Library-based Prefetching for Pointer-intensive Applications

Varun Malhotra and Christos Kozyrakis
Computer Systems Laboratory
Stanford University
{vsagar@gmail.com, christos@ee.stanford.edu}

February 2006

Abstract

Processor speed has been improving faster than memory latency for over two decades. Thus, an increased portion of execution time is spent stalling on loads, waiting for data from the memory hierarchy. Prefetching is an effective mechanism to hide memory latency for applications with low temporal locality. However, existing hardware prefetching techniques work well for array-based programs but not effective effective for pointer-intensive applications. Existing software prefetching techniques require recompilation or even modifications to application code before running on a new system with different hardware parameters.

In this paper, we exploit two opportunities to address this challenge. First, chip-multiprocessors (CMPs) are quickly becoming the norm, providing spare processors for prefetching tasks. Second, productivity reasons encourage programmers to make frequent use of standard or popular data-structure libraries. Hence, we propose a novel software prefetching scheme for pointer-based data-structures in which prefetching is performed by a helper thread included in the data-structure library code. The prefetch thread runs on a spare processor and relies on the library's knowledge of the data-structure type and access pattern to perform effective prefetching. All the development effort for prefetching is concentrated in the library code and is done once. The user application is not modified at all, as the library API remains the same. We implement and evaluate the library-based prefetch scheme. We demonstrate that it performs accurate prefetching and leads to an average reduction in execution time of 26% *for pointer-intensive benchmarks with appreciable memory stall time. Furthermore, we show that, using dynamic adaptation, the benefits from library-based prefetching are robust across a range of memory system and application parameters without the need for recompilation.*

1 Introduction

Fundamental trends in the semiconductor industry are causing the gap between memory and processor speeds to increase at a rapid pace. Cache hierarchies with multiple levels can hide memory access latency for programs with good *temporal* or *spatial* locality. However, caches are ineffective for applications with poor locality or large working sets. For such cases, data prefetching can reduce the impact of memory latency by initiating memory fetch well in advance so that they overlap with useful computation. Prefetching works well for array-based programs where spatial locality makes it easy to predict future accesses. Hardware prefetching is available in all processors today. It is highly effective for array-based computation, but has limited applicability to pointer-intensive programs which access heap data in irregular patterns.

The goal of this paper is to improve the performance of pointer-intensive, memory-bound applications through prefetching. Previous attempts have used elaborate hardware prefetching techniques that make specific assumptions about data-structure organization and access patterns that cannot be changed after chip fabrication. Compiler-

based prefetching has been limited by the inaccuracy of static pointer analysis. Moreover, recompilation is necessary before we run the application on a new system with different memory characteristics. Preexecution-based prefetching is also limited by the dependence on profiling information on the specific system parameters and datasets used in the profiling runs. In contrast, we propose a software prefetching technique that does not require significant hardware support, does not rely on pointer analysis, and provides robust results for a range of system parameters.

Our approach is based on two basic observations. First, chip multiprocessors (CMPs) are quickly becoming the norm for server and desktop systems. For the majority of current applications which are not parallel, CMPs provide idling processor cores for software prefetching tasks. Second, programmers frequently use popular or standard libraries of data-structures in their applications. Apart from the obvious link between code reuse and productivity, libraries typically include code that is better debugged and better tuned than what the programmer may be able to do within a short period of time. These two observations lead us to *library-based prefetching (LBP)*, where the library code forks a prefetch thread on a spare processor (if available) that performs software prefetching for the data-structures described in the library. LBP is transparent to the user code as the library API remains the same.

We have implemented library-based prefetching for two popular data-structure libraries, the C++ Standard Template Library (STL) and the LEDA C++ library [24, 22]. Prefetching pointer-intensive data-structures within library is effective as the library code knows the data-structure organization, the location of pointers, and even the type and progress of traversals. Our prefetch thread code exploits this information to perform accurate prefetching for pointer-based structures such as linked lists, k-ary trees, and graphs. The helper thread[1] tracks active data-structures, detects the type of the current traversal, and issues prefetch accesses ahead of the application. To maintain high accuracy, the library portion executing within the application thread communicates to the helper thread, information about data-structure accesses in a one-way, asynchronous manner.

LBP works with any existing application that uses the enhanced library. The user code is not modified because the library API remains the same. In the common case of dynamically linked libraries, LBP does not require recompilation and works with existing binaries. We also show that using dynamic adaptive control, LBP is robust to a wide range of system and application parameters such as dataset size and memory latency. Hence, the prefetch code in the library can be developed once and used successfully with multiple systems. Of course, LBP is limited to applications that use data-structure libraries. However, given its simplicity (no hardware or application changes) and extendability (can add to any data-structure library), we believe it is an important tool for improving the latency of memory-bound, sequential applications on CMP systems.

The specific contributions of our work are:

- We develop a framework for library-based prefetching in CMP systems that exploits data-structure information for accurate prefetching for pointer-intensive applications.
- We tune the prefetching framework, reduce its overhead and prevent excessive prefetching which might cause cache pollution. Furthermore, we develop an *adaptive control* scheme that automatically adjusts the speed of prefetching given the observed memory latency and application behavior.
- We evaluate library-based prefetching and obtain speedups of over 26% for the pointer-intensive memory-bound benchmarks. The results are robust across a wide range of memory latency parameters.

We should stress that critical insights of LBP are where prefetching is implemented (in the library) and how (transparent to user code, automatically tuned without recompilation). The prefetching patterns exploited by LBP are not different from those exploited through compiler-based or manual software prefetching. Essentially, this paper presents how libraries should be implemented to address the problem of memory latency.

[1]Throughout the text, we use the terms prefetch thread and helper thread interchangeably.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 presents the basic concepts behind library-based prefetching. Sections 4 and 5 describe the algorithms and tuning necessary to implement LBP. We present results of the experiments in Section 6, and conclude with Section 7.

2 Related Work

Initial work on hardware prefetching focused on sequential or strided array accesses [14, 25, 2, 26]. Correlation-based prefetching schemes used a prediction table to prefetch future accesses [13]. Mehrotra et al. extended the model to detect recurrent memory accesses [21]. Roth et al. proposed a dependence based prefetching scheme and jump-pointer framework for prefetching linked data-structures (LDS) [29, 30]. Subsequent improvements were proposed in [3, 18]. Recently, Chilimbi and Hirzel proposed a correlation-based software scheme [6]. Recent research includes work by Cooksey et al. on content directed prefetching [10]. Collins et al. used a pointer cache to assist in prefetching [7]. Yang and Lebeck used a hybrid software-hardware prefetching scheme [33]. Hardware prefetching for pointer-intensive data-structures has the advantage of access to dynamic program behavior. On the other hand, it has no specific knowledge of the data-structure organization and a simple change in the data-structure code can render a hardware prefetch engine useless.

There is also significant work on software-based prefetching [12, 27]. Mowry et al. and Luk et al. improve on previous schemes by performing static analysis to avoid unnecessary prefetch instructions [23, 17]. They also make the case for history-pointer prefetching wherein LDS nodes are augmented with prefetching pointer fields [17]. Even though the space overhead is low, maintaining these history pointers for dynamically changing data structures can introduce significant overhead. In general, compiler-based prefetching has to handle the difficulty of data-structure layout analysis and pointer disambiguation. Furthermore, static insertion of prefetch instructions cannot handle varying memory latencies across different systems or when using different dataset sizes.

More recently, *pre-execution* techniques that use idling hardware resources to run *prefetch threads* ahead of the application programs, have shown promise. Pre-execution can be controlled using hardware to extract pre-execution code from dynamic instruction traces [1, 8]. Alternately, compiler-assisted software techniques have been proposed to automatically extract pre-execution code from program source code [15] or compiled program binaries [9, 34]. Pre-execution techniques typically rely on profiling information which makes the code less portable across systems and datasets. They also require significant hardware modification as the pre-execution threads typically execute in a speculative manner. Compiler-based pre-execution can also run into the limitations of pointer-analysis or static prefetch placement.

Our work builds upon previous work. However, we propose to add prefetching code in common libraries. Libraries include significant macroscopic information about the data-structure organization and behavior to assist in making the prefetch decisions. The prefetch code development must be done only once by the library developer. All library users can benefit by merely dynamically linking the optimized library. Dynamic behavior is tracked without profiling techniques or extensive hardware modifications. Overall, our approach simplifies the problem of prefetching for the most common pointer-intensive data-structures such as linked-lists, trees, generalized graphs, and hash-tables, by moving the prefetching code and tuning problem into the library.

A similar though orthogonal approach is taken by STAPL to parallelize data structures [28]. The STAPL project focuses on parallelization and data-placement in NUMA systems while we focus on data prefetching.

3 Library-based Prefetching

3.1 Prefetching Overview

Our library-based prefetching (LBP) technique exploits an idling processor in a CMP system to run a prefetch thread (*PT*) included in a data-structure library. The prefetch thread is spawned by the library code the first time it is called to allocate a new data-structure object. Its goal is to track active data-structures, detect regular traversals,

and prefetch data-structure elements to reduce the observed memory latency for the main program (*MP*). The same prefetch thread remains alive until the main program exits.

Figure 1 presents an abstract implementation of the prefetch thread. The thread can be in one of three possible states. When no data-structures are actively accessed or no regular traversals are detected, the thread is in *idle* state. New data-structure accesses in the main program trigger a switch to the *active* state. The PT tracks the following information for each active data structure: the data-structure type (linked-list, tree etc.), the type of traversal (forward, breadth-first-search, etc.), a counter that indicates the element currently accessed by the main program, and a counter that indicates the element currently prefetched. The prefetch thread attempts to stay ahead of the main program in the current traversal by a number of elements refered to as the *stayAhead* distance. When the distance between PT and MP drops below *stayAhead* for one of the active data structures, the prefetch thread switches to the *prefetch* state and actively prefetches elements according the traversal type. Once the required distance is restored, the PT switches back to the active state. If no activity is detected for a certain period of time, the thread switches to idle state. Note that stayAhead parameter is a property of the data structure instance. Different data structures and even different instances of the *same* data structure can have different values of this parameter. As we discuss later, we dynamically adapt this parameter for each data-structure instance based on observed memory latency and application behavior.

```
state = IDLE; /* initial state */
while (!ExitFlag) {
  while(state==IDLE)
    if (new_DS_activity) state=ACTIVE;
    else wait;

  /* Initialize info for active data structure */
  void* local_DSptr = global_ActiveDSptr;
  local_iterator = local_DSptr→iterator;
  traversal = predict_traversal_type();
  stayAhead = init_stayAhead;
  PTcounter = 0;

  /* Compute number of elements to prefetch */
  diff = MPcounter−PTcounter+stayAhead;
  if (diff>0) state=PREFETCH;
  for (i = 0; i ≤ diff; i++, PTcounter++)
    Prefetch(local_iterator++);
    /* Note: ++ operator is overloaded;
    its implementation depends on DS and traversal*/

  /* set state ACTIVE or IDLE based on activity */
  state = set_next_state();
  /* adjust stayAhead based on dynamic information */
  stayAhead = adjust_stayAhead();
}
```

Figure 1. Abstract code for the prefetch thread. For simplicity, the code assumes a single active data-structure.

The PT code is written as part of the data-structure library. The code for detecting regular traversals and

```
list < int > ll;
list < int >:: iterator iter;
for (iter = ll.begin(); iter ! = ll.end(); iter++) work(*iter);
```

Figure 2. Example of forward traversal in a C++ linked list. The *begin*, $++$ and *end* methods include additional code for MP - PT communication.

prefetching elements utilizes knowledge about the algorithmic properties of the data-structure (potential traversals, characteristics that differentiate them) and the data organization used in the library (location and role of pointers in data-structure elements). The PT code requires a couple of Kbytes for a separate stack and some heap storage. The ordinary data-structure code remains mostly unmodified, with a few annotations added to communicate information to the prefetch thread as the main program accesses data-structure elements.

The prefetch thread may experience difficulties when a wrong traversal type is predicted or when the main program restructures the data-structure during the traversal (deletions, rebalancing, etc.). Difficulties manifest themselves either as errors on loads issued by PT (invalid addresses) or excessive miss rates in the main thread indicated by hardware performance counters. In both cases, we terminate the current traversal and switch the thread to the idle state. When the data-structure is traversed again, the PT will attempt prefetching again. The prefetch thread only loads data-structure elements and never writes. Hence, it cannot corrupt the data of the main program. To avoid unnecessary virtual memory faults, the PT uses non-faulting load instructions.

3.2 Main Program (MP) - Prefetch Thread (PT) Communication

The communication between the main program (MP) and the prefetch thread (PT) is a critical aspect of the system. Frequent communication is necessary for the PT to notice active data-structures, detect regular traversals, and track the MP's location. On the other hand, low overhead communication is needed to ensure that the benefits of prefetching are not cancelled out by excessive communication costs.

We use an one-way, asynchronous communication scheme that meets the above requirements. The MP communicates with the PT by writing to a set of shared variables. Two words are necessary to indicate a new traversal on some data-structure: a flag indicating new activity and a pointer to the specific data-structure. Another one or two words are needed per active data structure to track the progress of the traversal (see Section 4 for details). With linked-lists, for example, a word is used as a counter to indicate the main program's progress in a forward or backward traversal. The PT monitors these variables but never writes to them. Hence, no inter-thread synchronization is necessary as there is a single writer. There is also no need to explicitly notify the prefetch thread about updates to the shared variables. It is sufficient for PT to periodically poll these variables while in the idle or active states. The polling frequency can be set to hundreds of cycles. Essentially, the prefetch thread can afford to learn about a block of MP element accesses at once instead of polling frequently to catch each one individually. Hence, we can reduce significantly the cache to cache transfers necessary to implement this communication.

All the code necessary to implement MP-PT communication is located in the data-structure access methods of the library code. For instance, for the linked-list traversal in figure 2, it is necessary to modify the library code that implements the *begin*, $++$, and *end* methods. The extra code in *begin* notifies the prefetch thread about a new traversal to the *ll* data-structure by setting one flag and one pointer. Each execution of $++$ increments the counter indicating the main program's location within the linked-list. Another shared memory variable set by the MP tells PT about the direction of traversal of the MP. Based on the value of this *direction* variable, *end* behaves similar to *begin* to handle backward traversals or acts as a termination trigger for the prefetch thread (for forward traversals).

Overall, we implemented the efficient MP-PT communication mechanism by introducing few lines of code per data-structure type in the implemented library. Section 6 shows that the run-time communication overhead for the

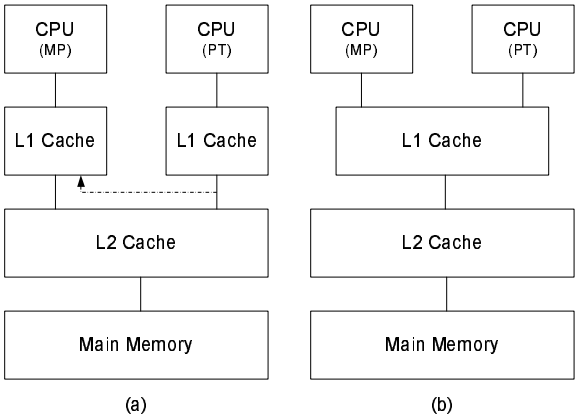

Figure 3. CMP processor alternatives for library-based prefetching. The optional forwarding path in (a) allows L1 refills for the one CPU to be pushed into the prefetch buffer of the other CPU.

main thread is practically negligible due to its one-way asynchronous nature.

3.3 Architecture Considerations

The proposed prefetching scheme is software-based. Nevertheless, the features of the CMP it runs on can affect its performance. Figure 3 shows the two alternative architectures that library-based prefetching could be used with. Figure 3.a presents a conventional CMP where the two CPUs for the MP and the PT share the L2 cache but have private L1. In this system, LPB hides L2 miss latency, which is a critical problem with wide-issue processors. L1 misses can be hidden if we allow cache refills from the CPU executing the prefetch thread to be *forwarded* into the prefetch buffer of the CPU executing the main program. This is a relatively simple modification that can be selectively activated when library-based prefetching is on.

Figure 3.b presents an alternative organization where the two CPUs share a single, highly banked L1-cache. Such an organization is possible when partitioning a large clustered processor like TRIPS [31] into multiple smaller CPUs or when using a multithreaded CPU (2 virtual CPUs). With such an organization, no additional hardware changes are needed to target both L1 and L2 misses. In this case, there can be some negative interference between the MP and PT in the L1-cache. Since the PT private footprint is small, if prefetching is accurate inteference is minimal.

3.4 Discussion

LBP has several advantages over conventional prefetching approaches. Compared to hardware-based approaches, it works without significant hardware changes, hence it does not pose another complexity burden for the processor design and verification. Even though it is software-based, it can exploit dynamic information to regulate how far ahead the helper thread runs or if it runs at all. The MP communicates its access pattern to the PT which can also access hardware counters that track miss rates. Compared to static compilation techniques, LBP does not require extensive pointer analysis capabilities. All the information about data-structure organization and regular traversals is available as the prefetching is implemented by the library developer. Furthermore, LBP works well even in the case of varying hardware parameters (cache sizes, cache and memory latencies) as the prefetch thread dynamically regulates itself (see Section 5.1). Compared to pre-execution based techniques, our approach does not rely on profiling information, hence it is easier to use on a larger variety of systems and datasets. Finally, compared to manual prefetching in the source code, our approach moves the burden from every application writer to a single, expert library developer. The development effort is a one time affair, and the prefetch thread does not have to be tuned

for every chip as well as for every application. The code that uses the library with prefetching capabilities remains the same. Assuming dynamic linking of the data-structure library, LBP does not even require recompilation.

The shortcomings of LBP are the following. First, it takes up a whole CPU or thread in a CMP system. Nevertheless, as most programs are currently sequential or not highly parallel and unable to utilize all CPUs in a useful way, we believe this is a reasonable trade-off to achieve performance improvements for single-threaded programs. If current trends continue, the number of CPUs per chip will keep increaing, providing further resources for LBP. Furthermore, the prefetch thread can be easily turned off (disabled) if the resources are used for other purposes without functionality implications for the library code or the application that uses the library. Second, LBP is limited to common data-structures and regular access patterns. There will always be special cases that require unconventional data-structures not captured by common libraries. However, a significant number of developers use common data-structure libraries for productivity purposes, hence our approach has a large audience to serve.

4 Algorithms for Prefetching

This section explains the traversals prefetched for each data-structure type in the implemented library. Our prefetching technique focuses on some of the commonly used, pointer-intensive data-structures like linked-lists (single and double linked), trees (n-ary, balanced and unbalanced), hash-tables, and general graphs. The approach can be generalized to other pointer-intensive data-structures with common traversal patterns. Previous work on software prefetching has thoroughly discused the prefetching opportunities for each data-structure type (see Section 2). Hence, our discussion focuses on how we handle each traversal in LBP.

Our specific implementation is based on the C++ Standard Template Library (STL), a highly parameterized generic library that is part of the C++ standard [24]. Since STL includes a small number of data-structures, we have extended it to include additional data-structures following the API specified by LEDA [22], another popular C++ data-structure library. Nevertheless, the traversal pattern detection and prefetching algorithms are not tied in any way to the specific libraries. They are applicable to other languages (e.g. Java) and other libraries.

Linked Lists

The traversals for linked lists are easy to handle as movement is only possible in one dimension. The only possible traversals are forward ($++$) and backward ($--$). The MP communicates to the PT by incrementing or decrementing a counter as it accesses elements in the list. This information is sufficient for the PT to determine the traversal direction for prefetches and to adjust its speed relative to the main program.

General Trees

Trees are somewhat more complicated. The regular access patterns we exploit are depth-first (pre-order) and breadth-first (level-order) traversals. There are two ways to detect these patterns. One is to automatically detect them in the PT using information from the main program. The MP updates a variable in shared memory by shifting in a few bits that describe the pointers used during its traversal ($logn + 1$ bits for an n-ary tree). It is straightforward to separate a depth-first from a breadth-first traversal by looking at the top few bits in this variable. An alternative is for the library to provide a high-level API that allows users to specifically describe regular tree traversals. Example APIs from the LEDA library and our extensions are shown in figure 4. It includes four high-level methods that describe the traversal and the work applied on each tree element. For depth-first traversals, three separate methods are needed depending on when work is applied to tree nodes. Such a high-level API makes it easier for the programmer to manipulate trees. It also allows the library code to explicitly notify the prefetch thread about the traversal type and save the overhead of automatic detection.

Once the traversal type has been detected, MP communicates its progress to PT through a simple counter; and that is sufficient to issue prefetches down the right path and regulate the helper thread. The prefetch thread allocates

```
void BFS(void(*func)(iterator iter, void *arg1), void *arg);
void in_order(void(*func)(iterator iter, void *arg1), void *arg);
void pre_order(void(*func)(iterator iter, void *arg1), void *arg);
void post_order(void(*func)(iterator iter, void *arg1), void *arg);
```

Figure 4. High-level methods for common tree-traversals in a data-structure library. Function *func* does the processing on every node.

and implements a queue or a stack with its code in order to prefetch the tree elements in a desired order. This may introduce some additional misses in caches that reduce the usefulness of prefetching.

Hashtables

Hashtables are primarily used for storing and retrieving objects based on *key* values. Most implementations comprise of an array of buckets, with each bucket containing pointer to a linked list (*open chaining*). For any search query with an input key, the MP indexes into this array of buckets based on the value of the key. Since accesses into the array of buckets can be completely arbitrary, it is difficult to prefetch the array location where the MP would index into. The library can attempt to lock non-empty buckets into the cache, but we have not explored this option in this study. Once the MP decides on which bucket to search for, the linked list corresponding to that bucket can be prefetched by the PT using the same techniques as described in subsection 4. Nevertheless, these linked-lists are often short and prefetching is of limited value.

General Graphs

Nearly all applications that use *general graphs* visit a *node* or an *edge* and then explore its neighbors in some order. The API for graphs in most popular libraries supports exactly these kinds of traversals [22]). The PT included in our library implements *neighborhood prefetching* for graphs. The PT *periodically* attempts to prefetch edges and nodes in a small neighborhood around the current location of the main program in the graph. It is likely that there is the data that the MP will access in the near future. If some information is available through the API about the traversal order, the PT can focus on a specific direction within the neighborhood. Otherwise, the best we can do is to dynamically regulate the size of the neighborhood based on the main program speed and the observed misses.

5 Extension and Tuning

5.1 Adaptive Control System

One major concern is to choose how far ahead of the main program should the prefetch thread be allowed to run. Recall from section 3.1 that the *stayAhead* parameter regulates distance between the PT and the MP. A large value may lead to conflict misses and cache pollution. A small value reduces the effectiveness of prefetching and increases the overhead of MP-PT communication. Furthermore, a single value is unlikely to be optimal for all systems and all applications.

To counter this problem, the prefetch thread in our implementation can adaptively adjust the stayAhead parameter for each active data-structure instance. Hence, two tree objects of the same type may have different stayAhead parameters. This is crucial as it allows us to have a single implementation of the library regardless of system details (cache hierarchy, memory latency, etc.) or the application characteristics (dataset size). The PT adapts itself based on the system and workload requirements. When the MP is accessing a data structure which invokes prefetching code, the PT monitors cache misses incurred by the main program. This system relies on sampling the processor

hardware counters, which are set up to count misses. Such counters are available in all modern processors. The functionality for reading hardware performance counters is provided by many available libraries like `PAPI` [16]. The prefetching thread reads these counters while it is idle (or not fetching data) since it needs only approximate information about these counters.

The value of stayAhead is adjusted when the PT observes that the number of cache misses incurred by the MP are increasing. This is trully dynamic adaptation, not hand crafting of our system. It is crucial to determine whether the cache misses in MP are due to cache pollution (causing conflict misses) or whether they are compulsory misses which could not be hidden by the prefetch thread. In the first case, the value of the stayAhead parameter is too large while in the latter case, it is too small. To address the issue without complicated control techniques with high overhead, we start with a *small* value of stayAhead parameter (10 for our experiments). If PT observes cache misses in MP then the value of stayAhead parameter is slowly incremented until increased cache misses are noticed again. This time, they are likely due to conflicts, and hence the stayAhead parameter is decremented accordingly. The code the implement this control system is only a few lines long.

5.2 Handling Data Pointers

A common problem with many programs is that the data structures might not store the actual *data objects* but pointers to the data objects. We can annotate the prefetching code to prefetch such pointers using prefetch instructions (non-blocking loads). For strongly typed-languages, the contents inside the data structures can be checked for pointer-types, hence data pointer prefetching can be highly accurate. For languages like C and C++, it is not possible to distinguish between pointers and other data types stored in the data-structure within the library code without the help of the compiler. In our implementation, when the first few elements are fetched we scan through their contents and use them as pointers for prefetch instructions. Note that prefetch instructions cause no exceptions with invalid address. We apply a simple filter that rejects elements whose contents do not look like heap addresses or look more like small negative constants [10]. After we have visited the first few elements, the PT has a good idea of where the pointers may exist in each element (if they exist at all) and can implement data pointer prefetching without the overhead of additional scanning. In our implementation, scanning is relatively inexpensive as we use an architecture with aligned memory addresses.

Note that, even if we know where the pointers are, data pointer prefetching is not always beneficial. Just because there are some pointers in an element does not mean that all pointers will be followed during the current traversal. Hence, data pointer prefetching may potentially lead to cache pollution.

5.3 Parallel Traversals and Nested Data-structures

Our current prefetch thread implementation is able to handle multiple active data-structures object traversed in parallel. Each data-structure has a separate state and an independent stayAhead parameters. Prefetch accesses for multiple active data-structures are interleaved.

Another interesting case is nested data-structures, such as trees of linked-lists. LBP is applicable in this case as well, even through we have not implemented it in the current version of the library. Note that prefetching nested data-structures is of interest only if both the inner and the outer data-structures contribute to cache misses. If one of them is much larger and dominates misses, single data-structure prefetching is sufficient.

6 Evaluation

6.1 Methodology

We evaluate LPB using a simulated CMP system with two wide-issue processor cores. Table 1 provides the details of the CMP configuration. We do not simulate a larger CMP as we need just one additional thread for LBP. Nevertheless, LBP is applicable to CMPs with more cores.

Processor Core (2 in the CMP)	
Issue Width	3
Pipeline Stages	7
Memory System	
Private L1 Instr. Cache	16 KB, 2-way set-associative, 2-cycle hit time (pipelined)
Private L1 Data Cache	16 KB, 2-way set-associative, 2-cycle hit time (pipelined)
Shared L2 Cache	512 KB, 8-way set-associative, 12-cycle hit time
Cache Line Size	32 bytes
Main Memory Latency	200 cycles

Table 1. Simulated machine parameters

Benchmark	**Data Structures Used**	**Dataset Size**
Bisort	Binary Tree	100,000 integers
Em3D	Linked Lists	2000 nodes
Health	Linked Lists	max level = 5 max time = 500
Mst	Hashtables Linked List	512 nodes
Perimeter	Quadtree	4096x4096 image
TreeAdd	Binary Tree	2048 nodes
Tsp	Binary Tree Linked Lists	60,000 cities
Bfs	Binary Tree	1024 nodes
Apsp	General Graph	400 nodes

Table 2. Description of the Olden benchmarks.

We implemented LBP by extending the C++ Standard Template Library (STL). The library, including the prefetch thread, was coded in C++ and compiled with an optimizing compiler at the -O3 level. The library code uses a standard POSIX thread library to fork the prefetch thread. The conventional part of the library code that runs within the main application thread was modified in two ways: (a) We added code to fork the prefetch thread the first time library code is invoked, and (b) we add code to communicate information from the library code in the main program to the prefetching thread. The modifications included approximately 20 lines of code per data-structure in the library. All the development effort for LBP is concentrated within the library code. The library API is not modified, hence the application code remains the same.

The prefetch thread, including the adaptive control code, was carefully optimized to reduce its overhead. Unlike the main library code that uses several function calls to implement the flexible API and support generic data-types, the prefetch code is stream-lined with no function calls. Hence, the prefetch thread can access elements much faster than main program, even when the main program performs no other work per element it accesses through the main library code. For that reason, we did not find it necessary to modify the STL library to insert *jump pointers* in its data-structure organization. Such pointers would be useful, but were not necessary.

In our evaluation, we first used a set of microbenchmarks to tune the prefetching algorithms for the various data-structures and the adaptive mechanism for adjusting the *stayAhead* parameter for different data structures. The microbenchmarks allow us to create workloads with varying traversal patterns, work per data-structure element, and frequency of misses, and help in fine tuning the adaptive mechanism. Due to limited space, we do not present these results. For an overall evaluation, we use the Olden suite of pointer-intensive benchmarks, which also facilitates comparisons with the previous prefetching literature which uses these benchmarks almost exclusively [5]. Table 2 describes the benchmarks used in this study. Out of the 10 Olden benchmarks, we evaluate seven.

Power and *Barnes-Hut* were left out as these programs use heterogenous[2] trees which are not commonly supported in popular data-structure libraries. *Voronoi* was left out as it runs virtually without cache misses. We have added two more benchmarks: (a) BFS involving a breadth-first traversal on a binary tree, and (b) APSP (all pairs shortest path) which solves all-pairs shortest path problem on a general graph. The APSP benchmark is a modified version of the code available in Andrew Goldberg's Network Optimization Library [11]. The benchmarks use our STL-like C++ based library but we did not tune or change them in any other way.

Some of the benchmarks in Table 2 use data pointers inside data structure objects, which the PT tries to identify and prefetch as well. We use the scheme proposed by Cooksey et al. to reduce the number of prefetch requests issued for incorrectly identified data pointers [10]. It comprises of a couple of filters: one for checking the higher order bits with a known pointer (if different, then not a pointer) and another one for filtering out small floating point addresses which often look like heap pointers. The pointer determination filters are able to prefetch data pointers efficiently and accurately without much overhead. The overhead is low because the detection is performed only during the beginning of the traversal.

6.2 Performance Improvement

Figure 5 presents the performance improvement with library-based prefetching. Execution time both without prefetching, labelled *NP* and with LBP prefetching labelled *P*, is reported. The portion of the bars labelled *exec cycles*, is the execution time without prefetching with an ideal data memory system. *Overhead* is the increase in execution time in the main program due to MP-PT communication. The remaining portion of the execution time is due to memory stalls (L1 and L2 misses). Library-based prefetching reduces the execution time for all benchmarks that spend a significant portion of time stalling for memory, namely *treeAdd*, *em3d*, *health*, *mst* and *bfs*). The execution time is reduced by 23% (*mst*, *treeAdd*) and 40% (*em3d*), with a reduction of 26% on average for these 5 memory intensive applications. No benchmark was slowed down due to prefetching overhead even those with minimal stall time to begin with.

For *mst*, we get a significant improvement from successfully prefetching the linked-lists. On the other hand, our approach is not particularly effective with hashtables as the linked lists in each bucket are short (2 to 3 elements). For *treeAdd* and *bfs*, we see good performance improvement by prefetching the binary tree. Some additional memory stall time remains due to accesses to other variables and the time needed by the prefetch thread to run ahead initially. For *em3d*, the remaining memory stalls are due to the data included in linked-list element. Each element contains a pointer to an array of approximately 10 elements. While we prefetch the pointer, we only get the first few elements of each array. When the remaining elements are accessed, misses still occur. Maximum speedup is achieved for *health* where LBP is able to hide most of the memory latency. Interesting results are observed for the *apsp* benchmark which accesses a general graph. Even though the memory stall time is reduced by half, overhead of communication for neighborhood prefetching (see Section 4) is also high. Nonetheless, there is 5% reduction in execution time. Memory stalls for *perimeter* are significantly reduced but their contribution to execution time is small. Finally, we achieve small memory latency reductions for *bisort* and *tsp* but they are offset by the MP-PT communication overhead. The opportunity for performance improvement was low for the last three benchmarks as they either have good hit rates or amortize the cost of each miss over a significant amount of work.

6.3 Prefetching Accuracy

To gain further insight into how effective LBP is, Figure 6 reports the cache miss *coverage*. Coverage is defined as the percentage of cache lines brought into the cache by the PT that is accessed by the MP before it are replaced. The coverage measurements also includes addresses prefetched that are accessed by the main thread before the refill is completed (partially successful prefetching). Figure 6 demonstratres that understanding the traversal type

[2]Data structures are homogeneous when all the data structure nodes/objects are of the same type. Power and Barnes-Hut have tree-like data structures where the individual tree nodes can be of different types.

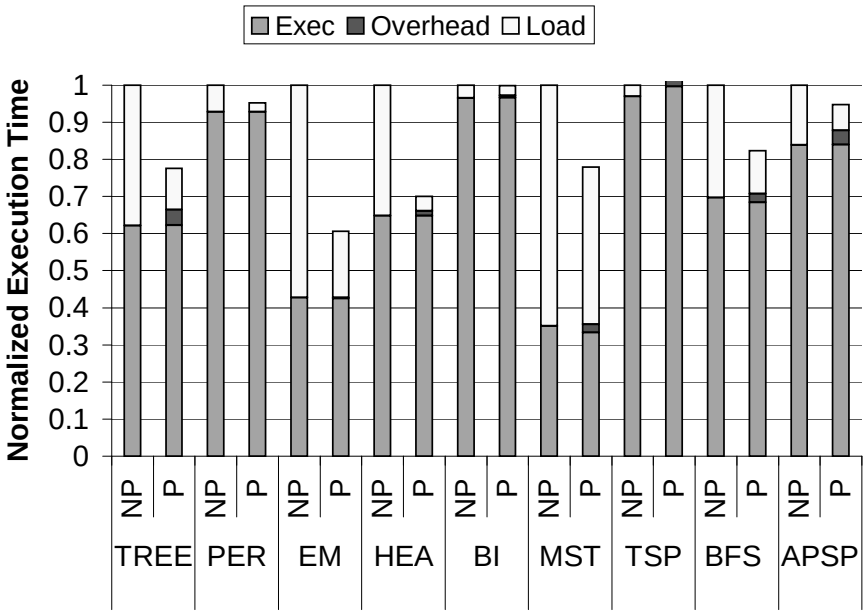

Figure 5. Normalized execution time broken into execution, overhead and memory stall components. The NP and P bars specify performance without prefetching and with LBP prefetching respectively. One (1) is the sequential execution time without prefetching.

and the data-structure organization in the library code can lead to highly accurate prefetch accesses even without help from hardware. The coverage for L2 prefetches is more that 95% for eight out of nine benchmarks. *Health* operates on several linked lists which are highly dynamic and incur some overhead by frequently switching between different active data-structures in the prefetch thread.

L1 coverage is lower than that of the L2 but is still high. On average 87% of the prefetches are later used by the main thread. The limited L1 capacity and associativity causes a certain portion of the prefetched data to be evicted before they are used. Furthermore, prefetching the data pointers for certain benchmarks creates coverage issues as not all pointers are used by the main program in every traversal (*em3d* and *health*).

6.4 Prefetch Timeliness

Another metric which gives greater insight into the effectiveness of prefetching is *timeliness*. It is a measure of how early a prefetch brings data into the cache before it is used by the application. Positive timeliness implies that the prefetched data arrived early, while negative timeliness implies that prefetch was issued late and was only partially successful at best. If the data is brought into the cache too early; it can lead to cache pollution. We look at two applications (*em3d* and *treeAdd*) that use different types of data structures. Figure 7(a) plots the distribution of time difference between refill of prefetch loads and its use by the corresponding load of the MP for *em3d*. A small fraction of prefetch loads fall into the negative side of the distribution. Work per element on the linked list for *em3d* is found to be roughly $200-250$ cycles (excluding memory latency). Thus, nearly 20% of the prefetch loads are refilled while the PT is 6 ($\frac{1200}{MemoryLatency} = \frac{1200}{200}$) elements ahead of the MP. Overall, most of the prefetch accesses are timely with a small fraction being rather late.

The *treeAdd* benchmark is more taxing for PT as a big portion of the refills from prefetch loads are separated from the corresponding subsequent loads by 1 memory access (200 cycles). The problem with tree data structures can be explained by a small example. Suppose $++$ for the tree data structure has been overridden to give the next element in depth-first (DFS) traversal. Unlike a linked list where every $++$ incurs the same amount of cost, $++$

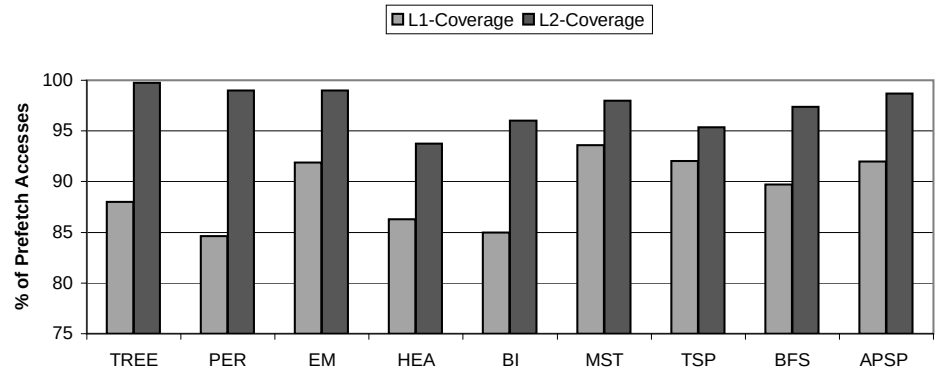

Figure 6. Percentage of prefetches which are completely covered the subsequent corresponding main program loads

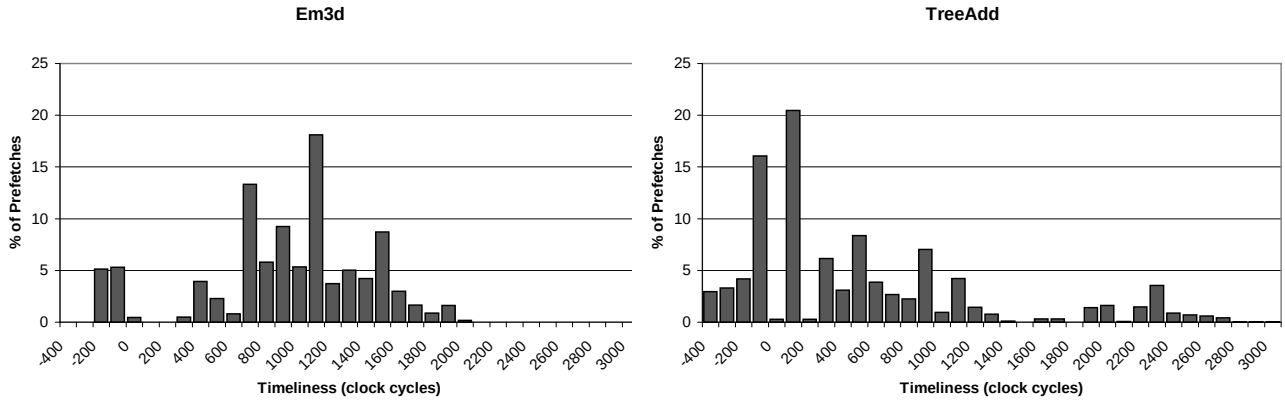

Figure 7. Timeliness of em3d & treeAdd prefetches (histogram).

method call in a tree DFS access can incur variable cost depending on the current position of the PT. Thus, the PT has a hard time to stay ahead of the MP when it has to follow long paths of pointers to get to the next element in the traversal. As seen in figure 7, the PT for *treeAdd* has a harder time staying ahead of the MP as compared to that of *em3d*. Nevertheless, we still see that majority of data from prefetch loads arrive well in advance for the corresponding loads (positive timeliness).

6.4.1 Bandwidth Requirements

We analyze the effect of LBP on the bandwidth used in the system. Figure 8 presents the average bandwidth at the L2 cache and main the memory. The *NP* and *P* bars specify performance without prefetching and with LBP prefetching respectively. Due to execution time being reduced by up to 40% with prefetching, the average bandwidth ($\frac{total\ bytes\ transferred}{total\ execution\ time}$) requirement nearly doubles. Even so, the bandwidth requirements in bytes per cycle are quite low in absolute numbers. Because our library-based prefetching scheme is accurate, the problem of redundant loads is lessened and does not cause additional bandwidth increases. The slight increase in bandwidth is due to the same number of loads being issued in a smaller time frame as execution time decreases appreciably as a result of LBP.

6.5 Effect of Main Memory Latency

We varied main memory latency to gain insight into how LBP would perform in different systems (current and future). We present results on two benchmarks: one with linked lists (*em3d*) and another operating on a binary

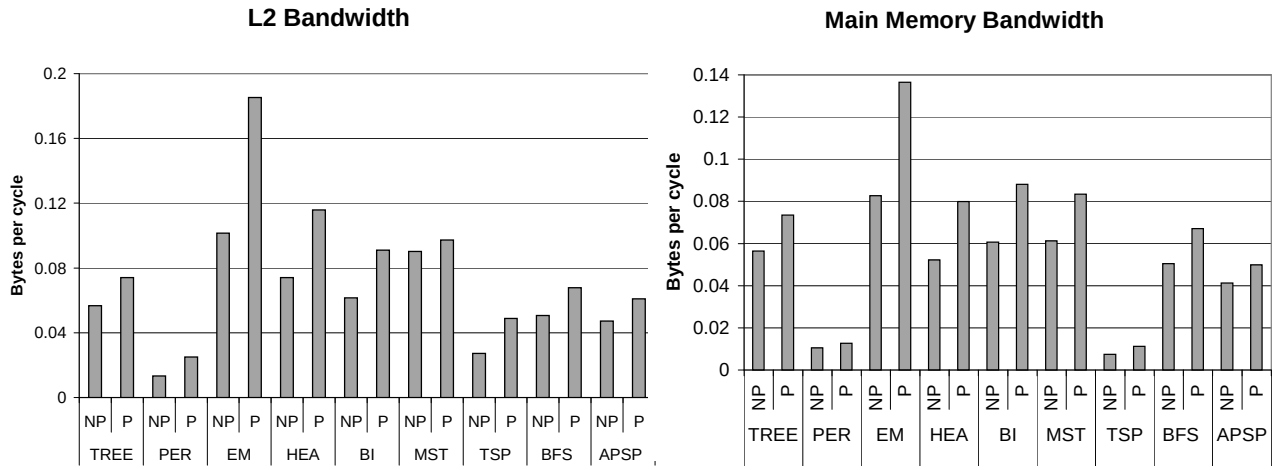

Figure 8. Average bandwidth requirements at the L2 cache and the main memory (bytes/cycle).

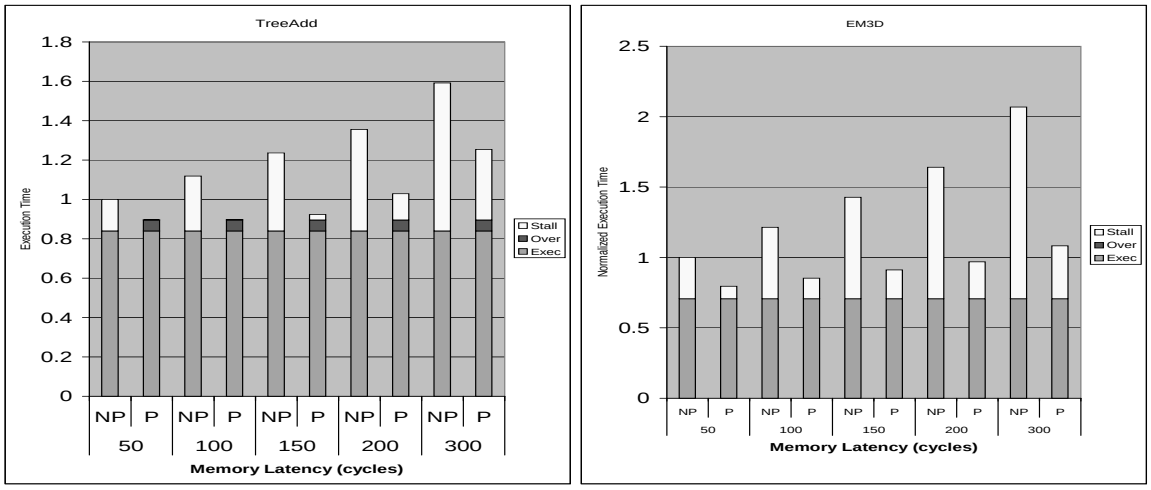

Figure 9. Effect of varying main memory latency on treeAdd & em3d

tree (*treeadd*). As memory latency increases, the prefetch thread automatically increases the stayAhead parameter to be able to hide memory latency. Nevertheless, it will take an increased amount of time for the prefetch thread to achieve the desired distance from the main thread, hence some increase in stalling will occur, particularly for reduced traversals.

With *treeadd*, figure 9 shows that when memory latency is small (50-100 cycles) the load-stall time is almost completely hidden by the prefetching system. However, the effect of memory latency increase affects how far ahead the PT is from the MP. As memory latency increases, the MP catches up with the PT in the traversal more rapidly as the PT takes longer to fetch each element. This effect is clearly visible in the case of *treeadd*. In *em3d*, there is sufficient work per element in the main program. Thus, even when memory latency is increased from 50-300 cycles, the prefetch thread is able to stay ahead of the main program.

These results suggest the LBP is effective across different systems with different memory parameters. They also suggest that LBP is even more worthwhile in light of increasing memory-processor speed gap.

7 Conclusions

This paper presents a library-based approach to prefetching data for pointer-based data-structures such as lists, trees, and graphs using spare processors in a CMP system. We demonstrate the efficacy of this approach on a set of pointer-intensive programs, achieving up to a 40% reduction in execution time. We observe that the prefetches issued by our approach are both *timely* and *accurate* in hiding most of the cache misses coming from data structure traversals. Our infrastructure also provides the flexibility to dynamically adapt at runtime to different workloads, and even scale back prefetching in cases where it would negatively impact performance.

Since the prefetching code is a part of the library, all the users of the library can benefit by re-compiling their code with a standard compiler or by dynamically linking the optimized library with their code. Hence, library-based prefetching is a simple and effective method to reduce the execution time for memory bound applications that make use of data-structure libraries. Library programmers should consider constructing libraries for the future with LBP and other such performance optimizations that can exploit the available resources in CMPs in ways transparent to user code.

References

[1] M. Annavaram, J. M. Patel and E. S. Davidson, *Data prefetching by dependence graph precomputation*, International Symposium on Computer Architecture (2001): 52–61

[2] J.-L. Baer and T.-F. Chen, *An effective on-chip preloading scheme to reduce data access penalty*, International Conference on Supercomputing (1991): 176–186

[3] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz and U. Weiser, *Correlated Load-Address Predictors*, International Symposium on Computer Architecture (1999): 54–63

[4] D. Butenhof, *Programming With POSIX Threads*, Addison Wesley, 1st Edition (1997)

[5] M. C. Carlisle and A. Rogers, *Software Caching and Computation Migration in Olden*, Principles and Practice of Parallel Programming (1995): 29–38

[6] T. M. Chilimbi, M. Hirzel, *Dynamic Hot Data Stream Prefetching for General-Purpose Programs*, Programming Language Design and Implementation (2002): 199–209

[7] J. D. Collins, S. Sair, B. Calder and D. M. Tullsen, *Pointer cache assisted prefetching*, International Symposium on Microarchitecture (2002): 62–73

[8] J. D. Collins, D. M. Tullsen, H. Wang and J. P. Shen, *Dynamic speculative precomputation*, International Symposium on Microarchitecture (2001): 306–317

[9] J. D. Collins, H. Wang, D. M. Tullsen, C. J. Hughes, Y.-F. Lee, D. M. Lavery and J. P. Shen, *Speculative precomputation: long-range prefetching of delinquent loads*, International Symposium on Computer Architecture (2001): 14–25

[10] R. Cooksey, S. Jourdan and D. Grunwald, *A stateless, content-directed data prefetching mechanism*, Architectural Support for Programming Languages and Operating Systems (2002): 279–290

[11] A. Goldberg, *Network Optimization Library*, http://www.avglab.com/andrew/soft.html

[12] E. H. Gornish, D. Granston and A. V. Veidenbaum, *Compiler-directed data prefetching in multiprocessors with memory hierarchies*, International Conference on Supercomputing (1990): 354–368

[13] D. Joseph and D. Grunwald, *Prefetching Using Markov Predictors*, International Symposium on Computer Architecture (1997): 252–263

[14] N. P. Jouppi, *Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers*, International Symposium on Computer Architecture (1990): 364–373

[15] D. Kim and D. Yeung, *Design and evaluation of compiler algorithms for pre-execution*, Architectural Support for Programming Languages and Operating Systems (2002): 159–170

[16] London, K., Dongarra, J., Moore, S., Mucci, P., Seymour, K. and Spencer, T., *End-user Tools for Application Performance Analysis, Using Hardware Counters*, International Conference on Parallel and Distributed Computing Systems, Dallas, TX, August 8–10, 2001

[17] C.-K. Luk and T. C. Mowry, *Compiler-Based Prefetching for Recursive Data Structures*, Architectural Support for Programming Languages and Operating Systems (1996): 222-233

[18] M. Karlsson, F. Dahlgren, P. Stenstrm, *A Prefetching Technique for Irregular Accesses to Linked Data Structures*, High-Performance Computer Architecture 2000: 206-217

[19] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally and M. Horowitz, *Smart Memories: a modular reconfigurable architecture*, International Symposium on Computer Architecture, Vol 15, 161–171 (2000)
[20] K. S. McKinley, S. Carr and C.-W. Tseng, *Improving Data Locality with Loop Transformations*, ACM Transactions on Programming Languages and Systems 18(4): 424–453 (1996)
[21] S. Mehrotra and L. Harrison, *Examination of a memory access classification scheme for pointer-intensive and numeric programs*, International Conference on Supercomputing (1996), 133–140
[22] K. Mehlhorn and S. Naher, *The LEDA Platform of Combinatorial and Geometric Computing*, Cambridge University Press, (1999)
[23] T. C. Mowry, M. S. Lam and A. Gupta, *Design and Evaluation of a Compiler Algorithm for Prefetching*, Architectural Support for Programming Languages and Operating Systems (1992): 62–73
[24] D. R. Musser, G. J. Derge and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library (2nd Edition)*, Addison-Wesley, Reading, MA, (2001)
[25] S. Palacharla and R. E. Kessler, *Evaluating Stream Buffers as a Secondary Cache Replacement*, International Symposium on Computer Architecture (1994): 24–33
[26] S. S. Pinter and A. Yoaz, *Tango: A Hardware-Based Data Prefetching Technique for Superscalar Processors*, International Symposium on Microarchitecture (1996): 214–225
[27] A. K. Porterfield, *Software Methods for Improvement of Cache Performance on Supercomputer Applications*, PhD thesis, Department of Computer Science, Rice University (1989)
[28] L. Rauchwerger, F. Arzu and K. Ouchi, *Standard Templates Adaptive Parallel library(STAPL)*, Languages, Compilers, and Run-Time Systems for Scalable Computers (1998): 402–409
[29] A. Roth, A. Moshovos and G. S. Sohi, *Dependance Based Prefetching for Linked Data Structures*, Architectural Support for Programming Languages and Operating Systems (1998): 115–126
[30] A. Roth and G. S. Sohi, *Effective Jump-Pointer Prefetching for Linked Data Structures*, International Symposium on Computer Architecture (1999): 111–121
[31] K. Sankaralingam, R. Nagarajan, H. Liu, J. Huh, C.K. Kim D. Burger, S.W. Keckler, and C.R. Moore, *Exploiting ILP, TLP, and DLP Using Polymorphism in the TRIPS Architecture*, 30th Annual International Symposium on Computer Architecture, pp. 422–433, June 2003.
[32] A. J. Smith, *Cache Memories*, ACM Computing Surveys, 14(3): 473–530 (1982)
[33] C.-L. Yang and A. R. Lebeck, *Push vs. pull: data movement for linked data structures*, International Conference on Supercomputing (2000): 176–186
[34] C. B. Zilles and G. S. Sohi, *Execution-based prediction using speculative slices*, International Symposium on Computer Architecture (2001): 2–13