
Contents

I	This is a Part	7
1	Data Cache Prefetching	11
	<i>Yan Solihin</i> ¹ and <i>Donald Yeung</i> ² North Carolina State University ¹ , University of Maryland at College Park ²	
1.1	Introduction	11
1.2	Software Prefetching	12
1.2.1	Architectural Support	12
1.2.2	Array Prefetching	13
1.2.3	Pointer Prefetching	17
1.2.4	Relationship with Data Locality Optimizations	21
1.3	Hardware Prefetching	22
1.3.1	Stride and Sequential Prefetching	23
1.3.2	Correlation Prefetching	25
1.3.3	Content-Based Prefetching	28
	References	31

List of Tables

List of Figures

1.1	Example illustrating Mowry's algorithm.	15
1.2	Example illustrating greedy pointer prefetching.	18
1.3	Example illustrating jump-pointer and prefetch-array pointer prefetching.	20
1.4	Illustration for stream buffer operation.	24
1.5	Correlation prefetching algorithms	27
1.6	Heuristics for pointer identification	29

Part I

This is a Part

Symbol Description

α	To solve the generator maintenance scheduling, in the past, several mathematical techniques have been applied.		algorithms have also been tested.
σ^2	These include integer programming, integer linear programming, dynamic programming, branch and bound etc.	$\theta\sqrt{abc}$	This paper presents a survey of the literature
Σ	Several heuristic search algorithms have also been developed. In recent years expert systems,	ζ	over the past fifteen years in the generator
abc	fuzzy approaches, simulated annealing and genetic	∂	maintenance scheduling. The objective is to
		sdf	present a clear picture of the available recent literature
		ewq	of the problem, the constraints and the other aspects of
		bvcn	the generator maintenance schedule.

Chapter 1

Data Cache Prefetching

Yan Solihin¹ and Donald Yeung²

North Carolina State University¹, University of Maryland at College Park²

1.1	Introduction	11
1.2	Software Prefetching	12
1.3	Hardware Prefetching	22

1.1 Introduction

Due to the growing gap in processor and memory speeds, a cache miss is becoming more expensive. Data prefetching is a technique to hide cache miss latency by bringing data closer to the processor ahead of the processor's request for it. One way to initiate prefetching is by inserting prefetch instructions into the program. This insertion is typically performed statically by the compiler. We will discuss it in the *Software Prefetching* section (Section 1.2). Prefetching can also be initiated dynamically by observing the program's behavior. Since this dynamic prefetch generation is typically performed by hardware, we will discuss it in the *Hardware Prefetching* section (Section 1.3). Note that this boundary is not firm, for example prefetch instructions may be inserted dynamically by run-time optimizer, whereas some hardware prefetching techniques may be implemented in software.

There are three metrics that characterize the effectiveness of a prefetching technique: coverage, accuracy, and timeliness. *Coverage* is defined as the fraction of original cache misses that are prefetched, and therefore become cache hits or partial cache misses. *Accuracy* is defined as the fraction of prefetches that are useful, i.e. they result in cache hits. And finally, *timeliness* defines how timely the prefetches are: whether the full cache miss latency is hidden, or only a part of the miss latency is hidden. Obviously, an ideal prefetching technique should have high coverage that it eliminates most of the cache misses, high accuracy that it does not consume much extra memory bandwidth by sending useless prefetches, and high timeliness, that most of the prefetches hide the full cache miss latency. Achieving this ideal is a challenge. By aggressively issuing many prefetches, a prefetching technique may achieve a high coverage but low accuracy. By conservatively issuing only prefetches that are highly predictable, a prefetching technique may achieve a high accuracy

but a low coverage. Finally, timeliness is also important. If a prefetch is initiated too early, it may pollute the cache, and may be replaced from the cache or prefetch buffer before it is used by the processor. If it is initiated too late, it may not hide the full cache miss latency.

Note that in addition to the performance metrics, there are also other practical considerations that are important, such as the cost and complexity of a hardware implementation and whether it needs code recompilation.

Most of the prefetching techniques discussed in this chapter were designed for uniprocessor systems, although they can also be applied to multiprocessor systems. Multiprocessor-specific prefetching techniques will not be discussed in this chapter. Finally, due to a very high number of publications in data cache prefetching, it is impossible to list and discuss all of them in this chapter. The goal of this chapter is to give a brief overview of existing prefetching techniques. Thus, it will only focus on a subset of representative techniques that hopefully motivate readers into reading other prefetching studies as well.

1.2 Software Prefetching

Software prefetching relies on the programmer or compiler to insert explicit prefetch instructions into the application code for memory references that are likely to miss in the cache. At run time, the inserted prefetch instructions bring the data into the processor's cache in advance of its use, thus overlapping the cost of the memory access with useful work in the processor. Historically, software prefetching has been quite effective in reducing memory stalls for scientific programs that reference array-based data structures [3, 7, 24, 31, 32]. More recently, techniques have also been developed to apply software prefetching for non-numeric programs that reference pointer-based data structures as well [22, 28, 39].

This section discusses both types of software prefetching techniques. We begin by reviewing the architectural support necessary for software prefetching. Then, we describe the algorithms for instrumenting software prefetching, first for array-based data structures and then for pointer-based data structures. Finally, we describe the interactions between software prefetching and other software-based memory performance optimizations.

1.2.1 Architectural Support

Although software prefetching is a software-centric technique, some hardware support is necessary. First, the architecture's instruction set must provide a *prefetch instruction*. Prefetch instructions are non-blocking memory loads (i.e., they do not wait for the reply from the memory system). They

cause a cache fill of the requested memory location on a cache miss, but have no other side effects. Since they are effectively NOPs from the processor's standpoint, prefetch instructions can be ignored by the memory system (for example, at times when memory resources are scarce) without affecting program correctness. Hence, prefetch instructions enable software to provide "hints" to the memory system regarding the memory locations that should be loaded into cache.

In addition to prefetch instructions, software prefetching also requires lockup-free caches [26, 40]. Since software prefetching tries to hide memory latency underneath useful computation, the cache must continue servicing normal memory requests from the CPU following cache misses triggered by prefetch instructions. For systems that prefetch aggressively, it is also necessary for the cache and memory subsystem to support multiple outstanding memory requests. This allows independent prefetch requests to overlap with each other.

Unless the compiler or programmer can guarantee the safety of all prefetches, another requirement from the architecture is support for ignoring memory faults caused by prefetch instructions. This support is particularly useful when instrumenting prefetch instructions speculatively, for example to prefetch data accessed within conditional statements [3].

Finally, a few researchers have investigated software prefetching assuming support for *prefetch buffers* [7, 24]. Instead of prefetching directly into the L1 cache, these approaches place prefetched data in a special data buffer. On a memory request, the processor checks both the L1 cache and the prefetch buffer, and moves prefetched data into the L1 cache only on a prefetch buffer hit. Hence, prefetched blocks that are never referenced by the processor do not evict potentially useful blocks in the L1 cache. In addition, usefully prefetched blocks are filled into the L1 cache as late as possible—at the time of the processor's reference rather than the prefetch block's arrival—thus delaying the eviction of potentially useful blocks from the L1 cache. On the other hand, prefetch buffers consume memory that could have otherwise been used to build a larger L1 cache.

1.2.2 Array Prefetching

Having discussed the architectural support for software prefetching, we now examine the algorithms for instrumenting prefetches into application code. We begin by focusing on techniques for array references performed within loops that give rise to *regular memory access patterns*. These memory references employ array subscripts that are affine—i.e., linear combinations of loop index variables with constant coefficients and additive constants. (We also discuss prefetching for indirect array references that give rise to *irregular memory access patterns*, though to a lesser extent).

Affine array references are quite common in a variety of applications, including dense-matrix linear algebra and finite-difference PDE solvers as well as image processing and scans/joins in relational databases. These programs

can usually exploit long cache lines to reduce memory access costs, but may suffer poor performance due to cache conflict and capacity misses arising from the large amounts of data accessed. An important feature of these codes is that memory access patterns can be identified exactly at compile time, assuming array dimension sizes are known. Consequently, programs performing affine array references are good candidates for software prefetching.

1.2.2.1 Mowry’s Algorithm

The best-known approach for instrumenting software prefetching of affine array references is the compiler algorithm proposed by Mowry *et al* [33, 31]. To illustrate the algorithm, we use the 2-D Jacobi kernel in Figure 1.1a as an example, instrumenting it with software prefetching using Mowry’s algorithm in Figure 1.1b. The algorithm involves three major steps: locality analysis, cache miss isolation, and prefetch scheduling.

Locality analysis determines the array references that will miss in the cache, and thus require prefetching. The goal of this step is to avoid *unnecessary prefetches*, or prefetches that incur runtime overhead without improving performance because they hit in the cache. The analysis proceeds in two parts. First, reuse between dynamic instances of individual static array references is identified. In particular, locality analysis looks for three types of reuse: spatial, temporal, and group. Spatial reuse occurs whenever a static array reference accesses locations close together in memory. For example, every array reference in Figure 1.1a exhibits spatial reuse. Since the i loop performs a unit-stride traversal of each inner array dimension, contemporaneous iterations access the same cache block. In contrast, temporal reuse occurs whenever a static array reference accesses the same memory location. (None of the array references in Figure 1.1a exhibit temporal reuse since all dynamic instances access distinct array elements). Lastly, group reuse occurs whenever two or more different static array references access the same memory location. In Figure 1.1a, all four B array references exhibit group reuse since many of their dynamic instances refer to the same array elements.

After identifying reuse, locality analysis determines which reuses result in cache hits. The algorithm computes the number of loop iterations in between reuses, and the amount of data referenced within these intervening iterations. If the size of this referenced data is smaller than the cache size, then the algorithm assumes the reuse is captured in the cache and the dynamic instances do not need to be prefetched. All other dynamic instances, however, are assumed to miss in the cache, and require prefetching.

The next step in Mowry’s algorithm, after locality analysis, is *cache miss isolation*. For static array references that experience a mix of cache hits and misses (as determined by locality analysis), code transformations are performed to isolate the misses from the hits in separate static array references. This enables prefetching of the cache-missing instances while avoiding unnecessary prefetches fully statically, i.e. without predicating prefetch instructions

```

a) for (j=2; j <= N-1; j++)
    for (i=2; i <= N-1; i++)
        A[j][i]=0.25*(B[j][i-1]+B[j][i+1]+B[j-1][i]+B[j+1][i]);

b) for (j=2; j <= N-1; j++) {
    for (i=2; i <= PD; i+=4) {          // Prologue
        prefetch(&B[j][i]);
        prefetch(&B[j-1][i]);
        prefetch(&B[j+1][i]);
        prefetch(&A[j][i]);
    }
    for (i=2; i < N-PD-1; i+=4) {      // Steady State
        prefetch(&B[j][i+PD]);
        prefetch(&B[j-1][i+PD]);
        prefetch(&B[j+1][i+PD]);
        prefetch(&A[j][i+PD]);

        A[j][i]=0.25*(B[j][i-1]+B[j][i+1]+B[j-1][i]+B[j+1][i]);
        A[j][i+1]=0.25*(B[j][i]+B[j][i+2]+B[j-1][i+1]+B[j+1][i+1]);
        A[j][i+2]=0.25*(B[j][i+1]+B[j][i+3]+B[j-1][i+2]+B[j+1][i+2]);
        A[j][i+3]=0.25*(B[j][i+2]+B[j][i+4]+B[j-1][i+3]+B[j+1][i+3]);
    }
    for (i=N-PD; i <= N-1; i++)        // Epilogue
        A[j][i]=0.25*(B[j][i-1]+B[j][i+1]+B[j-1][i]+B[j+1][i]);
}

```

FIGURE 1.1: a). 2-D Jacobi kernel code b). instrumented with software prefetching using Mowry’s algorithm [32].

with IF statements that incur runtime overhead.

The appropriate cache miss isolation transformation depends on the type of reuse. For spatial reuse, *loop unrolling* is performed. Figure 1.1b illustrates loop unrolling for the 2-D Jacobi kernel (see the second nested loop). As described earlier, the i loop of the Jacobi kernel performs a unit-stride traversal of the A and B inner array dimensions. Assuming 8-byte array elements and a 32-byte cache block, cache misses for each static array reference occur every $\frac{32}{8} = 4$ iterations. By unrolling the loop 4 times, the leading array references incur cache misses every iteration while the remaining unrolled references never miss, thus isolating the cache misses. For temporal reuse, cache misses are isolated via *loop peeling*. In this case, some subset of iterations (usually the first) incur all the cache misses, and the remaining iterations hit in the cache. The cache-missing iteration(s) are peeled off and placed in a separate loop. Lastly, for group reuse, no explicit transformation is necessary since cache misses already occur in separate static array references. Analysis is performed simply to identify which static array reference is the leading reference in each group, and hence will incur all the cache misses on behalf of the other static references.

Once the cache misses have been isolated, prefetches are inserted for the static array references that incur them. Figure 1.1b illustrates the prefetch instrumentation for the 2-D Jacobi kernel. Assuming the group reuse between

$B[j][i + 1]$ and $B[j][i - 1]$ is captured by the cache but the group reuse across outer loop iterations (i.e., the j subscripts) is not, then there are four static array references that incur all the cache misses— $A[j][i]$, $B[j][i + 1]$, $B[j - 1][i]$, and $B[j + 1][i]$. Prefetches are instrumented for these four array references, as illustrated in Figure 1.1b (again, see the second nested loop). Notice, prefetches are not instrumented for the remaining static array references, thus avoiding unnecessary prefetches and reducing prefetch overhead.

The last step in Mowry’s algorithm is *prefetch scheduling*. Given the high memory latency of most modern memory systems, a single loop iteration normally contains insufficient work under which to hide the cost of memory accesses. To ensure that data arrive in time, the instrumented prefetches must initiate multiple iterations in advance. The minimum number of loop iterations needed to fully overlap a memory access is known as the *prefetch distance*. Assuming the memory latency is l cycles and the work per loop iteration is w cycles, the prefetch distance, PD , is simply $\lceil \frac{l}{w} \rceil$. Figure 1.1b illustrates the indices of prefetched array elements contain a PD term, providing the early prefetch initiation required.

By initiating prefetches PD iterations in advance, the first PD iterations are not prefetched, while the last PD iterations prefetch past the end of each array. These inefficiencies can be addressed by performing *software pipelining* to handle the first and last PD iterations separately. Software pipelining creates a *prologue loop* to execute PD prefetches for the first PD array elements, and an *epilogue loop* to execute the last PD iterations without prefetching. Along with the original loop, called the *steady state loop*, the transformed code initiates, overlaps, and completes all prefetches relative to the computation in a pipelined fashion. Figure 1.1b illustrates the prologue, steady state, and epilogue loops created by software pipelining for the 2-D Jacobi kernel.

1.2.2.2 Support for Indexed Arrays

In addition to affine arrays, another common type of array is *indexed arrays*. Indexed arrays take the form $A[B[i]]$, in which a data array and index array are composed to form an *indirect array reference*. While the inner array reference is affine and regular, the outer array reference is *irregular* due to memory indirection. Indexed arrays arise in scientific applications that attempt complex simulations. In computational fluid dynamics, meshes for modeling large problems are sparse to reduce memory and computation requirements. In N-body solvers for astrophysics and molecular dynamics, data structures are irregular because they model the positions of particles and their interactions. Indexed arrays are frequently used to express these complex data relationships since they are more efficient than pointers. Unfortunately, the cache performance of indexed arrays can be poor since both spatial and temporal locality is low due to the irregularity of the access pattern.

Software prefetching of indexed arrays was first studied by Callahan *et al* [3]. Later, Mowry extended his algorithm for affine arrays discussed in Sec-

tion 1.2.2.1 to handle indexed arrays [34, 31]. Mowry’s extended algorithm follows the same steps discussed earlier, with some modifications. The modifications stem from two differences between indexed and affine array references. First, static analysis cannot determine locality information for consecutively accessed indexed array elements since their addresses depend on the index array values known only at runtime. Hence, the algorithm conservatively assumes no reuse occurs between indexed array references, requiring prefetching for all dynamic instances. Moreover, since the algorithm assumes all dynamic instances miss in the cache, there is also no need for cache miss isolation.

Second, before a data array reference can perform, the corresponding index array reference must complete since the index value is used to index into the data array. Hence, pairs of data and index array references are serialized. This affects prefetch scheduling. As in affine array prefetching, the computation of the prefetch distance uses the formula, $PD = \lceil \frac{1}{w} \rceil$. However, the adjustment of array indices for indexed arrays must take into consideration the serialization of data and index array references. Since data array elements cannot be prefetched until the index array values they depend on are available, prefetches for index array elements should be initiated *twice* as early as data array elements. This ensures that an index array value is in cache when its corresponding data array prefetch is issued.

1.2.3 Pointer Prefetching

In Section 1.2.2, we discussed software prefetching techniques for array-based codes. This section focuses on techniques for *pointer-chasing codes*. Pointer-chasing codes use linked data structures (LDS), such as linked lists, n-ary trees, and other graph structures, that are dynamically allocated at run time. They arise from solving complex problems where the amount and organization of data cannot be determined at compile time, requiring the use of pointers to dynamically manage both storage and linkage. They may also arise from high-level programming language constructs, such as those found in object-oriented programs.

Due to the dynamic nature of LDS creation and modification, pointer-chasing codes commonly exhibit poor spatial and temporal locality, and experience poor cache behavior. Another significant performance bottleneck in these workloads is the *pointer-chasing problem*. Because individual nodes in an LDS are connected through pointers, access to additional parts of the data structure cannot take place until the pointer to the current portion of the data structure is resolved. Hence, LDS traversal requires the sequential reference and dereference of the pointers stored inside visited nodes, thus serializing all memory operations performed along the traversal.

The pointer-chasing problem not only limits application performance, but it also potentially limits the effectiveness of prefetching. As discussed earlier, an important goal of software prefetching is to initiate prefetches sufficiently early to tolerate their latency. Doing so requires knowing the prefetch memory

```

a)  struct node {data, next}      b)  struct node {data, left, right}
     *ptr, *list_head;           *ptr;

ptr = list_head;                 void recurse(ptr) {
while (ptr) {                     prefetch(ptr->left);
  prefetch(ptr->next);             prefetch(ptr->right);
  ...                              ...
  ptr = ptr->next;                 recurse(ptr->left);
}                                   recurse(ptr->right);

```

FIGURE 1.2: Example pointer prefetching for a). linked list and b). tree traversals using greedy prefetching [28].

address in advance as well. This is straight forward for array data structures since the address of future array elements can be computed given the desired array indices. However, determining the address of a future link node in an LDS requires traversing the intermediate link nodes due to the pointer-chasing problem, thus preventing the early initiation of prefetches.

LDS prefetching techniques try to address the pointer chasing problem and its impact on early prefetch initiation. We present the techniques in two groups: those that use the natural pointers in the LDS only, and those that create special pointers called *jump pointers* for the sole purpose of prefetching.

1.2.3.1 Natural Pointer Techniques

The simplest software prefetching technique for LDS traversal is *greedy prefetching*, proposed by Luk and Mowry [28]. Greedy prefetching inserts software prefetch instructions immediately prior to visiting a node for all possible successor nodes that might be encountered by the traversal. To demonstrate the technique, Figure 1.2 shows the prefetch instrumentation for two types of LDS traversals. Figure 1.2a illustrates greedy prefetching for a loop-based linked list traversal. In this case, a single prefetch is inserted at the top of the loop for the next link node in the list. Figure 1.2b illustrates greedy prefetching for a recursive tree traversal. In this case, prefetches are inserted at the top of the recursive function for each child node in the sub-tree.

Greedy prefetching is attractive due to its simplicity. However, its ability to properly time prefetch initiation is limited. For the linked list traversal in Figure 1.2a, each prefetch overlaps with a single loop iteration only. Greater overlap is not possible since the technique cannot prefetch nodes beyond the immediate successor due to the pointer-chasing problem described earlier. If the amount of work in a single loop iteration is small compared to the prefetch latency, than greedy prefetching will not effectively tolerate the memory stalls. The situation is somewhat better for the tree traversal in Figure 1.2b. Although the prefetch of `ptr->left` suffers a similar problem as the linked list traversal (its latency overlaps with a single recursive call only), the prefetch of `ptr->right` overlaps with more work—the traversal of the entire left sub-tree. But the timing of prefetch initiation may not be ideal. The latency for

prefetching `ptr->right` may still not be fully tolerated if the left sub-tree traversal contains insufficient work. Alternatively, the prefetched node may arrive too early and suffer eviction if the left sub-tree traversal contains too much work.

Another approach to prefetching LDS traversals is *data linearization prefetching*. Like greedy prefetching, this technique was proposed by Luk and Mowry [28] (a somewhat similar technique was also proposed by Stoutchinin *et al* [44]). Data linearization prefetching makes the observation that if contemporaneously traversed link nodes in an LDS are laid out linearly in memory, then prefetching a future node no longer requires sequentially traversing the intermediate nodes to determine its address. Instead, a future node’s memory address can be computed simply by offsetting from the current node pointer, much like indexing into an array. Hence, data linearization prefetching avoids the pointer-chasing problem altogether, and can initiate prefetches as far in advance as necessary.

The key issue for data linearization prefetching is achieving the desired linear layout of LDS nodes. Linearization can occur either at LDS creation, or after an LDS has been created. From a cost standpoint, the former is more desirable since the latter requires reorganizing an existing LDS via data copying at runtime. Moreover, linearizing at LDS creation is feasible especially if the order of LDS traversal is known a priori. In this case, the allocation and linkage of individual LDS nodes should simply follow the order of node traversal. As long as the memory allocator places contemporaneously allocated nodes regularly in memory, the desired linearization can be achieved. Notice, however, periodic “re-linearization” (via copying) may be necessary if link nodes are inserted and deleted frequently. Hence, data linearization prefetching is most effective for applications in which the LDS connectivity does not change significantly during program execution.

1.2.3.2 Jump Pointer Techniques

Both greedy and data linearization prefetching do not modify the logical structure of the LDS to perform prefetching. In contrast, another group of pointer prefetching techniques have been studied that insert special pointers into the LDS, called *jump pointers*, for the sole purpose of prefetching. Jump pointers connect non-consecutive link nodes, allowing prefetch instructions to name link nodes further down the pointer chain without traversing the intermediate link nodes and without performing linearization beforehand. Effectively, the jump pointers increase the dimensionality and reduce the diameter of an LDS, an idea borrowed from Skip Lists [37].

Several jump pointer techniques have been studied in the literature. The most basic approach is *jump pointer prefetching* as originally proposed by Luk and Mowry [28]. Roth and Sohi [39] also investigated jump pointer prefetching, introducing variations on the basic technique in [28] that use a combination of software and hardware support to pursue the jump pointers.

```

a)  struct node {data, next, jump}
      *ptr, *list_head, *prefetch_array[PD], *history[PD];
      int i, head, tail;

      for (i = 0; i < PD; i++) // Prologue Loop
        prefetch(prefetch_array[i]);

      ptr = list_head;
      while (ptr->next) { // Steady State Loop
        prefetch(ptr->jump);
        ...
        ptr = ptr->next;
      }

b)  for (i = 0; i < PD; i++) history[i] = NULL;
      tail = 0;
      head = PD-1;

      ptr = list_head;
      while (ptr) { // Prefetch Pointer Generation Loop
        history[head] = ptr;
        if (!history[tail])
          prefetch_array[tail] = ptr;
        else
          history[tail]->jump = ptr;
        head = (head+1)%PD;
        tail = (tail+1)%PD;
        ptr = ptr->next;
      }

```

FIGURE 1.3: Example pointer prefetching for a linked list traversal using jump pointers and prefetch arrays [22].

Figure 1.3a illustrates the basic technique, applying jump pointer prefetching to the same linked list traversal shown in Figure 1.2a. Each linked list node is augmented with a jump pointer field, called `jump` in Figure 1.3a. During a separate initialization pass (discussed below), the jump pointers are set to point to a link node further down the linked list that will be referenced by a future loop iteration. The number of consecutive link nodes skipped by the jump pointers is the prefetch distance, PD , which is computed using the same approach described in Section 1.2.2 for array prefetching. Once the jump pointers have been installed, prefetch instructions can prefetch through the jump pointers, as illustrated in Figure 1.3a by the loop labeled “Steady State.”

Unfortunately, jump pointer prefetching cannot prefetch the first PD link nodes in a linked list because there are no jump pointers that point to these early nodes. In many pointer-chasing applications, this limitation significantly reduces the effectiveness of jump pointer prefetching because pointer chains are kept short by design. To enable prefetching of early nodes, jump pointer prefetching can be extended with *prefetch arrays* [22]. In this technique, an array of prefetch pointers is added to every linked list to point to the first PD

link nodes. Hence, prefetches can be issued through the memory addresses in the prefetch arrays before traversing each linked list to cover the early nodes, much like the prologue loops for array prefetching prefetch the first PD array elements. Figure 1.3b illustrates the addition of a prologue loop that performs prefetching through a prefetch array.

As described earlier, the prefetch pointers must be installed before prefetching can commence. Figure 1.3b shows an example of prefetch pointer installation code which uses a *history pointer array* [28] to set the prefetch pointers. The history pointer array, called `history` in Figure 1.3b, is a circular queue that records the last PD link nodes traversed by the initialization code. Whenever a new link node is traversed, it is added to the head of the circular queue and the head is incremented. At the same time, the tail of the circular queue is tested. If the tail is NULL, then the current node is one of the first PD link nodes in the list since PD link nodes must be encountered before the circular queue fills. In this case, we set one of the `prefetch_array` pointers to point to the node. Otherwise, the tail's jump pointer is set to point to the current link node. Since the circular queue has depth PD , all jump pointers are initialized to point PD link nodes *ahead*, thus providing the proper prefetch distance. Normally, the compiler or programmer ensures the prefetch pointer installation code gets executed prior to prefetching, for example on the first traversal of an LDS. Furthermore, if the application modifies the LDS after the prefetch pointers have been installed, it may be necessary to update the prefetch pointers either by re-executing the installation code or by using other fix-up code.

1.2.4 Relationship with Data Locality Optimizations

There have been several software-centric solutions to the memory bottleneck problem; software prefetching is just one of these software-based solutions. Another important group of software techniques is *data locality optimizations*. Data locality optimizations enhance the locality of data memory references, either by changing the layout of data in memory or by changing the order of accesses to the data, thus improving the reuse of data in the cache.

The type of optimization depends on the memory reference type. For affine array references, *tiling* is used [45]. Tiling combines strip-mining with loop permutation to form small tiles of loop iterations which are executed together. Alternatively, for indexed array references, *runtime data access reordering* is used [12, 29, 30]. This technique reorders loop iterations at runtime using an inspector-executor approach [11] to bring accesses to the same data closer together in time. Finally for pointer references, *cache-conscious heap allocation* [2, 8] is used. Cache-conscious heap allocation places logically linked heap elements physically close together in memory at memory allocation time to improve spatial locality. (This technique is related to data linearization described in Section 1.2.3.1).

Although both software prefetching and data locality optimizations improve

memory performance, they do so in very different ways. Prefetching initiates memory operations early to hide their latency underneath useful computation, whereas data locality optimizations improve reuse in the data cache. The former is a *latency tolerance* technique while the latter is a *latency reduction* technique.

Because prefetching only tolerates memory latency, it requires adequate memory bandwidth to be effective. As software prefetches bring required data to the processor increasingly early, the computation executes faster, causing a higher rate of data requests. If memory bandwidth saturates, prefetching will not provide further performance gains, and the application will become limited by the speed of the memory system. In contrast, data locality optimizations reduce the average memory latency by eliminating a portion of the application's cache misses. Hence, it not only reduces memory stalls, it also reduces traffic to lower levels of the memory hierarchy as well as the memory contention this traffic causes.

Compared to data locality optimizations, software prefetching can potentially remove more memory stalls since it can target *all* cache misses. Provided ample memory bandwidth exists, software prefetching can potentially remove the performance degradation of the memory system entirely. Data locality optimizations cannot remove all stalls, for example those arising from cold misses. Another advantage of software prefetching is software transformations are purely speculative—they do not change the semantics of the program code. In contrast, data locality optimizations can affect program correctness; hence, compilers must prove the correctness of transformations before they can be applied. In some cases, this can reduce the applicability of data locality optimizations compared to software prefetching.

1.3 Hardware Prefetching

The previous section has given an overview of software prefetching techniques. This section will give an overview of hardware prefetching techniques.

Based on which party initiates the prefetching, hardware prefetching techniques can be categorized as *processor-side* or *memory-side*. In the former approach, the processor or an engine in its cache hierarchy issues the prefetch requests [5, 6, 20, 21, 23, 25, 36, 38, 41, 47, 10]. In the latter approach, the engine that prefetches data for the processor is in the main memory system [42, 43, 1, 4, 17, 35, 46].

The advantages of processor-side prefetching is the abundant information that can be used by the processor, which includes the program counter, virtual addresses of memory references, etc, that are not always available to the memory system. One advantage of memory-side prefetching is that it eliminates

the overheads and state bookkeeping that prefetch requests introduce in the paths between the main processor and its caches. In addition, the prefetcher can exploit its proximity to the memory to its advantage, for example by storing its state in memory and by exploiting low latency and high bandwidth access to the main memory. In microprocessors, processor-side prefetching is typically implemented at the level of L1 or L2 cache, using simple engines that detect and prefetch spatial locality of a program, such as in the Intel Pentium 4 [14], and IBM Power4 [19]. At the system level, memory-side prefetching is typically implemented in the memory controller that buffers the prefetched data from the memory and targets prefetching strided accesses, such as in NVIDIA's DASP engine [35].

Based on the type of data access patterns that can be handled, *sequential prefetching* detects and prefetches for accesses to contiguous locations. *stride prefetching* detects and prefetches accesses that are s -cache block apart between consecutive accesses, where s is the amount of the stride. Therefore, an s -strided accesses would produce address trace of $A, A + s, A + 2s, \dots$. Note that sequential prefetching is a stride prefetching where $s \leq 1$. Finally, some accesses that are not strided may appear random to the prefetcher, and is therefore not easy to detect and prefetch. These accesses may result from a linked data structure traversal, where the allocation sequence of the instances is such that they are not located in contiguous (or strided) memory locations. They may also result from indirect array references, such as in $\mathbf{a}[\mathbf{b}[i]]$. There are ways to deal with this. One type of techniques relies on the fact that although the accesses appear random, their sequence is often repeated during program execution. Therefore, by recording the sequence, address correlation can be learned and used for prefetching. This type of technique is called *correlation prefetching*. Finally, for accesses that result from dereferencing pointers in linked data structures, prefetching can be initiated if a pointer is identified and the data pointed is prefetched, before the actual dereferencing. This technique is called *content-directed prefetching*.

Finally, there are various places where prefetching can be initiated and where the destination where the prefetched data can be placed. Prefetching can be initiated in the L1 cache level, L2 cache level, the memory controller, or even in the memory chips. The prefetched data is usually stored at the level where the prefetch is initiated, either in the prefetch-initiating cache, or in a separate prefetch buffer at the same level as the prefetch-initiating cache. In some cases, the initiating level is not the same as the destination. For example, a memory-side prefetching may be initiated near the memory but the prefetched data is pushed into the L2 cache.

1.3.1 Stride and Sequential Prefetching

Early stride and sequential prefetching studies include the Reference Prediction Table by Chen and Baer [6], and stream buffers by Jouppi [21]. This section will describe stream buffers scheme, which is attractive due to its

simplicity. A stream buffer allows prefetching sequential accesses. A stream buffer is a FIFO buffer, where each entry contains a line that has the same size as a cache line, an address (or tag) of the line, and an “available” bit. To prefetch for multiple streams in parallel, more than one stream buffers can be used, where each buffer prefetches from one stream.

On a cache access, the cache and the head entries of the stream buffers are checked for a match. If the requested line is found in the cache, no action on the stream buffers is performed. If the line is not found in the cache, but found at the head of a stream buffer, the line is moved into the cache. The head pointer of the stream buffer moves to the next entry, and the buffer prefetches the last entry’s successor into the freed entry, i.e., if the last entry’s line address is L , $L + 1$ is prefetched. If the line is not found in both the cache and all the stream buffers, a new stream buffer is allocated. The line’s successors are prefetched to fill the stream buffers. While a prefetch is in flight, the available bit is set to ‘0’, and is set to ‘1’ only when the prefetch has completed.

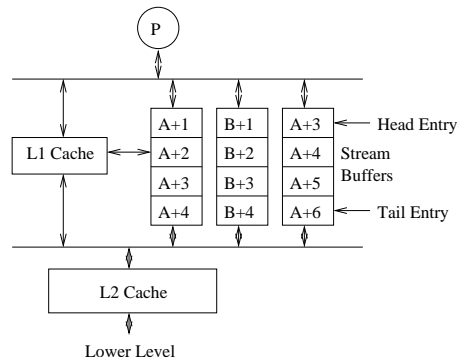


FIGURE 1.4: Illustration for stream buffer operation.

Figure 1.4 illustrates the stream buffer operation with three stream buffers. Suppose that the addresses from access streams are A , B , $A+2$, $B+1$, $A+4$, $B+2$, \dots and that the cache and stream buffers do not contain any of the blocks initially. The access to A results in a cache miss, which results in allocating a stream buffer for A 's successors, which are $A+1$, $A+2$, $A+3$, $A+4$. Similarly, the access to B results in a cache miss, and a stream buffer is allocated for B 's successors $B+1$, $B+2$, $B+3$, $B+4$. Next, for an access to $A+2$, the cache and the head entries of the stream buffers are checked for a match. Since there is no match, a new stream buffer is allocated for $A+2$'s successors $A+3$, $A+4$, $A+5$, $A+6$. However, an access to $B+1$ will find a match in the stream buffer, and $B+1$ will be moved into the L1 cache, and the freed

entry is used to prefetch the successor of the tail entry.

Note that now we have two stream buffers allocated for A 's successors and $A + 2$'s successors, and some entries overlap in two stream buffers ($A + 3$ and $A + 4$). This is caused by the inability to identify a stride of 2 for A 's stream. In fact, it cannot handle all non-sequential accesses efficiently, including non-unit strides ($s > 1$) or even a negative sequential accesses ($s = -1$). For non-unit stride accesses, each access could result in a stream buffer miss because the address is only compared to the head entries of the buffers. Not only this produces overlapping entries, a single stream can also occupy several stream buffers. To overcome that, Palacharla and Kessler [36] suggested two techniques to enhance the stream buffers: allocation filters and a non-unit stride detection mechanism. The filter waits until there are two consecutive misses for the same stream before allocating a stream buffer, making sure that a stream buffer is only allocated for strided accesses. A dynamic non-unit stride detection was also proposed by determining the minimum signed difference between the miss address and the past N miss addresses. Farkas, et al. [13] attempt to accurately detect the stride amount by using not only load addresses, but also the PCs of the load instructions. A stride address prediction table, indexed by the PC of a load is used to learn and record dynamic strides. Note that, with larger cache blocks, more strided access patterns will appear sequential. Sherwood, et al. [41] augments the stream buffers with a correlation-based predictor that detects non-strided accesses.

Overall, prefetching for sequential and strided accesses is achievable with relatively simple hardware. Several recent machines, such as Intel Pentium 4 and IBM Power4 architectures implement hardware prefetchers that are similar to stream buffers at the L2 caches [14, 19]. For example, the L2 cache in the Power4 can detect up to eight sequential streams. The L2 cache line is 128 bytes, and therefore a sequential stream detector can catch most of strided accesses. On a cache miss to a line in the L2 cache, the fifth successor line is prefetched. This is similar to a 4-entry stream buffers. The L3 cache block is four times larger than the L2 (512 bytes), and therefore it only prefetches the next successive line on an L3 miss. Since the prefetching is based on physical addresses, prefetching of a stream is terminated when a page boundary is encountered. However, for applications with long sequential accesses, continuity in the prefetching can be supported if large pages (16-MB) are used.

1.3.2 Correlation Prefetching

While prefetching strided or sequential accesses can be accomplished with relatively simple hardware, such as stream buffers, there is less clear solution to prefetching for non-strided (*irregular*) accesses. This irregularity often results from accesses in linked data structures (pointer dereferences) or sparse matrices (matrix or array dereferences). Fortunately, although these accesses appear random, they are often repeated. For example, in a linked list, if the

structure is traversed, the addresses produced will be repeated as long as the list is not modified too often. For a more irregular linked data structures such as a tree or a graph, it is still possible to have access repetition if the traversal of the structures is fairly unchanged. This behavior is exploited by correlation prefetching.

Correlation Prefetching uses past sequences of reference or miss addresses to predict and prefetch future misses [42, 43, 1, 5, 20, 27, 41, 9, 15, 16]. It identifies a correlation between pairs or groups of addresses, for example between a miss and a sequence of successor misses. A typical implementation of pair-based schemes uses a *Correlation Table* to record the addresses that are correlated. Later, when a miss is observed, all the addresses that are correlated with its address are prefetched. Correlation prefetching has general applicability in that it works for any access or miss patterns as long as the miss address sequences repeat. This includes strided accesses, even though it will be more cost-effective to use a specialized stride or sequential prefetcher instead.

Several studies have proposed hardware-based implementations [1, 5, 20, 27, 41], typically by placing a custom prefetch engine and a hardware correlation table between the processor and L1 caches or between the L1 and L2 caches, or in the main memory for prefetching memory pages. Figure 1.5-(a) shows how a typical (*Base*) correlation table, as used in [5, 20, 41], is organized. Each row stores the tag of an address that missed, and the addresses of a set of *immediate* successor misses. These are misses that have been seen to *immediately* follow the first one at different points in the application. The parameters of the table are the maximum number of immediate successors per miss (*NumSucc*), the maximum number of misses that the table can store predictions for (*NumRows*), and the associativity of the table (*Assoc*). According to [20], for best performance, the entries in a row should replace each other with a LRU policy.

Figure 1.5-(a) illustrates how the algorithm works. The figure shows two snapshots of the table at different points in the miss stream ((i) and (ii)). Within a row, successors are listed in Most Recently Used (MRU) order from left to right. At any time, the hardware keeps a pointer to the row of the last miss observed. When a miss occurs, the table learns by placing the miss address as one of the immediate successors of the last miss, and a new row is allocated for the new miss unless it already exists. When the table is used to prefetch ((iii)), it reacts to an observed miss by finding the corresponding row and prefetching all *NumSucc* successors, starting from the MRU one. In the figure, since the table has observed *a*, *b* and *a*, *d* in the past, when *a* is observed, both *b* and *d* are prefetched. Note that the figure shows two separate steps: the learning step, where the table records and learns the address patterns ((i) and (ii)) and prefetching step, where the table is used for prefetching. In practice, the prefetching performs better when both steps are combined [20, 42, 43].

There are enhancements to the *Base* correlation prefetching. Sherwood, et al. combines the prefetcher with traditional stream buffers, combining the

ability to prefetch regular and irregular accesses [41]. However, its location near the processor limits the size of the table, limiting its effectiveness for programs with large working sets. Lai, et al. uses the PC information as well as addresses to index the correlation table [27]. Although the correlation information can be made more accurate, it is more complicated because PCs need to be obtained. It also requires longer training.

A drawback of the *Base* approach is that to be effective, it needs a large correlation table. Proposed schemes typically need a 1-2 Mbyte on-chip SRAM table [20, 27], while some applications with large footprints even need a 7.6 Mbyte off-chip SRAM table [27]. In addition, prefetching only the *immediate* successors for each miss [5, 20, 41] limits the prefetcher’s performance. The prefetching needs to wait until a “trigger” miss occurs before it can prefetch its immediate successors. The trigger misses cannot be eliminated since they are needed by the prefetcher to operate. Therefore, it has a low *coverage*, defined as the fraction of the original of misses that are prefetched [20]. In addition, the immediate successor prefetch may not be timely enough to hide its full memory latency.

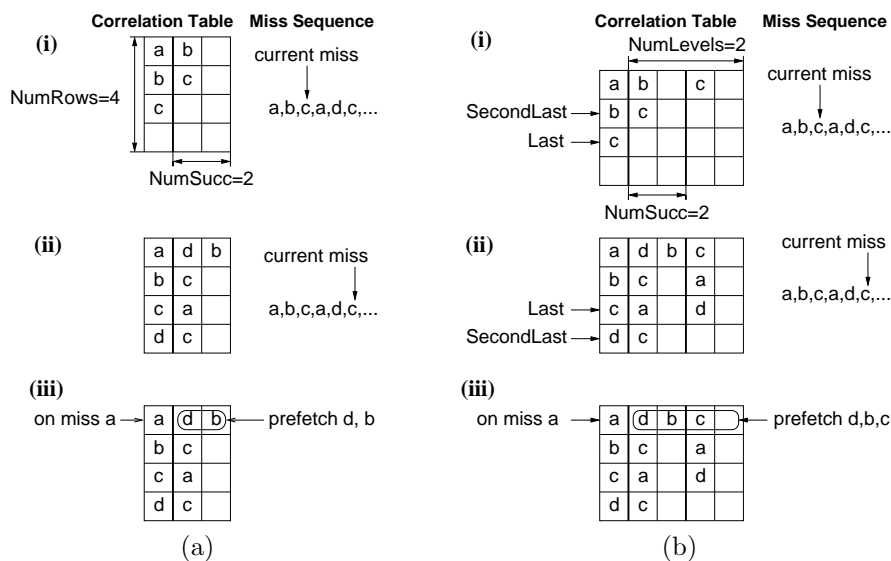


FIGURE 1.5: Correlation prefetching algorithms: *Base* (a) and *Replicated* (b). The figure is modified from [42].

One possible extension to the *Base* scheme is to reindex the table and follow the successor chains starting from the current miss address. This approach, which can potentially increase coverage and prefetching timeliness, has a drawback in that the accuracy of prefetching quickly degrades as the

chain is traversed [42, 43]. To deal with the drawbacks of *Base* correlation prefetching, Solihin, et al. proposed an alternative *Replicated* correlation table [42, 43] that records multiple levels of successors in each entry of the correlation table. This scheme is shown in Figure 1.5b. As shown in the figure, *Replicated* keeps *NumLevels* pointers to the table. These pointers point to the entries for the address of the last miss, second last, and so on, and are used for efficient table access. When a miss occurs, these pointers are used to access the entries of the last few misses, and insert the new address as the MRU successor of the correct level ((i) and (ii)). In the figure, the *NumSucc* entries at each level are MRU ordered. Finally, prefetching in *Replicated* is simple: when a miss is seen, all the entries from different successor levels in the corresponding row are prefetched ((iii)).

Replicated can prefetch multiple levels of successors more accurately compared to traversing a successor chain, because they contain the *true MRU* successors at each level. This is the result of grouping together all the successors from a given level, irrespective of the path taken. In [42, 43], three levels of successors can be predicted without losing much accuracy.

Correlation prefetching can be implemented in hardware [1, 5, 20, 27, 41, 15, 16] or in software [42, 43]. A software implementation is slower, and therefore is more suitable for an implementation near the memory. An advantage of a software implementation is that the correlation tables can be stored in the main memory, eliminating the need for an expensive specialized storage. In addition, the prefetching algorithm can be customized based on the application characteristics [42, 43].

1.3.3 Content-Based Prefetching

Another class of prefetching examines the content of data being fetched to identify whether it contains an address or pointer that is likely to be accessed or dereferenced in the future, and prefetch it ahead of time. Note that this technique only works for prefetching pointers in linked data structures, and do not tackle regular or indirect matrix/array indexing.

Roth, et al. [38] proposed a technique to identify a pointer load, based on whether a load (consumer) is an address produced by another load instruction (producer). If the producer and consumer are the same static load instructions, they are categorized as *recurrent loads*. Recurrent load results from pointer chasing, for example in the form of $p = p \rightarrow \text{next}$. Otherwise, they are categorized as *traversal loads*. Other loads that do not fetch pointers are categorized as *data loads*. A hardware that dynamically detects and prefetches the various pointer loads was proposed. Ibanez, et al. characterized the frequency of different types of loads, including pointer loads in the Spec benchmarks and proposed a prefetching scheme for them [18].

Cooksey, et al. [10] proposed content-directed prefetching, where pointers are identified and prefetched based on a set of heuristics applied to the content of a memory block. The technique borrows from conservative garbage

collection, in that when data is demand-fetched from memory, each address-sized word of the data is examined for a "likely" address. Candidate addresses need to be translated from the virtual to the physical address space and then issued as prefetch requests. As prefetch requests return data from memory, their contents are also examined to retrieve subsequent candidates. A set of heuristics are needed to identify "likely" addresses.

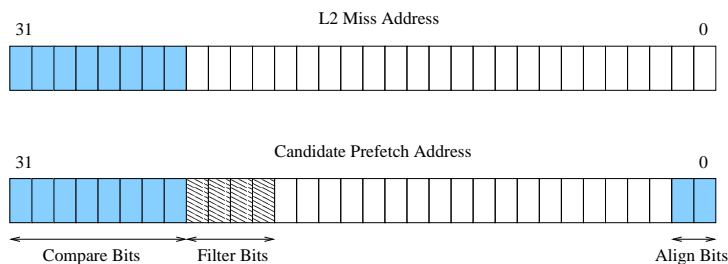


FIGURE 1.6: Heuristics for pointer identification. The figure is taken from [10], with minor modifications.

The heuristics are based on scanning each four-byte chunk on every loaded cache line. Each chunk is divided into several sections, as shown in Figure 1.6. The figure shows an L2 miss address that causes a block to be brought into the L2 cache, and a four-byte chunk from the block just brought in that is being considered as a candidate prefetch address. The first heuristic compares a few upper bits (*compare bits*) of the candidate prefetch address with ones from the miss address. If they match, the candidate prefetch address maybe an address that shares the same base address as the miss address. This heuristic relies on the fact that linked data structure traversal may dereference many pointers that are located nearby, i.e., they share the same base address. Other pointers that jump across large space, for example a pointer from the stack to heap region, cannot be identified by the heuristic as addresses. The heuristics works well except for the two regions where the compare bits are all 0's or all 1's. Small non-address integers with 0's in their compare bits should not be confused as addresses, and therefore need a different heuristic to distinguish them from addresses. In addition, large negative numbers may have all 1's in the compare bits, and therefore need a different heuristic to avoid them to be identified as addresses. One alternative would be to not consider prefetch addresses that have all 0's and all 1's in their compare bits. Unfortunately, these cases are too frequent to ignore because many operating systems allocate stack or heap data in those locations. To get around it, a second heuristic is employed. If the compare bits are all 0's, the next several bits (*filter bits*) are examined. If a non-zero bit is found within the filter bit range, the data value

is deemed to be a likely address. In addition, if the compare bits are all 1's, the next several bits (*filter bits*) are examined. If a zero bit is found within the filter bit range, the data value is deemed to be a likely address. Finally, a few lower bits (*align bits*) also help in identifying pointers. Due to memory alignment restriction, it is likely that addresses pointed by a pointer begin at the start of 4-byte chunks in memory. Therefore, the lower 2 bits of the pointer should be '00' in this case.

One inherent limitation of content-based prefetching is that a prefetch cannot be issued until the content of a previous cache miss is available for examination. Therefore, for dependent pointer loads, such as in linked data structure traversal, pointer chasing of n -linked nodes still result in n times the latency of a single pointer load. If each pointer load results in a cache miss, the pointer chasing critical path may still be significant. This limitation does not apply to sequential/stride and correlation prefetching which do not need to observe pointer load dependences.

References

- [1] T. Alexander and G. Kedem. Distributed Predictive Cache Design for High Performance Memory Systems. In *the Second International Symposium on High-Performance Computer Architecture*, pages 254–263, February 1996.
- [2] B. Calder, C. Krintz, S. John, and T. Austin. Cache-Conscious Data Placement. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 139–149, San Jose, CA, October 1998. ACM.
- [3] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.
- [4] J. B. Carter et al. Impulse: Building a Smarter Memory Controller. In *the 5th International Symposium on High-Performance Computer Architecture*, pages 70–79, January 1999.
- [5] M. J. Charney and A. P. Reeves. Generalized Correlation Based Hardware Prefetching. *Technical Report EE-CEG-95-1, Cornell University*, February 1995.
- [6] T. F. Chen and J. L. Baer. Reducing Memory Latency via Non-Blocking and Prefetching Cache. In *the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, October 1992.
- [7] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. mei W. Hwu. Data Access Microarchitectures for Superscalar Processors with Compiler-Assisted Data Prefetching. In *Proceedings of the 24th International Symposium on Microarchitecture*, 1991.
- [8] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-Conscious Structure Layout. In *In Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999. ACM.
- [9] J. Collins, S. Sair, B. Calder, and D. Tullsen. Pointer Cache Assisted Prefetching. In *Proc. of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov 2002.

- [10] R. Cooksey, S. Jourdan, and D. Grunwald. A Stateless, Content-Directed Data Prefetching Mechanism. In *ASPLOS*, 2002.
- [11] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, Sept. 1994.
- [12] C. Ding and K. Kennedy. Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 229–241, Atlanta, GA, May 1999. ACM.
- [13] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, 1997.
- [14] G. Hinton and D. Sager and M. Upton and D. Boggs and D. Carmean and A. Kyker and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, (First Quarter), 2001.
- [15] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior. In *29th International Symposium on Computer Architecture*, May 2002.
- [16] Z. Hu, M. Martonosi, and S. Kaxiras. Tag Correlating Prefetchers. In *9th Intl. Symp. on High-Performance Computer Architecture*, Feb 2003.
- [17] C. J. Hughes. Prefetching Linked Data Structures in Systems with Merged DRAM-Logic. Master's thesis, University of Illinois at Urbana-Champaign, May 2000. Technical Report UIUCDCS-R-2001-2221.
- [18] P. Ibanez, V. Vinals, J. Briz, and M. Garzaran. Characterization and Improvement of Load/Store Cache-Based Prefetching. In *International Conference on Supercomputing*, pages 369–376, July 1998.
- [19] IBM. *IBM Power4 System Architecture White Paper*, 2002. <http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html>.
- [20] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. In *the 24th International Symposium on Computer Architecture*, pages 252–263, June 1997.
- [21] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *the 17th International Symposium on Computer Architecture*, pages 364–373, May 1990.

- [22] M. Karlsson, F. Dahlgren, and P. Stenstrom. A Prefetching Technique for Irregular Accesses to Linked Data Structures. In *Proceedings of the 6th International Conference on High Performance Computer Architecture*, Toulouse, France, January 2000.
- [23] M. Karlsson, F. Dahlgren, and P. Stenstrom. A Prefetching Technique for Irregular Accesses to Linked Data Structures. In *the 6th International Symposium on High-Performance Computer Architecture*, pages 206–217, January 2000.
- [24] A. C. Klaiber and H. M. Levy. An Architecture for Software-Controlled Data Prefetching. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 43–53, Toronto, Canada, May 1991. ACM.
- [25] D. Koufaty and J. Torrellas. Comparing Data Forwarding and Prefetching for Communication-Induced Misses in Shared-Memory MPs. In *International Conference on Supercomputing*, pages 53–60, July 1998.
- [26] D. Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of 8th International Symposium on Computer Architecture*, pages 81–87. ACM, May 1981.
- [27] A. Lai, C. Fide, and B. Falsafi. Dead-Block Prediction and Dead-Block Correlating Prefetchers. In *the 28th International Symposium on Computer Architecture*, pages 144–154, June 2001.
- [28] C.-K. Luk and T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Cambridge, MA, October 1996. ACM.
- [29] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.
- [30] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Newport Beach , LA, Oct. 1999.
- [31] T. Mowry. Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching. *Transactions on Computer Systems*, 16(1):55–92, February 1998.
- [32] T. Mowry and A. Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

- [33] T. Mowry, M. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 62–73, Boston, Massachusetts, October 1992. ACM.
- [34] T. C. Mowry. Tolerating Latency Through Software-Controlled Data Prefetching, PhD Thesis. Technical report, Stanford University, March 1994.
- [35] NVIDIA. Technical Brief: NVIDIA nForce Integrated Graphics Processor (IGP) and Dynamic Adaptive Speculative Pre-Processor (DASP). <http://www.nvidia.com/>.
- [36] S. Palacharla and R. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *the 21st International Symposium on Computer Architecture*, pages 24–33, April 1994.
- [37] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6), June 1990.
- [38] A. Roth, A. Moshovos, and G. Sohi. Dependence Based Prefetching for Linked Data Structures. In *the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, October 1998.
- [39] A. Roth and G. S. Sohi. Effective Jump-Pointer Prefetching for Linked Data Structures. In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, GA, May 1999.
- [40] C. Scheurich and M. Dubois. Concurrent Miss Resolution in Multiprocessor Caches. In *Proceedings of the 1988 International Conference on Parallel Processing*, 1998.
- [41] T. Sherwood, S. Sair, and B. Calder. Predictor-Directed Stream Buffers. In *the 33rd International Symposium on Microarchitecture*, pages 42–53, December 2000.
- [42] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *29th International Symposium on Computer Architecture (ISCA)*, May 2002.
- [43] Y. Solihin, J. Lee, and J. Torrellas. Correlation Prefetching with a User-Level Memory Thread. *IEEE Transactions on Parallel and Distributed Systems*, June 2003.
- [44] A. Stoutchinin, J. N. Amaral, G. R. Gao, J. C. Dehnert, S. Jain, and A. Douillet. Speculative Pointer Prefetching of Induction Pointers. In *Compiler Construction 2001, European Joint Conferences on Theory and Practice of Software*, Genova, Italy, April 2001.

- [45] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 1991.
- [46] C.-L. Yang and A. R. Lebeck. Push vs. Pull: Data Movement for Linked Data Structures. In *International Conference on Supercomputing*, pages 176–186, May 2000.
- [47] Z. Zhang and J. Torrellas. Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching. In *the 22nd International Symposium on Computer Architecture*, pages 188–199, June 1995.