does not add extra overhead to the critical path of the instruction execution.

The extra support unit is built besides the normal ALU. A Run Ahead Table (RAT), controllable by the user program, used to trigger prefetch action. The RAT can include the target address, stride of the corresponding references, and other control parameters. There is a fill_run_ahead instruction which is usually executed before a loop execution so that the firing of prefetch streams can be prepared in advance. When the prefetched block is returned from the next level of memory hierarchy, the data is put directly into the data cache as opposed to buffers, queues or register files. This way, additional hardware costs and the danger of adding cycle time can be avoided. The solution in Hare is to schedule the firing of prefetches one memory latency time ahead. This is the key to this prefetching method – prefetching should be activated far ahead of actual execution to determine what values need to go in to the RAT (and hence the name Hare).

Good compile time strategies are also key to the success of Hare. Compiler support consists of tasks that are similar to techniques used in software prefetching such as loop unrolling and software pipelining. But this method does not need to unroll loops to increase the prefetch distance as the fill_run_ahead instructions are used to represent prefetch predicates.

Since there is no compiler optimization used with this method, prefetch instructions are added to the address traces. The candidates for prefetches are those references with constant strides among consecutive executions of the same instructions. Whenever the stride accesses has been changed since the RAT should be modified or reset, the assumption is that a fill_run_ahead instruction is required. Otherwise a normal prefetch is initiated by the Hare engine without incurring any prefetch instruction overhead.

The experiments were run with three design choices – a normal on−chip data cache without prefetching(*no prefetching*), a data cache with software prefetching(*software prefetching*) and an on−chip data cache with the prefetch engine, *Hare*. The results of the experiments were evaluated with a metric called MCPI, which stands for cycles per instruction contributed by memory accesses.

It is expected that increasing the cache size will significantly reduce the access penalty, especially for applications have a significant fraction of references with a fixed non−unit stride. As expected, the access penalty for these applications by using *Hare* is observed to have decreased by over 80% which is significantly better than the *no prefetching* and *the software prefetching* design choices. However, applications where most cache misses are due to conflict misses were found to be an exception to the above behavior. In addition to the access penalty, the authors claim that the performance can be further enhanced by reducing the prefetch overhead.

The cost effectiveness of this proposed architecture is also evaluated by considering chip area and cycle time. When prefetching is employed, the use of a chip area equivalent to a small cache can achieve significantly good performance. The hardware complexity of the RAT is relatively costly for small cache sizes (1–2KB); however, the cost becomes comparatively less significant when larger cache sizes are employed.

3.1.4 Evaluation and effectiveness of Sequential and Stride prefetching [4,5]

The *effectiveness* paper was published in 1995 while the *evaluation* paper was published in 1996, both by Dahlgren and Senstrom.

Sequential Prefetching exploits spatial locality across block boundaries while stride prefetching detects and prefetches blocks associated with strides only. A notable difference between the two is the additional amount of hardware support needed for stride prefetching; Sequential prefetching needs no detection mechanism and hence is less complex.

Dahlgren and Senstrom separate stride prefetching into two phases – detection and prefetching phases. The detection phase, as the term indicates, refers to the detection of a stream that has constant stride whereas the prefetching phase is where the actual data is brought in from memory.

The detection phase is further classified into two types I–detection and D–detection. I–detection is where the Prefetcher relies on the instruction address stored in the Reference Prediction Table[1] to detect constant strides. D–detection, however, does not rely on instruction addresses but analyses the data accesses to determine if there is a constant stride. A shortcoming of the I−detection schemes is that it relies on the availability of the program counter of the load instruction that causes the miss. D−detection schemes do not need this.

The prefetching phase can be classified into either a fixed method or an alive method. Fixed is when a certain fixed number of blocks are prefetched according to the degree of prefetching d. This will result in a new miss if the stride sequence is longer than d. If the first prefetched data is a hit, then in the alive method it will be replaced by the (d+1)th block, so that d is adjusted real−time. There will be no read misses as long as the processor accesses the same stride sequence.

Overall, the results show that despite its simpler hardware mechanisms, sequential prefetching is more effective at reducing the number of read misses, while the I−detection scheme in general has a higher prefetch efficiency because it is more selective in the detection phase. . They found that the most common stride is 1 block (76% of all stride accesses) which means that most of the stride−access misses are also covered by sequential prefetching. . They found that the most common stride is 1 block (76% of all stride accesses) which means that most of the stride−access misses are also covered by sequential prefetching. Sequential prefetching does better because most strides are one block, which means sequential prefetching is as effective for most stride sequences as stride prefetching and it also exploits the locality of read misses for nonstride accesses. However, where the memory system bandwidth is a limitation, stride prefetching can perform better than sequential prefetching because of the lower fraction of useless prefetches.

## 3.2 Branch Prediction based Data Cache Prefetching

### 3.2.1 Branch directed and Stride−based data cache prefetching[3]

Since branch instructions determine which instruction path is followed, data access patterns are also dependent upon branch behavior.

Branch directed prefetching attempts to incorporate data prefetching with instruction prefetching techniques. It associates data references with branch instructions.

Branch directed prefetching was first presented by Kaeli et al. in 1994. The algorithm associates data prefetching with branch instructions. With a highly accurate branch predictor, this prefetching algorithm will prefetch useful data. Three advantages of branch directed prefetching are 1) no extra table needed, as in stride prefetching as the Branch Target Buffer can be used 2) the BTB can have fewer unique entries compared to an RPT, which saves tag area, and 3) both conditional and unconditional branches can be considered.

A 2−bit counter produces an 85% correct prediction rate for the benchmark programs which were used. To meet the objective of prefetching only useful data into the data cache, another Finite State Machine was designed to decide whether stride prefetching should be used.

Data hit rates for no prefetching, tagged prefetching and branch directed prefetching using 1, 2, and 3 data entries were compared. The results for Gcc, Xlisp, Eqntott, Sc and Espresso show that both tagged prefetching and branch directed prefetching improve the data cache hit ratio. Branch directed prefetching is generally as good as, if not better than tagged prefetching. A major advantage of tagged prefetching is that it does not require a BTB to store extra data. For branch directed prefetching, the major advantage is that it can handle changing stride−based access patterns which can not be prefetched efficiently using tagged prefetching. Also, branch−directed prefetching produces less cache pollution.

For 2−way, 4−way and 8−way set associative data caches, branch directed and tagged prefetching provided nearly the same data cache hit ratio. If the correct prediction rate was increased by using a more sophisticated prediction mechanism like a two−level predictor, the authors found that the accuracy of branch directed prefetching should be even better. They also found that BTBs with 2 data entries have better performance than the BTBs with a single data entry.

While tagged prefetching provides improved cache hit rates by assuming a sequential data stream, the amount of bus traffic introduced is substantial. The same high hit rates using branch directed prefetching can be enjoyed while considerably reducing the amount of bus traffic from 20−50%.

The effect of combining branch directed prefetching and tagged prefetching was also considered. The main benefit of combining the two is that each scheme can compensate for the other's weakness. Tagged prefetching takes advantage of spatial locality, while branch directed prefetching exploits both spatial (stride) and temporal (loop−related) locality. Results of combining the two indicate that cache hit rates can be improved up to 2.6% in addition to when each method is used alone. The hit rates were around 94% using either method, so combining the two methods essentially provides a cache hit rate of up to 96.6%.

### 3.2.2 Branch directed and Pointer−based data cache prefetching[2]

This paper is a follow−on to "Branch directed and Stride based data cache prefetching" by Kaeli and Liu. In this paper, Pointer−based prefetching is added to their previous work [3] which attempts to identify when accesses are being made to dynamically allocated data structures [e.g. linked lists]. By recording access patterns to these structures, it is possible to effectively prefetch the next element prior to access.

Simulation results indicate that even using a simple greedy pointer–based prefetching scheme significantly reduces the number of data cache misses. As associativity is increased, pointer–based prefetching continues to provide an advantage, due to the increased associativity being able to tolerate the cache pollution introduced by pointer–based prefetching.

In addition to pointer–based prefetching, they have further investigated the benefits of combining tagged prefetching and branch–directed based prefetching and have explored some optimizations for this combination.

## 3.3 Data Cache Prefetching for Loops

### 3.3.1 Hardware driven prefetching for pointer data references[14]

Chi and Cheung, in this paper, propose a fresh look into the data access regularity and predictability from the instructions that use the same register for their source and destination. It is found that most pointer index registers used in memory access instructions in loops are updated by self modifying instructions, each with one source operand used in the address calculation equal to the destination register of the same instruction. Based on this observation, they propose a hardware driven prefetch scheme, called Reference Value Prediction Caching (RVPC). RVPC is based on the detection of self–modifying instructions in loops and the use of self–modifying registers in data references. RVPC can handle not only array accesses with constant strides, but also pointer references with data structures such as linked lists and tree structures.

The authors describe the three steps to develop an efficient RVPC scheme. The first step is to identify registers that are used as the index pointers to those data references with access regularity. The second step is to predict accurately the values of these index registers in the (i+1)th loop iteration when the current (i)th iteration is being executed. The third step is to develop a mechanism to trigger the next iteration loop prefetching for data references that use the index registers found in the first two steps.

For a loop level i, where $0<i<n$, a register is said to be predictable or unpredictable depending on whether its next iteration value can be estimated by the RVPC scheme. Predictable registers are further classified into three categories: loop invariant, self–modifying and dependent–modifying. A self–modifying register r is said to exist at loop level i if there exists only one instruction I with destination register equal to r in the loop at level less than or equal to i.

The RVPC scheme is to issue prefetch requests for data that are expected to be used in the (i+1)th loop iteration when the (i)th iteration is being executed. To achieve this goal, a hardware table, called the Register Value Prediction Table (RVPT), is used to keep track and store the predicted value of each register. When a LOAD/STORE instruction is decoded, the RVPT is checked to see if any prefetch request should be issued. The decision of issuing prefetch requests is mainly based on the availability of the predicted register values.

Benchmark programs with linear as well as non–liner array accesses were used to evaluate the effectiveness of the RVPC scheme. Standard benchmarks were used as well as the authors developed their own programs based on database applications and programs that traverse data with complicated data structures.

Data references whose memory addresses can be predicted accurately by the RVPC scheme in the ideal situation were analyzed. They found that these data references fell into five categories: (1) scalar reference or pointer references with zero strides (2) pointer references with constant strides (3) pointer references with non–constant single ALU stride updating function, (4) pointer references with non–

constant single LOAD stride updating function, and (5) other pointer references with non–constant stride updating functions that do not fall into (3) or (4).

They classify the benchmarks they use based on the above information and find that because over 95% of the data references in scientific computation belong to zero stride, most highly selective prefetching schemes are very effective in generating accurate prefetch requests for most data references in these applications. They also find that with query processing benchmarks, more than 60% of the data references have non–constant strides, which places and upper limit to the performance gain achievable by current highly selective schemes such as software prefetching and hardware prefetching techniques like RPT [Chen].

Since RVPC can handle both constant and non–constant strides, the prefetch coverage is about 80–90% of total data references as compared to about 30–40% of other highly selective schemes. And, for array data structures, the updating function of data pointers is so complicated that about 50% of the data references cannot be predicted by any of current prefetch schemes, including RVPC.

The RVPC scheme is compared to three other schemes – no prefetch, prefetch–on–miss, and Chen's RPT scheme. Among these three, only RVPC can achieve significant performance improvement in applications with both constant stride and non–constant stride data accesses. This shows the larger prefetch coverage of the scheme over a wide range of data structures.

The authors, Chi and Cheung, investigate the effectiveness of the chosen schemes on constant stride and non–constant data access applications. With non–constant stride data access application programs, they find that the MCPI is successfully reduced by 25% to 96% using the RVPC scheme while the RPT scheme achieves 5% average improvement. With constant–stride applications, both the

RPT and the RVPC schemes achieve very good performance improvement. However, the MCPI penalty of the RVPC scheme is slightly larger than that of the RPT scheme. So, in summary, the RVPC scheme works well especially in applications where pointer references are dominant.

### 3.3.2 Effective Hardware–Based Data Prefetching for High Performance Processors[9]

The intent of this paper is to demonstrate that a simple hardware assist, on–chip, can reap important benefits in reducing the data access penalty between an on–chip data cache and the next level in the memory hierarchy. This is significant because a miss in the cache and a hit in the main memory has a penalty of roughly an order of magnitude.

The authors describe three variations of a basic design that keeps track of data access patterns in a Reference prediction table. The three variations differ mostly on the timing of the prefetching. In the simplest scheme, called *basic*, prefetches can be generated one iteration ahead of actual use. The *lookahead* variation takes advantage of a lookahead program counter that stays as much as possible one memory latency time (i.e. potentially several iterations). The *correlated* scheme uses a more sophisticated design to detect patterns across loop levels.

The common basis of these schemes is to predict the instruction execution stream and the data access patterns far enough in advance that the required data can be prefetched and be in the cache when the "real" memory access instruction is executed. This is accomplished through an RPT that holds data access patterns of load/store instructions. The RPT is organized as an instruction cache. Minimally, each entry in the table will contain a tag related to the instruction address, fields to record the memory operand address and its stride, and a state transition field. It is suggested that this scheme will be most appropriate for high performance processors with small first–level caches with a small block size where software–directed approaches may have significant overhead.

The basic mechanism in the RPT is to record the effective address of the memory operand, compute the stride for that access, and set a state controlling the prefetching by comparing the previously recorded stride with the one just computed. The state can be one of four – initial, transient, steady and no prediction. Initial is obvious, transient is when the system is unsure of the previous prediction, and steady refers to when the same stride has been detected for twice in a row.

While the basic scheme prefetches one loop iteration ahead, the Look–ahead scheme prefetches one memory latency time ahead. This is because if the loop body is too small, the prefetched data may arrive too late for the next access, and if the loop body is too large, early arrival of prefetched data may cause pollution. The lookahead prediction is done with the help of a pseudo program counter called the Look–Ahead Program Counter that will remain as much as possible one memory latency time ahead of the regular PC. The LA–PC will access the RPT to generate prefetches and is incremented as the regular PC. It is used with a Branch–Prediction Table to take full advantage of the look–ahead feature similar to other work that has been done with instruction prefetching. However, unlike the instruction prefetch structure, the LA–PC does not need to decode the predicted instruction stream. Instead, the Lookahead mechanism is based on the history information of the execution stream, since the LA–PC is just a pointer to detect the prefetching in the RPT. Another difference between the basic and the lookahead schemes is that the RPT in the LA–PC needs to be dual–ported, to allow simultaneous accesses of PC and LA–PC.

In the previous two designs, reference prediction was based on the regularity between adjacent data accesses. These schemes work well for predicting references in inner loops. However, the results are less significant for those execution segments with small inner–loop bodies or triangle shaped loop patterns because of the frequent stride change in the outer iterations. The key behind the third variation, called *correlated reference prediction,* is to keep track not only of those adjacent accesses in inner loop, but also of those correlated by changes in the loop level. Since branches in the inner loop are taken until the last iteration, a non–taken branch will trigger correlation to the next level up. The implementation adds a shift register to record the outcome of the last branches and an extended RPT with separate fields for computing the strides of the various correlated accesses. Correlation is restricted to two–level nested loops to avoid cache pollution.

All three proposed architectures were evaluated by using cycle–by–cycle trace generation combined with on–the–fly simulation. For comparison purposes, a baseline architecture consisting of a processor with perfect pipelining and direct–mapped data cache is also simulated. Ten applications from the SPEC benchmark suite were used and they were also instrumented using pixie. The applications were Tomcatv, Fpppp, Matrix, Spice, Doduc, Nasa, Eqntott, Espresso, Gcc, and Xlisp. The main metric used was MCPI.

The results show that the prefetching organizations are always superior to the pure cache scheme since they have the same amount of cache and, in addition, the prefetching component. When the cache is too small to contain the working set of the application, the best prefetching scheme can reduce the data access penalty from 16% to 97%. The data access penalty is reduced by over 90% in Matrix and Espresso, 35–70% in Tomcatv, Nasa, Eqntott, and Xlisp and about 30% in Spice, Doduc, Gcc, and Fpppp. The additional cost (RPT and logic) is approximately equivalent to a 4Kbyte D–cache. This additional cost paid for prefetching is justified by the significant performance improvement. However, the difference between the *lookahead* and *correlated* variations is always small, less than 2%. The difference between *lookahead* and basic is more significant. This shows that the *lookahead* logic is worth implementing, since it provides the flexibility to prefetch at the correct time while the complexity required to help data accesses in outer loops (as in the correlated schemes) plays a much less significant role.

The authors summarize their results as follows: 1) the hardware prefetching schemes all yield significant reductions in the data access penalty when compared with regular caches, 2) the benefits are greater when the hardware assist augments small on−chip caches, and 3) the look ahead scheme is preferred for cost−performance reasons. The authors also point out that when the latencies are two orders of magnitude larger than the processor cycle time, prefetching data by software−directed techniques may be more beneficial.

### 3.3.3 Sunder: for Numerical Loops[6]

This paper by Chiueh is a proposal for a new architecture that combines the flexibility and accurateness of s/w prefetching and the transparency and low−overhead of hardware prefetching. They call their architecture 'Sunder', which has a dedicated prefetch engine that is programmable at run time by the software. The prefetch engine is completely isolated from the normal instruction execution pipeline except for a loop counter to keep the two synchronized at the boundaries of loop iterations.

Sunder is designed specifically for array references in scientific programs. In particular, the focus is on array references in the inner loops. Sunder is for both Instruction and Data prefetching, unlike most other schemes that are targeted at either instruction or data. Sunder executes prefetch operations completely independently of normal instruction execution, therefore there is no extra overhead as in software prefetching. Compared to hardware prefetching, there is no need for address prediction because the memory addresses of target data words are gathered through compile−time analysis. Also, since Sunder decouples prefetching hardware from the instruction execution pipeline, it reduces the potential performance impact on processor cycle time to the minimum.

A detailed simulation study was done by the authors, based on several complicated loops in the Livermore loop suite. Sunder achieves an average relative performance advantage in the range of 28−46% over cache−only architectures, depending on the cache block size.

### 3.4 Markov Predictors[11]

This paper, by Grunwald et al., presents the Markov Prefetcher and simulates its performance over a number of significant applications.

The Markov Prefetcher acts as an interface between the on−chip and off−chip cache, and can be added to existing computer designs. It is different from other prefetching methods by prefetching multiple reference predictions from the memory subsystem and then prioritizing the delivery of those references to the processor.

The Markov Prefetcher uses the miss address stream from the on−chip L1 cache as a prediction source. Based on this miss address stream, a Markov model is built that approximates the reference string using a transition frequency which indicates the percentage of times a particular sequence occurs. In a way, it is similar to (and an extension of) correlation based prefetching because of the use of previous reference patterns to predict the future occurrences.

Three factors that influence the performance of a prefetcher are considered. They are coverage, accuracy and timeliness. Benchmarks used are Sdet, Ladis, Netperf, TPCB, Abaqus, G92, MM4, and Spice. The performance of the Markov prefetcher was compared to other prefetch mechanisms (namely stream prefetching, stride prefetching and correlation prefetching).

The authors point out that even though this could be considered an extension of some of the prior work on correlation based prefetching, there are a number of unique design issues that the Markov Predictor presents. The Prefetcher must be able to launch multiple prefetch requests and prioritize them, and must have some mechanism to limit the bandwidth devoted to prefetching.

This study shows that a simple, effective and realizable markov prefetcher can be built as an off–chip component of the memory subsystems. This design requires fewer resources than a normal demand–fetch cache, yet reduced the memory CPI by an average of 54% over the applications examined. The Markov prefetchers examined provide better prefetch coverage and more timely prefetches than other prefetchers at the expense of reduced accuracy. However, combination of bandwidth reduction filters and prioritization of the prefetch requests makes the Markov Prefetcher the most effective prefetcher examined.

The Markov prefetcher could be improved by having the ability to abstract a reference pattern from several instances of reference sequences. This is because both the correlation and Markov prefetchers must observe a reference sequence prior to predicting references. Also, it is possible to have a mechanism to issue prefetch requests for patterns or sequences that have not yet occurred, eliminating the training period needed by the Markov preditor.

### 3.5 Tango[7]

This paper is basically an extension of [1] to superscalar processors. They highly recommend the hardware–based technique instead of the s/w based one as it may increase the length of the critical path by inserting extra prefetch instructions.

Pinter and Yoaz propose a new hardware construct called the Program Progress Graph which is an extension of the Branch Target buffer. The PPG is used to implement a fast pre–program counter, pre–PC, that travels only through memory reference intructions, rather than scanning all the instructions manually. In a single clock cycle, the pre–PC extracts all the predicted memory references in some future block of instructions to obtain early data prefetching. The prefetch requests are scheduled to “tango” with the core requests from the data cache, by using only free time slots on the existing data cache tag ports. They also reduce the number of prefetch requests sent to the core–cache bus and to the memory bus by removing prefetch requests that are already in the cache and a simple optimization on the cache LRU mechanism.

The scheme comprises three functional parts – a special lookahead mechanism, a mechanism to generate data access predictions, and a mechanism for filtering out redundant prefetch requests.

Simulation results on the SPEC92 benchmark show an average speedup of 1.36 (CPI ratio). Detailed results are discussed along with the effect of data cache size, associativity, and data cache line size. Different Tango parameters are also examined.

### 3.6 Tolerating medium latencies on data caches with hardware−based prefetching[12]

In this paper, Moreno et al. analyze a single−bus multiprocessor using a stochastic timed petri net (STPN) model to study the effect of various parameters such as latency, degree of prefetching and network contention on different prefetching methods. Using STPN modeling, they are able to conclude that fixed sequential prefetching with a degree of prefetching of 4 would improve the average speed−up for medium latencies, which they define as 64 processor cycles on a 80MHz processor. Beyond 4, the system performance is shown to degrade.

They claim that the main contributions of this paper are 1) proposing one simple model of petri nets to execute prefetching on a bus−based multiprocessor as opposed to the evaluation methodologies of execution/program/trace driven simulation used in other work, 2) analyzing the effect of the medium latencies on the system when prefetching is implemented, and 3) showing that fixed sequential prefetching is useful for improving the performance of the system.

### 3.7 A modified approach to data cache management[13]

The techniques described in this paper to reduce the average data access time are largely orthogonal to stanadard miss rate/miss penalty reduction techniques. Tyson et al. examine the benefit of deciding whether to cache a particular data item based on the address of the load instruction generating the request.

It has been pointed out in earlier research [26] that a large percentage of data cache misses are caused by a very small number of instructions. Using this information, if the data cache is smarter about what it decides to cache and does not allow these troublesome instructions to allocate space in the cache, the data cache miss rate can be improved. The miss rate associated with individual load and store instructions were measured using the ATOM program trace facilities and the SPEC92 suite of benchmarks. They find that, for the compress benchmark, on average 1.77% of the load/store instructions account for 95% of the data references. This, they indicate, is an effect of the 90/10 locality rule which states that approximately 90% of execution time is spent in only 10% of the code.

One of their proposed methods is called the static method, where all load instructions that do not meet a threshold for cache hit rate are marked as C/NA (Cacheable/Not Allocatable). They find that when setting the threshold at 75%, one of the SPEC92 benchmarks run on a 16K−byte 2−way set associative data cache requires 61.62% less bandwidth than that required by the same program run on the same hardware without the C/NA transformations.

In the improved static method, only instructions that do not perform a useful prefetch are marked C/NA. Once the C/NA instructions were identified as in the previous static method, they were analyzed to determine if they were performing a useful prefetch. If at least ¾ of the misses prove to be prefetches, then the instruction was removed from the C/NA list, resulting in a less aggressive application of C/NA.

With this improved static method, the total change in memory activity is only 0.96% compared to the static method described above. This shows that by applying the improved static method to a program, the cache hit rates can be maintained while simultaneously decreasing the amount of traffic to memory.

The static approach requires training runs of the program, and the introduction of new instruction in order to specify the alternate cache behavior. Both these factors markedly decrease the applicability of the static approach and hence, the dynamic cache model is proposed. The goal with the dynamic cache model is to develop a scheme that will provide the same performance enhancement as the static model transparently. This was done by simulating miss prediction tables using a 2−bit counter associated with each load instruction. A miss prediction counter is initially set to zero and it is incremented each time a load causes a cache. When the counter enters its highest state ("11"), the instruction is marked C/NA. From running experiments using this method, it was found that the number of instructions that spend some amount of time in the C/NA state is much larger than for either of the static methods. Further comparisons between the static results and dynamic results indicate that, as one might expect, the 2−bit dynamic scheme increased the average number of memory references by 92.15%.

Since the 92% increase in memory references indicates that the scheme is marking an instruction as C/NA too quickly, the 2−bit scheme was modified to mimic the Improved Static scheme. Results indicate that the number of memory references were reduced such that there is actually a 1.36% decrease compared to a conventional cache. It was suggested that because of the small change in the number of memory references, this improved strategy is too conservative in marking instructions whose data should not be cached.

In essence, this study indicates that the amount of memory activity and the required memory bandwidth could be substantially reduced by not caching all data items.

## 4.0 STREAM BUFFERS

This section presents a set of papers that propose using buffers that are close to the processor and the cache to store the data that is prefetched, instead of taking up valuable space in the cache. These buffers are called stream buffers and they can be organized just like a cache to enable easy look−up.

Stream buffers are classified as a separate section because they apply to both instruction and data cache prefetching.

4.1 Improving direct mapped cache performance by the addition of a small fully−associative cache and Stream buffers (both instruction and data) [23]

A stream buffer consists of a series of entries, each consisting of a tag, unavailable bit, and a data line. When a miss occurs, the stream buffer begins prefetching successive lines starting at the miss target. These lines are placed in the stream buffer and not the cache, and thus, avoids polluting the cache with data that may never be used.  Subsequent accesses to the cache compare their address against the first item stored in the buffer. If a reference misses in the cache but hits in the buffer the cache can be reloaded in a single cycle from the stream buffer. When a line is moved from the stream buffer into the cache, the entries in the stream buffer can shift up by one and a new successive address is fetched.

Stream buffers were able to remove 72% of the instruction cache misses, but only 25% of the data cache misses. To improve this performance, multi−way stream buffers which consist of four stream buffers in parallel were simulated. When a miss occurs in the data cache that does not hit in any stream buffer, the stream buffer hit least recently is cleared and it starts fetching at the miss address.

The performance of the instruction stream buffer remains virtually unchanged, which indicates that the simpler single stream buffer will suffice for instruction streams. However, the multi−way stream buffer can remove 43% of the data cache misses for the six benchmarks used.

The effects of cache size and cache line size were also studied. The data stream buffer performance increased from a 15% reduction in misses for a data cache size of 1KB to a 35% reduction in misses for a data cache size of 128KB.  The reduction in misses provided by a single data stream buffer falls by a factor of 6.8 going from a line size of 8B to a line size of 128B, while a 4−way stream buffer's contribution fully by a factor of 4.5.

4.2 Retrospective [24]

Stream buffers were proposed as a method for reducing primarily compulsory and capacity misses. They essentially make the cache look bigger. The net result was a memory system with higher performance for a given transistor count. In this paper, Jouppi talks about four enhancements to his previous paper [23] on stream buffers.

1. Jouppi presents an enhancement to stream buffers which could supply prefetch data from any position in the stream buffer, resulting in better utilization of data in the stream buffer and higher stream buffer hit rates.
2. Jouppi points out that stream buffers were more effective than any of the hardware prefetching techniques that had been discussed in other papers.
3. Stream buffers were more effective in reducing cache misses than victim caches [25]
4. Adding multiple stream buffers to a baseline cache design increases effective cache size.

Stream buffers have appeared in a number of systems (HP PA7100 microprocessor, Cray T3D and T3E), although their use is by no means wide spread.  The most compelling application remains prefetching instruction into stream buffers due to the highly sequential nature of instruction miss streams.

Jouppi believes that stream buffers are an important and lasting contribution  to system performance. He also believes that as technology integration increases, stream buffers will still be important for obtaining the best system performance for ever limited cache resources and that there is still more room for further advances in hardware prefetching.

4.3 How useful are stream buffers in multiple issue processors[15]

The stream buffer model used in this paper by Jouppi et al. is the one described in [23]. However, four enhancements to the above are made. This paper deals with more than just stream buffers as the title indicates, but only stream buffers are presented here. The enhancements are allocation filters, hardware support for dynamic strides, enforcement of non−overlapping prefetch paths, and direct access to the stream buffer entries.

The allocation filter prevents a stream buffer from being allocated until two misses occur for the same stream. On the second miss, a stream buffer is allocated and it begins prefetching the block subsequent to the one corresponding to the second miss. Hardware support for dynamic strides is provided, as opposed to the unit stride of the original proposal. And, when there are multiple stream buffers, they ensure that a given block of data resides in at most one stream buffer. Finally, the original stream buffer proposal was extended to include direct access to non−head entries in the queues.

Benchmark results show that unit−stride stream buffers provide a performance improvement of up to18% while non−blocking loads provide 17%. With the addition of allocation filters and dynamic strides, the CPI improvement for stream buffers is 31% and non−blocking loads is 43%. The authors find that non−unit−stride stream buffers provide better performance than either unit−stride stream buffers or non−blocking loads. And, non−blocking loads and stream buffers together provide better performance than either one used alone. They also find that for the large miss penalty, non−blocking loads have smaller improvement factors than stream buffers. With smaller miss penalties, non–blocking loads are relatively more effective.

## 5.0 CONCLUSION

Prefetching is an active research field and there is a lot of room for additional research. Currently, prefetching is widely used in many processor designs. And, since the memory latency problem is not going away (no apparent leap in physical memory technology apparent), I believe that prefetching will continue to be an effective method.

The variety of the papers studied as part of this project show that when it comes to prefetching, "one size does not fit all". Specific applications benefit from a given prefetching technique. It is up to the user to evaluate the effectiveness of each technique and use it according to his needs.

In a few years, the industry is approaching processors that run at 1GHz. However, the memory is technologically limited. There are plenty of opportunities for further enhancements to already published work. But, more apparent is the need for clear breakthroughs to reduce the ever–increasing memory latency.

## 6.0 BIBLIOGRAPHY


1. T–F.Chen. *Effective programmable prefetch engine for on–chip caches*. Proceedings of the 28th annual international symposium on microarchitecture, page 237.

2. M. Dimitri, Y. Liu and D. Kaeli. *Branch directed and Pointer–based data cache prefetching.* Journal of Systems Architecture, to appear in the special issue on Microprocessor architectures, January 1999.

3. Y. Liu and D. Kaeli. *Branch directed and stride based data cache prefetching*. Proceedings of International conference on Computer Design, page 225, October 1996.

4. F. Dahlgren and P. Senstrom. *Evaluation of Hardware–based stride and sequential prefetching in shared–memory multiprocessors*. IEEE transactions on parallel and distributed systems, page 385, April 1996.

5. F. Dahlgren and P. Senstrom. *Effectiveness of hardware–based stride and sequential prefetching in shared memory multiprocessors*, Proceedings of 1st IEEE Symposium on High Performance computer Architecture, page 68, January 1995.

6. T–C.Chiueh. Sunder*: A Programmable hardware prefetch architecture for numerical loops*. Proceedings of Supercomputing, page 488, November 1994.

7. S. S. Pinter and A. Yoaz. Tango*: A hardware–based data prefetching technique for superscalar processors*. Proceedings of annual international symposium on microarchitecture, page 214, 1996.

8. F. Dahlgren, F. Dubois, P. Stenstrom. *Sequential Hardware prefetching in shared memory multiprocessors*. IEEE Transactions on parallel and distributed systems, page 733, July 1995.

9. T–F. Chen. *Effective Hardware–based data prefetching for high performance processors. IEEE transactions on computers*, page 609, May 1995.

10. S. Kim, A. Veidenbaum. *Stride–directed prefetching for secondary caches. Proceedings of the International conference on parallel processing*, page 314, Aug 1997.

11. D. Joseph, K. Grunwald. *Prefetching using Markov Predictors*. Proceedings of the International symposium of computer Architecture, page 252, 1997.

12. E. D. Moreno, S. T. Kofuji, C. A. P. S. Martins. *Tolerating medium Latencies on data caches with hardware based prefetching*. IEEE Canadian conference on Electrical and Computer Engineering, page 720, may 1997.

13. G. Tyson, M. Farrens, J. Matthews, and A. Pleszkun. *A modified approach to Data Cache Management*. Proceedings of the 28th annual international symposium on microarchitecture, page 93, 1995.

14. C−H. Chi, C−M Cheung. *Hardware driven Prefetching for Pointer Data References. Proceedings of the international conference on supercomputing*, page 377, 1998.

15. K. Farkas, N. Jouppi, P. Chow. *How effective are non−blocking loads, stream buffers, and speculative execution in multiple issue processors?* IEEE Symposium on High Performance Computer Architecture, page 78, January 1995.

16. Gi−Ho Park, Oh−Young Kwon, Tack−Don Han, Shin−Dug Kim, Sung−Bong Yang. *An improved lookahead instruction prefetching*. IEEE Proceedings High Performance Computing on the Information Superhighway HPC Asia, page 712, May 1997

17. Wei−Chung Hsu and J.E. Smith. *A Performance Study of instruction cache prefetching methods. IEEE transactions on Computers*, page 497, May 1998.

18. J. Pierce and T. Mudge. *Wrong Path Instruction Prefetching. Proceedings of the international symposium on microarchitecture*, page 165, 1996.

19. I−C. K. Chen, L. C−C. Lee, T. Mudge. *Instruction Prefetching Using Branch Prediction Information*. International conference on Computer Design, VLSI in Computers and Processors, page 593 October 1997

20. S−A. Chi, R−M. Shiu, J−C Chiu, S−E Chang, and C−P. Chung. *Instruction Cache Prefetching with Extended BTB. Proceedings of the international conference on parallel and distributed systems*, page 360, December 1997.

21. H.C. Young, E.J. Shekita. *An intelligent I−cache prefetch mechanism. Proceeding of IEEE Inernational Conference on Computer Architecture*, page 44, October 1993.

22. A. Veidenbaum, *Instruction cache prefetching using multi−level branch prediction. International symposium on High Performance Computing*, page 51, November 1997.

23. N. Jouppi. *Improving direct mapped cache performance by the addition of a small fully−associative cache and Stream buffers*. 25 years of the international symposia on computer Architecture, page 388, 1998.

24. N. Jouppi. *Retrospective: Improving direct mapped cache performance by the addition of a small fully−associative cache and stream buffers*. 25 years of the international symposia on computer Architecture, page 71, 1998.

25. D. Patterson and J. L. Hennessey. *Computer Architecture: A quantitative approach*. Morgan Kaufman Publishers, San Francisco, 1996.

26. S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. *Predictability of Load/Store Instruction Latencies*. Proceedings of the 26th annual International Symposium on Microarchitecture, Austin, Texas, 1993, page 139.