Effective Jump-Pointer Prefetching for Linked Data Structures

Amir Roth and Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin, Madison
{amir, sohi}@cs.wisc.edu

Abstract

Current techniques for prefetching linked data structures (LDS) exploit the work available in one loop iteration or recursive call to overlap pointer chasing latency. Jump-pointers, which provide direct access to non-adjacent nodes, can be used for prefetching when loop and recursive procedure bodies are small and do not have sufficient work to overlap a long latency. This paper describes a framework for jump-pointer prefetching (JPP) that supports four prefetching idioms: queue, full, chain, and root jumping and three implementations: software-only, hardware-only, and a cooperative software/hardware technique. On a suite of pointer intensive programs, jump-pointer prefetching reduces memory stall time by 72% for software, 83% for cooperative and 55% for hardware, producing speedups of 15%, 20% and 22% respectively.

1 Introduction

Linked data structures (LDS) are common in many applications, and their importance is growing with the spread of object-oriented programming. The popularity of LDS stems from their flexibility, not their performance. LDS access, often referred to as *pointer-chasing*, entails chains of data dependent loads that serialize address generation and memory access. In traversing an LDS, these loads often form the program's critical path. Consequently, when they miss in the cache, they can severely limit parallelism and degrade performance.

Prefetching is one way to hide LDS load latency and recover performance. *Address prediction based* techniques can generate addresses in non-serial fashion, prefetch nodes arbitrarily far ahead of their anticipated use and tolerate long latencies. However, LDS access streams rarely display the high levels of arithmetic regularity required to support accurate address prediction.

Recently proposed *scheduling based* techniques [11, 16] prefetch nodes serially but attack issue delays that aggravate serialized latencies by issuing LDS loads as soon as their inputs are ready. Scheduling methods can pre-calculate LDS addresses accurately, but their pace is dictated by the critical path through the pointer chain. Scheduling methods are inadequate when the amount of work available for overlapping with the critical chain is limited, due to either a tight loop or a slow memory. Handling these situations, which will worsen as the processor/memory speed gap grows, requires a mechanism that can address and prefetch arbitrary LDS nodes.

To illustrate our point, Figure 1(a) shows a list traversal loop (e.g., `for (l = list; l; l = l->next) ...`) with the long latency of the induction loads (instances of **`l = l->next`**) exposed. Scheduling methods hide this latency by issuing the induction load early in the iteration (Figure 1(b)). For short iterations or long latencies (Figure 1(c)), an induction load will stall the next iteration no matter how early within its own iteration it issues. For full efficiency, it must be overlapped with work from *multiple* iterations.

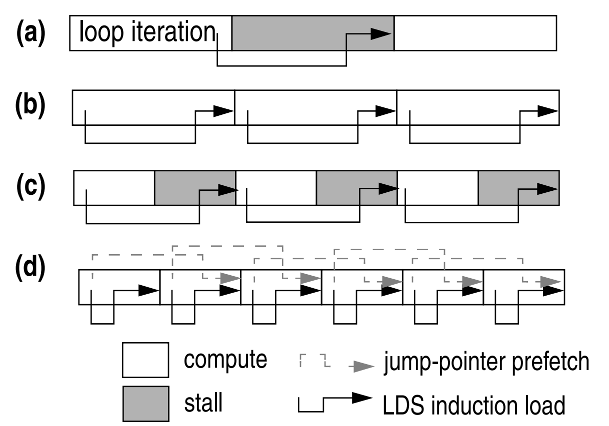

Figure 1. Hiding LDS load latency. *(a) Exposed induction load latency can be hidden by (b) scheduling it early in an iteration. (c) This approach is ineffective if a single iteration has insufficient work. (d) Jump-pointers can leverage the work of multiple iterations.*

We present a method for overlapping LDS load latency with the work of multiple iterations via the structured use of *jump-pointers*. Jump-pointers are used strictly for prefetching. Residing at some or all LDS nodes, they point to nodes that are likely to be accessed in the near future, not ones that are functionally adjacent. As shown in figure 1(d), *jump-pointer prefetching* (JPP) overcomes the serial nature of LDS address generation and obtains the address of an otherwise non-adjacent LDS node via a single low-latency lookup. This in turn allows us to overlap the access latencies of multiple nodes, or equivalently, to overlap the latency of one node with multiple iterations.

Our general framework combines jump-pointer prefetching with chained prefetching, which uses the pointers available in the original unmodified program. We show that jump-pointer prefetching and chained prefetching can be combined in different ways to create *four prefetching idioms* which we call *queue jumping*, *full jumping*, *chain jumping* and *root jumping*. Since both jump-pointer prefetching and chained prefetching can be implemented in either hardware or software, each idiom can be instantiated in one of *three implementations*: *software*, *hardware*, and *cooperative*. The cooperative scheme handles jump-pointer prefetching in software and chained prefetching in hardware.

Each idiom/implementation combination has advantages and drawbacks that make it suitable for certain kinds of LDS traversals. We study a set of pointer intensive benchmarks and attempt to isolate the program features that best guide idiom and implementation selection. Our experiments show that software, cooperative and hardware prefetching eliminate an average of 72%, 83% and 55% of the total memory stall time in these programs, translating into speedups of 15%, 20%, and 22% respectively. This is a significant improvement over other known schemes.

This rest of the paper is organized as follows. The next section presents our JPP framework and a benchmark characterization. The three implementations are described in Section 3 and evaluated in Section 4. The last sections discuss related and future work and our conclusions.

2 Jump-pointer Prefetching Framework

Our prefetching framework can be described in terms of two building blocks: *jump-pointer prefetches* and *chained prefetches*. *Jump-pointers* are pointers added to the program's data structures for prefetching purposes only. We say that a jump-pointer resides in a *home* node and points to a *target* node. *Jump-pointer prefetches* prefetch target nodes using the jump-pointer at the home node. For prefetching to succeed, the target of a jump-pointer must point to a node that is likely to be referenced some time after the corresponding home node. Chained prefetches, on the other hand, do not require jump-pointers, they prefetch using the original pointers in the structure. Each of these types of prefetch provides different benefits and has different associated performance costs. Jump-pointer prefetches can prefetch arbitrary LDS nodes, hide arbitrary amounts of latency and allow otherwise serial prefetches to execute in parallel. However, jump-pointers require storage and maintenance, imposing overheads on the program. Chained prefetches incur no explicit overheads and require no additional maintenance, but provide a more limited amount of latency tolerance.

Jump-pointer prefetches and chained prefetches can, to some degree, be traded off for one another and combined to create efficient prefetching solutions. Our framework comprises four *idioms* that represent points along this trade-off/combination spectrum. On one end, *full jumping* uses jump-pointer prefetches exclusively. At the other, *root jumping* uses few jump-pointer prefetches, and relies heavily on chained prefetching. *Chain jumping* is somewhere in the middle. Finally, *queue jumping* is a special case that handles simple structures using jump-pointer prefetches only. The rest of the section describes these idioms and provides a benchmark characterization in which high level program features are used to guide idiom selection. However, we first provide a short overview of the creation and use of jump-pointers.

2.1 Creating Jump-pointers Using a Queue

When prefetching, the distance (in dynamic nodes traversed) between the home and target nodes of a jump-pointer should be proportional to the target node access latency. For instance, if each node visit contains 10 cycles of work and node access takes 40 cycles, a jump-pointer's home node should be four nodes ahead of its target node. A shorter distance would allow only part of the target access latency to be hidden. On the other hand, using a distance that is too long may cause the prefetched block to be evicted before it can be used.

Although ideal distances may vary from node to node, such information is difficult to gather, express or use. Instead, we choose a fixed *interval I*, usually the maximum (or average) required distance per node, and set all jump-pointers *I* nodes ahead of their targets. This is easily accomplished using a queue of length *I*. On LDS creation, or first traversal, a queue maintains the last I node addresses. As each new node is added (traversed) a jump-pointer is created with the node at the head of the queue as its home and the current node as its target. The current node is then enqueued at the tail of the queue, while the home node at the head is removed.

The running example in this section uses the routine *check_patients_waiting* from the Olden benchmark [15] *health,* a hierarchical health-care system simulator. Every iteration, health visits a tree of hospitals bottom up. *Check_patients_waiting* scans the waiting patient list, possibly removing or adding some patients. The main loop is shown in Figure 2(a); the loads in bold are responsible for a large fraction of the cache misses in the program. Figure 2(b) shows jump-pointer creation using the queue method.

2.2 Four Prefetching Idioms

Jump-pointer prefetching and chained prefetching can be combined in various ways to form different prefetching *idioms*. The first idiom we present, *queue jumping*, is not really a conscious combination of these blocks but rather a degenerate case. Queue jumping is applied to simple "backbone" structures which contain nodes of only one type connected in any regular way, such as a list, tree, or graph. In queue jumping, jump-pointers are added to every structure node using the queue method, and these are used to prefetch the entire structure. The trade-offs we spoke of come into play when we deal with "backbone-and-ribs" structures which contain a primary pointer structure with secondary structures at every primary node. The list used by *check_patients_waiting* is such a structure, with the list nodes forming the "backbone", and the patient records the "ribs". Even in these cases, queue jumping can be used to prefetch only the "backbone".

Full jumping, originally introduced by Luk and Mowry [11] in a programmer-controlled context, prefetches "backbone-and-ribs" structures using only jump-pointer prefetches. Full jumping is shown in Figure 2(b). Each node is augmented with *two* jump-pointers: **j_list** points to the node *I* iterations (hops) ahead, and **j_patient** points to that node's patient record. With an appropriate choice of interval, **prefetch list->j_list** hides the **p = list->patient** load latency and **prefetch list->j_patient** hides the latency of **p->time**. Our convention for prefetch statements follows Luk and Mowry's. That is, **prefetch x** means "prefetch the address that is the value of **x**". In software, this is a load followed by a dependent non-binding prefetch.

Chain jumping applies jump-pointer prefetches to the "backbone" and chained prefetches to the "ribs", reducing jump-pointer overheads. At the same time, chain jumping can tolerate as much latency as a full queue jumping solution by exploiting the fact that a jump-pointer can tolerate *any* amount of latency if set with a suitable choice of interval. In our full jumping example, we maintained jump-pointers for both list node *and* patient record, allowing the prefetches to proceed in parallel. In chain jumping shown in Figure 2(c), we keep just the list node jump-pointer and prefetching the patient record *through* it, halving the number of jump-pointers (not to mention jump-pointer updates). The price for this overhead efficiency is that the two prefetches must now execute in series (**prefetch list->j_list->patient** blocks until **prefetch list->j_list** completes). Again, to fully hide the latency of both loads chain jumping must use a longer interval than full jumping. For instance, suppose each iteration in our example contains 10 cycles of computation, while **list->forward** and **p->time** each take 20 cycles to complete. Full jumping has only 20 cycles of latency to cover and can install jump-pointers at two node intervals. Chain jumping incurs the latencies in series and must use a four-node interval.

Root jumping is most suitable for collections of small, highly dynamic pointer structures. It relies almost exclusively on chained prefetching. All jump-pointers installed in highly dynamic structures, such as the lists processed by *check_patients_waiting,* eventually become invalid. Keeping jump-pointers updated is one way to deal with this problem. However, continuous updates are expensive and updates on insertions/deletions only are complex to implement. *Root jumping* avoids the update problem by prefetching in a way that is transparent to LDS mutation. In root jumping, an entire LDS is prefetched in chained jumping fashion using a single pointer to the root. In Figure 2(d), **&vlg->j_vlg.hosp.waiting** computes the address of the root of the list for the next hospital. As the current list is accessed, the next list is prefetched using the original program pointers. On the negative side, root jumping

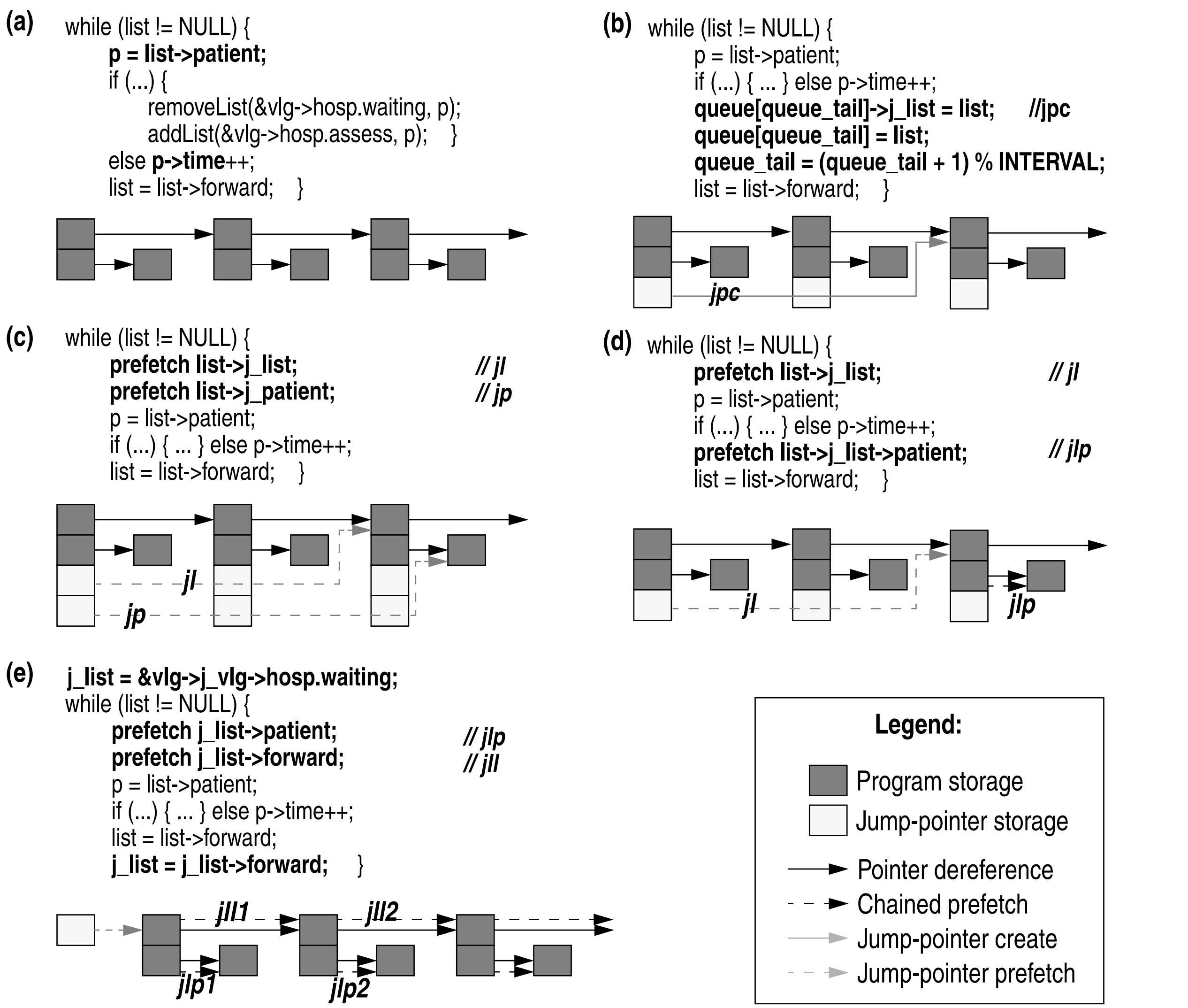

Figure 2. Jump-pointer prefetching idioms. *(a) Unoptimized check_patients_waiting procedure from the health benchmark: the loads in bold traverse a list of patient records and incur many cache misses that combine to serialize the routine. (b) Jump-pointer creation: pointers are installed using the queue method. (c) In full jumping, each list node is fitted with jump-pointers to a future node and its patient record. (d) Chain jumping achieves the same effect without maintaining the second jump-pointer. (e) Root jumping can prefetch an entire list with a single jump-pointer.*

Bench	Parameters	Inst Count	LDS Miss	Miss Overlap	Data Structures/Runtime Behavior	Prefetching Idiom
bh	2K bodies	1788M	1.6%	0.12	static octree rebuilt at each iteration	queue
bisort	250,000 numbers	565M	4.8%	0.26	binary tree nodes flipped	queue
em3d	2000 nodes	60M	21.7%	1.62	static list, pointer array at each node	queue, chain
health	5 levels, 500 iters	162M	23.3%	0.22	static quadtree, dynamic lists at each node	full, chain, root
mst	1024 nodes	199M	13.7%	0.32	dynamic list, static hash table at each node	queue, root
perimeter	4K x 4K image	1570M	8.6%	0.53	static quadtree	queue
power	10,000 nodes	791M	0.4%	0.12	static multiway tree, lists at each node	queue
treeadd	1M nodes	72M	3.4%	0.00	static binary tree,	queue
tsp	100,000 cities	328M	3.9%	0.76	binary tree converted to list	queue
voronoi	60,000 points	317M	0.6%	0.05	static binary tree	queue

Table 1. Olden benchmarks. *The statistical characterization shows the fraction of loads that are both LDS related and miss in a 64KB L1 data cache (LDS miss) and the average degree of L1 miss overlapping. The structural characterization shows types and runtime behaviors of data structures used. We combine information from both analyses to select the appropriate prefetching idiom(s) for each benchmark.*

magnifies chained prefetch serialization effects. Consequently, it is well suited for collections of LDS that are both dynamic *and* short, like hash table buckets.

2.3 Selecting the Appropriate Idiom

With this idiomatic framework in place, we are faced with the task of choosing (explicitly for software implementations, implicitly for hardware) the appropriate idiom for a given program. Some programs may not need a jump-pointer prefetching solution, they may not incur many LDS cache misses or alternatively have sufficient parallel work to overlap with those misses. Others, by virtue of their algorithmic structure, cannot support a JPP implementation. The specific set of programs we will study is the Olden pointer-intensive benchmark suite [15] which has previously been used to study both hardware and software prefetching mechanisms [11, 16]. A summary of the benchmarks is shown in Table 1.

In considering the *need* for jump-pointer prefetching, we measure the fraction of loads that are both LDS related *and* incur full or partial cache misses. Equivalently, this is the product of the overall miss rate and the fraction of misses accounted for by LDS loads. Other kinds of loads (arrays, stack, global) and loads that hit in the cache will not be affected by JPP. To obtain a measure of parallelism, we count the average number of in-flight first level cache misses sampled at cache misses themselves. A low value indicates that few cache misses are being overlapped. Combined with a sizable miss ratio, this implies that LDS misses are serializing the program and points out a significant need for the parallelism enabled by JPP.

Table 1 gives LDS miss fraction and miss overlap numbers for a 64KB, 32B line, 2-way associative data cache and a super-scalar out-of-order processor core as described in Table 2. This preliminary analysis indicates that *power* and *voronoi* may not require a JPP solution as JPP can attack fewer than 1% of the loads in these programs. In addition, *em3d* appears to have sufficient parallelism to overlap a significant number of LDS misses. A serial LDS prefetching mechanism, like dependence based prefetching [16], will probably suffice to handle them.

Regardless of the need for it, JPP is not applicable in every situation. For instance, large structures that are *extremely* dynamic and data dependent traversals (tree searches) are difficult to prefetch even using jump-pointers. The last part of Table 1 details the kinds of data structures used in each benchmark, their runtime behavior, and the jump-pointer idiom(s) we judged to be appropriate for each case.

Bh, *bisort*, *perimeter*, *power*, *treeadd*, *tsp* and *voronoi* all use “backbone-only” structures, making queue jumping the only choice. Actually, we may not want to explicitly implement any idiom on *bisort* and *tsp*, as these programs use structures that are both large and extremely volatile. For these, jump-pointer techniques may be both complex to implement and insufficiently effective to offset the overheads of any software components. Because *em3d* and *health* have “backbone-and ribs” structures, we can use chain and full jumping for these. Finally, *health* and *mst* use dynamic lists that suggest the use of root jumping. With this characterization in mind, we proceed to the discussion and evaluation of our three implementations.

3 Implementations

Both of our JPP building blocks: jump-pointer prefetching and chained prefetching, can be implemented in either hardware or software yielding four possible combinations. We present three: *software-only*, *hardware-only,* and a *cooperative* scheme in which jump-pointer prefetching is done in software and chained prefetching is handled in hardware. Although it is possible for software and hardware to cooperate in reversed roles, this final combination makes little sense in terms of both complexity and performance. So that we can introduce hardware techniques gradually, we describe the three plausible implementations in the following order: software, cooperative, and finally hardware.

3.1 Software

Software JPP implementations require no special hardware support and, if implemented by hand, benefit directly from the programmer’s high-level knowledge of the code. He/

she can choose the appropriate idiom or even construct a special purpose algorithm that exploits high-level program invariants. On the downside, software is restricted to use only architected resources, a constraint that manifests in three major ways. First, jump-pointer storage consumes user memory and increases the program's data footprint. Second, jump-pointer maintenance and prefetching code increases both static program size and dynamic instruction count. Finally, software chained prefetches introduce *serialization artifacts* into the program.

The code examples in Figure 2 are representative of software prefetching implementations for each idiom. In software, jump-pointer creation is simple to implement and is inexpensive in terms of execution time and cache footprint overheads. Jump-pointer creation handles recently referenced nodes and, although it consumes cache bandwidth, rarely causes cache misses. Storage overhead (measured in terms of additional distinct first level cache blocks accessed) is even less of a problem. Although every jump-pointer adds four bytes to the program data set, only the *em3d* full jumping implementation showed *any* memory overhead, and then only a 13% increase in distinct cache blocks accessed. We attribute this phenomenon to the implementation of memory allocators which, for efficiency reasons, allocate small heap objects in only a few fixed sizes. LDS nodes that are not of some preferred size are padded. Jump-pointers can be stored in this would-be padding with no cache footprint increase. Although memory overhead will appear if non-padding allocators are used, it is difficult to estimate the performance impact without an empirical study.

Software implementations of jump-pointer prefetches are also inexpensive: of the two dependent loads required to implement a prefetch, the first is likely to hit in the cache and the second is non-binding, completing on issue. In stark contrast, however, chained prefetches have bad execution characteristics and must be implemented carefully to avoid performance penalties. Since they traverse the pointers of the original program, chained prefetches have the same dependences and dependence chains as the loads for which they are trying to prefetch. Furthermore, these are typically long latency dependence chains since prefetches typically access data that is not in the cache. In software, these long latency chains will clog the out-of-order engine unless chained prefetches are spaced sufficiently far apart. This sort of scheduling is difficult in situations where iterations have little work, an unfortunate problem considering that these are precisely the situations that force us to use jump-pointers in the first place.

We implemented the selected idiom(s) for each benchmark by hand. We first profiled the benchmarks to determine which LDS loads contributed the majority of the cache misses, and traced these back to their source level statements. We chose the appropriate prefetch idiom by studying the program source, then inserted the corresponding code. The human component of the entire process typically took about one hour per benchmark. Only in one case, *mst*, did we exploit knowledge of a program invariant to streamline the jump-pointer creation process. Given the uniformity of jump-pointer creation and prefetching, it seems likely that jump-pointer prefetching can be automated in a compiler. However, the structure resizing and realignment needed to create jump-pointer storage requires guarantees about pointer arithmetic that may be difficult to obtain in a language like C. A more likely place for these implementations is a data structure repository such as the C++ Standard Template Library.

3.2 Cooperative

Cooperative JPP introduces modest hardware support to allow chained prefetching to be implemented in hardware, reducing both the direct (instruction count) and indirect (serialization artifact) costs of software implementations.

The hardware component of cooperative JPP is nothing more than the previously proposed dependence-based prefetching mechanism (DBP) [16]. DBP observes an executing program and dynamically identifies LDS loads and their data dependence relationships, effectively isolating the "kernel" responsible for LDS traversal. To prefetch, we speculatively and aggressively unroll the "kernel" in dataflow fashion, alongside the original program. Data is prefetched when it is accessed by the "kernel". In effect, DBP allows the speculative issue of LDS loads that have yet to be scheduled or even seen by the sequential processing core. The central DBP component is a dependence predictor that represents the data dependences among LDS loads. Completed LDS loads access this predictor to determine which, if any, LDS loads can be speculatively issued as prefetches using the just-loaded value as an input address. Completed (arrived) prefetches are sent back to the predictor to potentially launch other prefetches. In this manner, an entire LDS can be prefetched given only its root address and a description of its traversal kernel. We propose a DBP implementation that contains two optimizations. To minimize resource contention, prefetch requests are queued (PRQ) until data cache ports are idle. To avoid cache pollution, prefetched blocks are installed into a prefetch buffer (PB).

With chained prefetching in hardware, software chained prefetches can be removed from the code, streamlining chain and root jumping implementations. For instance, consider the software root jumping implementation for *health* from Figure 2(e). A cooperative version eliminates the statements **prefetch j_list->patient**, **prefetch j_list->forward**, **j_list = j_list->forward**. Not only does the software version execute more instructions, it potentially serializes the program along the **j_list = j_list->forward** dependence. The cooperative counterpart of this dependence executes in hardware and does not serialize the program.

To make a cooperative implementation work, software prefetches must be made to trigger chained prefetches in the hardware. These chained prefetches correspond to speculative instances of original program LDS loads. One simple way to achieve this communication is to have the dependence predictor learn the relationships between jump-pointer prefetch instructions and other LDS loads. Once these connections are in place, the hardware automatically issues chained prefetch instances of any loads that depend on a jump-pointer prefetch. In addition to eliminating software chained prefetches, this communication mechanism allows the remaining software jump-pointer prefetches to be streamlined. Recall, a software prefetch is implemented using two dependent loads, the second of which is non-binding. By performing the second load in hardware, the corresponding software sequence is reduced to the first load which now can be made non-binding.

3.3 Hardware

Hardware JPP has the advantage that it imposes *no* explicit execution overhead on the program. However, hardware JPP faces challenges in finding jump-pointer storage and may perform poorly when high level program understanding is needed to construct a prefetching solution.

For a hardware-only implementation, we extend the DBP mechanism with structures that direct jump-pointer creation (storage) and prefetching (retrieval). Our particular mechanism implements *chain jumping*: restricting jump-pointer prefetching to recurrent "backbone" loads and using DBP to automatically chain prefetch "rib" loads. This solution automatically provides queue jumping where appropriate. These two idioms are simple to implement in hardware and handle most programs. Full and root jumping are not implemented, due to difficulties with finding jump-pointer storage and a reliance on high level program understanding, respectively. In this section we explain the processes of jump-pointer creation and retrieval.

For jump-pointer creation, we implement the queue method in hardware. Each *static* load identified as being recurrent ("backbone") is associated with a queue that tracks its most recent input addresses. Address queues for the set of active recurrent loads are stored in the Jump Queue Table (JQT). When an instance of a recurrent load commits, it accesses the JQT and creates a jump-pointer from the node sitting at the head of the queue to the node corresponding to its own input address. This process is illustrated in Figure 3(b). **list = list->forward** creates a jump-pointer from the node visited four hops ago, **A**, to the current node, **E**. A request for storing this jump-pointer is generated while the queue is updated to reflect the access of the current node.

Jump-pointer retrieval and prefetch initiation is a more delicate process which we first explain at a high level using the example in Figure 3(c). Whenever an LDS "backbone" load issues, the jump-pointer residing at the corresponding home node is placed (magically for now) into a special non-architected location called the Jump-pointer Register (JPR). A jump-pointer prefetch is created using a speculative instance of the load with the JPR *value* as its input. A completed jump-pointer prefetch may access the predictor and spawn chained prefetches.

The main issue in implementing hardware jump-pointers is not which pointers to create, but rather where they should be stored. Two storage options are available: a non-architected on-chip table and user memory. Non-architected on-chip storage is attractive because of its implementation simplicity. However, its non-scalability is a major problem. Prefetching a 16K-node LDS requires 64KB of jump-pointer storage, with potentially more storage for tags. It may be difficult to justify the construction of a special purpose on-chip predictor of this size. Another serious problem is the volatility of table contents, both when traversing structures with more nodes than table entries and across context switches. Our experiments show that, with the exception of *em3d* which has only 4000 nodes in its "backbone" data structure, most benchmarks experience negligible speedups (less than 2%) from a 16K entry on-chip *jump-pointer cache*.

Although more complicated, storing jump-pointers in *user memory* is more promising. Earlier we observed that software jump-pointers are often stored in what otherwise would be allocator padding. We believe that hardware can and should use this same padding. Padding storage is available in quantities proportional to the number of nodes. It provides a natural, tagless way of attaching jump-pointers to their home nodes and guarantees fast jump-pointer access since the jump-pointer is brought into the cache when its home node is referenced. One concern with this approach is that it creates a different memory image than the one dictated by the program. However, this point is mitigated since the storage in question would not have otherwise been read. Previously, Martin et. al. [12] used this argument to justify cancelling would-be stores of dead memory values. We use a similar argument to justify storing non-program values in would-be unused locations.

Convinced of the advantages of allocator padding storage, we now need a safe and automatic way for detecting and using the padding. The method we present adds instruction set and memory allocator interface extensions to our otherwise *pure* hardware scheme. However, these play no *active* part in either jump-pointer creation or prefetching. We leave other possibilities for future exploration.

Most allocators (e.g., GNU C library) allocate small memory chunks in sizes that are strictly powers of two; we suggest solidifying this convention so that it can be assumed by the compiler. Next, we add four or five load variants to the instruction set (call these lw8, lw16, etc.) and use them to implement recurrent load accesses with the particular variant chosen based on the size of the referenced object. Specifically, if the size is exactly a power of two such that no padding is available, then the unvaried load is used. Otherwise, storage for at least one jump-pointer is available at the end of the allocated block. To annotate its location, we use the load variant corresponding to the object size rounded up to the next power of two. In this way, an annotated recurrent load can be used to compute a jump-pointer address in addition to the standard effective

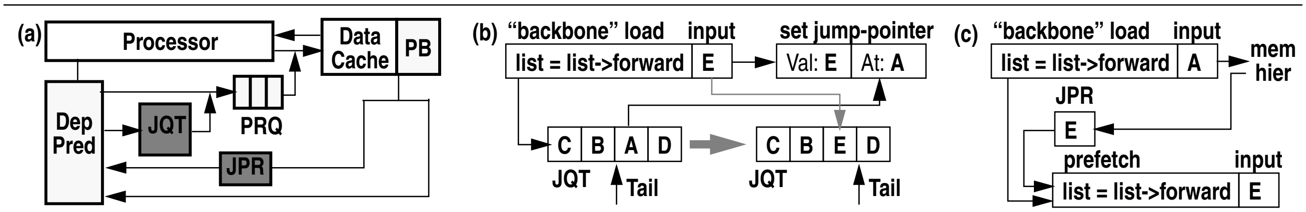

__Figure 3. Hardware JPP.__ (a) Block diagram with DBP specific parts in light gray and JPP components in dark gray. (b) Installing jump-pointers: the Jump Queue Table (JQT) entry contains the previous four input addresses of the load __list = list->forward__. When a new instance commits, it creates a jump-pointer from the node at the queue tail, A, to the current node, E. It then updates the JQT, advancing the queue. (c) Jump-pointer prefetching: As a "backbone" load issues, the jump-pointer in the corresponding home node is placed in the JPR and used to launch a prefetch.

address. This second address can be used to both store a jump-pointer and to fill the JPR with the appropriate word without performing any explicit additional loads.

4 Evaluation

Our performance evaluation proceeds in four steps. We begin with a per benchmark comparison of the JPP idioms. For each benchmark, the best idiom is chosen as the representative software/cooperative solution. With software and cooperative schemes fixed, we quantitatively evaluate all three JPP implementations for each benchmark and compare them to other prefetching schemes. Using a few selected benchmarks and some extrapolated current trends, we project the performance impact of JPP on future architectures. Finally, we attempt to quantify both the direct and implicit costs of JPP implementations.

To perform our experiments, we modified the Olden benchmarks by hand to execute on a single processor, and compiled them for the MIPS-I architecture using the GNU GCC 2.7.2 compiler with flags `-O2 -finline-functions`. Many of the benchmarks contain long allocation-dominated initialization phases that are not accelerated by prefetching; we did not discount these in any way. Our evaluation tool was the SimpleScalar timing simulator [1], with micro-architectural parameters as shown in Table 2. We always report execution time as a decomposition of *memory access time* and *compute time*. We define compute time as execution time assuming uniform single cycle data memory access but with realistic cache bandwidth. Compute time encapsulates stalls resulting from branch mispredictions, instruction cache misses, structural hazards and insufficient memory bandwidth. For each bar, the compute portion was obtained using a second simulation.

4.1 Comparing Idioms

We evaluate the relative merits of the JPP idioms in the context of the software and cooperative implementations. We ignore hardware prefetching for now because it implements only one idiom. Results are shown in Figure 4. With chain and root jumping each implemented in only two benchmarks, we discuss the results on a case basis.

In *em3d*, the loads that would most benefit from prefetching access pointer arrays stored at every node. It is costly to implement jump queues and explicit jump-pointers for arrays only in software and, consequently, full jumping cannot be used. In a cooperative environment, however, implementing these array prefetches is simpler. Consequently, an algorithm that performs only explicit queue jumping in software and the array prefetches to be implemented in the hardware is the most effective method here. *Mst*'s short hash table bucket chains are ideal for a root jumping implementation. Although *health*'s dynamic lists suggest root jumping, the lists are too long for this idiom to be effective, chain jumping is the choice here.

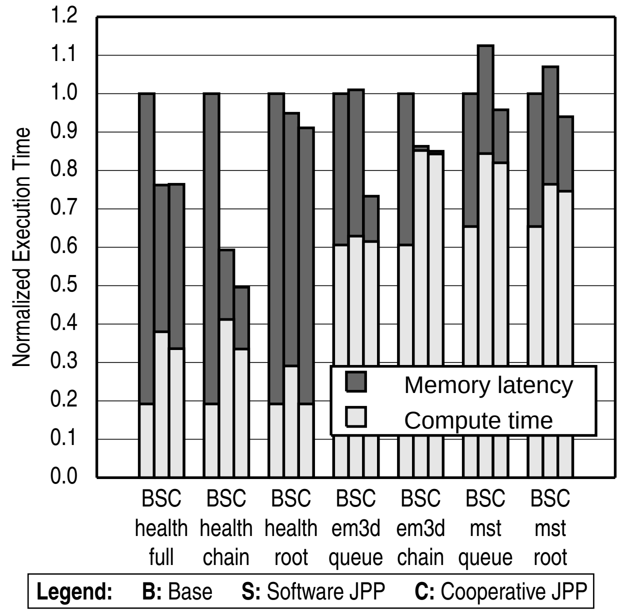

***Figure 4. Comparing idiom performance.** Normalized execution times of software and cooperative implementations of the three prefetching idioms.*

We believe that in general chain jumping, a combination of jump-pointer prefetching for recurrent "backbone" loads and chained prefetching for "rib" loads is the most

Out-of-Order Core	5 stage, 4 way superscalar, out-of-order pipeline with 64 instructions in-flight. Wrong path execution modeled. Loads and stores issue via a 32 entry queue with a 1 cycle load bypass. Loads wait for all previous store addresses before issuing.
Branch Prediction	8K entry combined 10-bit history gshare and 2-bit predictors. 2K entry, 4-way associative BTB.
Memory System	32KB, 32B lines, 2-way associative, 1 cycle access first level instruction cache. 64KB, 32B lines, 2-way associative, 1 cycle access, first level and data cache. A maximum of 8 outstanding data misses. 16-entry ITLB, 32-entry DTLB with 30-cycle hardware miss handling. Shared 512KB, 64B line, 4-way, 12 cycle access second level cache. 70-cycle memory latency. 8B busses to L2 cache and main memory clocked at 1/2 and 1/4 processor frequency, respectively, with cycle level utilization modeled.
Functional Units (latency)	4 int ALU (1), 1 int mult (3), 1 int div (20), 2 FP add (2), 1 FP mult (4), 1 FP div (24), 2 load/store (1)
Software Prefetches (where applicable)	Non-binding, complete on issue, and can initiate TLB miss handling.
Dependence Based Prefetch Mechanism (where applicable)	256 entry, 4-way associative dependence predictor that allows two queries per cycle. Prefetched blocks are stored in a 2KB, 32B line, 8-way associative, 1 cycle access prefetch buffer and subsequently installed into the cache if used. Prefetch requests wait on an 8 entry request queue.
Jump-pointer Mechanism (where applicable)	32 entry, fully associative jump queue table (JQT) with fixed 8-address queues. One Jump-pointer Register (JPR) allowing a single jump-pointer access per cycle.

***Table 2. Simulated Machine Configuration.** Base simulator configuration for all of our experiments.*

effective idiom. In pure software, it can achieve the same effect as full jumping given an appropriate choice of interval and careful scheduling. In a cooperative implementation, it can take advantage of the automatic chained prefetching performed by the dependence hardware. Root jumping can be the most effective idiom in certain specialized cases, in *mst* for instance, but is not a general purpose technique. Chain jumping is also the idiom implemented by the hardware mechanism.

4.2 Comparing Implementations

With the most efficient idioms selected for our software and cooperative implementations, we can now evaluate the hardware scheme alongside them. For added insight, we compare our JPP implementations to DBP, a hardware mechanism that does not use jump-pointers. The results are shown in Figure 5. Again, we discuss specific cases before making general observations.

As our benchmark characterization predicted, both *power* and *voronoi* have very small memory latency execution components. Even the smallest computation overheads introduced by software prefetching overwhelm the potential benefit and produce an overall slowdown. In *voronoi*, software and cooperative prefetching actually *increase* the total memory latency, as useless prefetches contend for memory resources with array based cache misses. Along similar lines, we noted that *bisort* and *tsp* are both highly dynamic structures for which any jump-pointer scheme will not remain valid for long enough to be useful. In fact, explicit jump-pointer prefetching has an adverse effect on *bisort*, as traversal order changes rapidly and any jump-pointer prefetches become purely overhead. Software or cooperative prefetching should not be implemented for these benchmarks. In contrast, while hardware JPP is useless, at the very least it does not degrade performance.

The remaining programs have sizable memory latency components and benefit from software and cooperative JPP implementations. For these programs, in fact, JPP in any form provides superior performance over prefetch mechanisms that do not possess the ability to break address generation serialization constraints. If we disregard *bh*, *bisort*, *power*, *tsp* and *voronoi*, software, cooperative and hardware JPP improved performance by averages of 15%, 20% and 22%, respectively while cutting the memory latency execution components by 72%, 83% and 55%. As we observed earlier, the performance returns on software and cooperative schemes would be even larger if not for their associated computation overhead.

In contrast with JPP, dependence based prefetching provided only an 11% average performance boost while cutting only 29% of the total memory latency component. This is not surprising since our earlier characterization indicated that most benchmarks are serialized by their LDS load chains. Indeed, the one benchmark which has some natural parallelism, *em3d*, benefits almost as much from the non-parallelizing dependence based prefetching as it does from the use of jump-pointers.

The relationships among the different JPP implementations are also interesting. One expected trend is that the cooperative implementation consistently outperforms the pure software one, by as much as 10% on benchmarks that use chain or root jumping like *bh*, *mst*, and *health*. These improvements are due to the elimination of software chained prefetches and their serialization artifacts. More modest improvements, 1 to 2%, are observed for full and queue jumping implementations of *bisort*, *tsp*, *treeadd*, and *perimeter*. These are due to the streamlined implementations of prefetches themselves.

Both software and cooperative implementations are fundamentally more effective than their observed speedups would indicate. Jump-pointer creation imposes non-trivial overhead that degrades from the observed impact. In the *health* chain jumping implementations, for instance, jump-pointer creation creates an *a priori* 12% slowdown that must be overcome by prefetching before any performance gains are obtained. If we correct for this initial degrada-

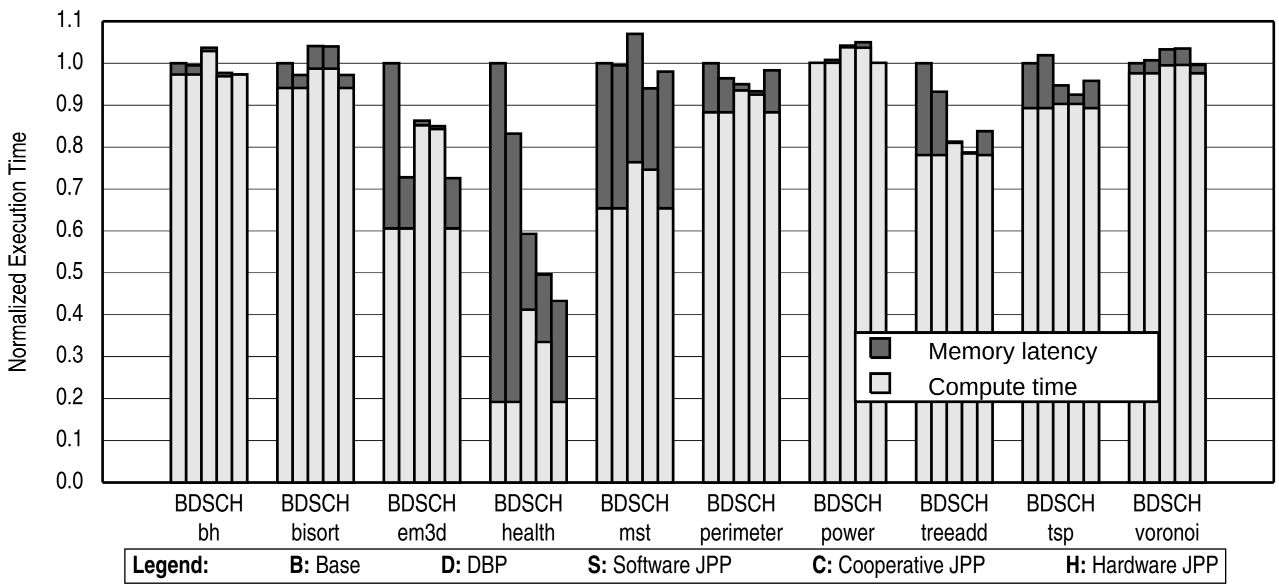

Figure 5. Comparing prefetching implementations. *Execution times (normalized to an unoptimized execution) for three JPP implementations: software, cooperative and hardware, and for dependence based prefetching.*

tion, we find that software JPP improves performance by 90%, not 68%, for this program and cooperative JPP achieves 130% rather than 103% speedups. These are the gains we would expect to see in a hardware only implementation, where jump-pointer creation is "free".

In reality, however, the relationship between the effectiveness of hardware JPP and its software and cooperative counterparts is variable. While hardware is more effective on *em3d* and *health*, the opposite holds for *mst*, *perimeter,* and to a lesser degree *treeadd.* The feature that distinguishes the first set of programs from the second is the number of traversals performed on the data set. *Em3d* and *health* perform 100 and 500 traversals, respectively, while *treeadd* makes four passes, and *mst* and *perimeter* make one each. Hardware JPP takes one full traversal to install jump-pointers, and so optimizes only second and subsequent passes over the data. The choice between cooperative and hardware implementations, between incurring explicit jump-pointer creation overhead and leaving the first pass unoptimized clearly depends on the total number of traversals in the program. In *health* and *em3d*, one unoptimized pass is negligible, on *treeadd* it forfeits one quarter of the total savings, while for single pass programs like *perimeter* and *mst*, it makes hardware JPP useless. To prefetch one-pass programs, jump-pointers must be installed as the LDS itself is built. For reasons including difficulties with dependence detection and potential mismatches between creation and traversal orders, this is a task seemingly more suited for software.

4.3 Comparing Bandwidth Requirements

Bandwidth consumption is another metric used to evaluate prefetching solutions. Ideally, prefetching should not change the overall number of bytes moved between the first and second level caches and memory. A mechanism that achieves this goal is perfectly efficient since it simply converts fetches to prefetches. However, most mechanisms prefetch some amount of useless data while unnecessarily evicting useful blocks. Figure 6 shows, for each program and each prefetching implementation, the number of bytes moved between the first and second level data caches per dynamic instruction in the original program. We do not count the instructions added to software and cooperative implementations as these would bias our results in their favor.

Some trends are evident from these results. First, jump-pointer prefetching solutions have only a slight impact on bandwidth consumption, increasing the number of bytes moved by 3%, 6% and 35% for software, cooperative, and hardware implementations respectively. This compares favorably with the 25% overheads incurred by dependence based prefetching. Among the three JPP implementations, it is clear that increasing software control over what is prefetched reduces prefetch waste. The additional bandwidth consumed by hardware and cooperative schemes is due largely to the dependence based prefetching mechanism, which prefetches "rib" structures in greedy fashion.

4.4 Tolerating Longer Latencies

Jump-pointer prefetching provides good performance gains on configurations typical of today's systems. However, architectures of the near future will have different characteristics. In this section, we extrapolate current trends to predict the performance of our benchmarks and to explore the importance of JPP in future designs.

We considered projections of several current trends including: wider issue pipelines, deeper pipelines, greater memory bandwidth at all levels, and relatively longer main memory latencies. We chose to disregard trends that project larger on-chip second level caches with the argument that the data sets for these benchmarks will increase as well. Our experiments show that pipeline enhancements and increased bandwidth do not have a significant impact on these benchmarks which, as a group, are highly serial. A relative increase in memory latency, however, translates directly into performance loss. In *health*, for instance, a 4-fold increase in memory latency produces a 2.5-fold increase in execution time. The bars in Figure 7 show normalized execution times for *health*, with different memory latencies (70 and 280 processor cycles) and varying jump-pointer prefetch intervals (8 and 16 nodes).

For benchmarks that lack parallelism, longer memory latencies reduce the effectiveness of serial prefetching schemes like dependence based prefetching, which compress but cannot flatten the memory dependence graph. For *health*, the impact of DBP drops from 17% in the low latency case to 9%. On the other hand, JPP remains effective as relative memory latencies grow. The relative impact of cooperative JPP grows from 50% to 65%, with similar trends observed for hardware and software JPP.

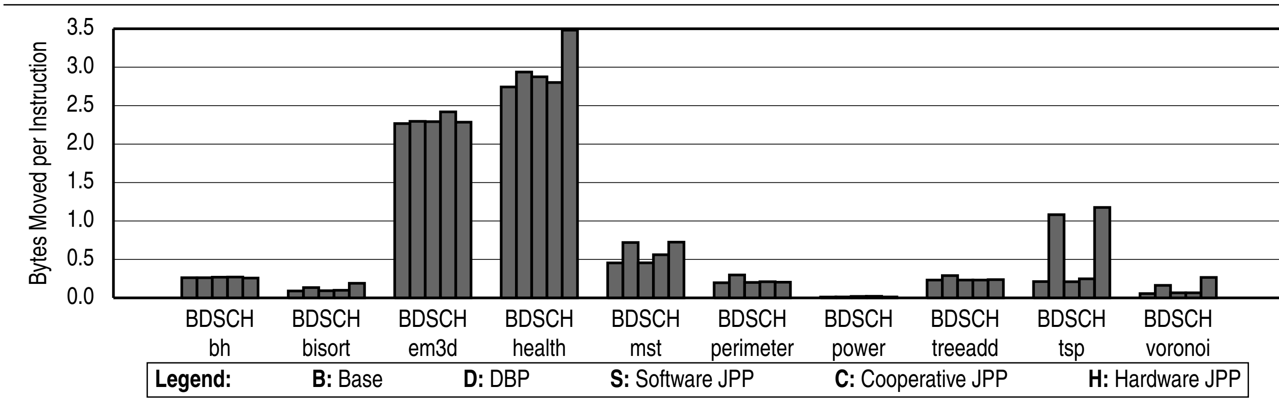

Figure 6. Comparing bandwidth requirements. *Bytes of data moved between the first and second level data caches divided by the number of instructions in the original programs.*

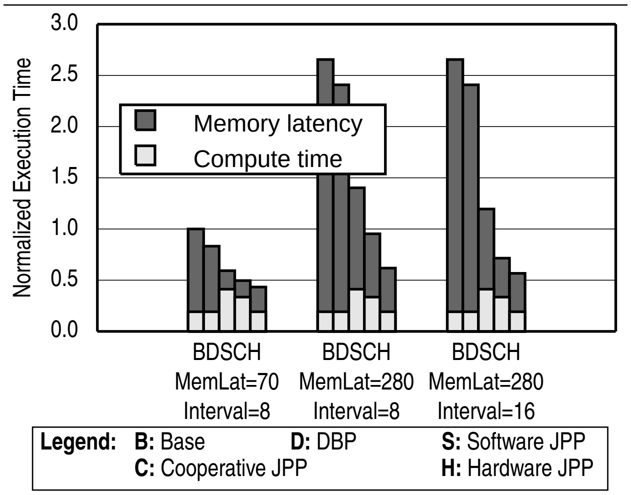

Figure 7. Tolerating longer memory latencies. *Execution times for health: the first group of bars uses the base configuration (70 cycle memory latency), the second and third simulate long memory latency (280 cycles). In terms of prefetching, the first two configurations use a jump interval (the distance between a jump-pointer's home and target nodes) of 8, the third uses an interval of 16.*

Not only does JPP retain its effectiveness, the amount of latency it hides can actually be *tuned* using the jump-pointer interval parameter. Increasing the interval requires more than changing a few lines of software, or increasing the size of the queues in the JQT. The prefetch buffer must be expanded to accommodate the potentially longer residence of prefetched blocks, and the number of possible outstanding prefetch requests must be increased. However, with these modifications, JPP can be used to tolerate even long latencies.

4.5 Other Costs

Among the three implementations, it appears that cooperative and hardware JPP are more effective than software solutions, with the most effective of these two depending on the particular benchmark and prefetch idiom used. Cooperative JPP has the advantage for single-traversal programs, since a hardware-only solution does not accelerate the first pass over a data structure. Cooperative JPP also outperforms hardware in programs suited for root jumping since this idiom typically requires knowledge of high-level program invariants. Hardware JPP has the edge in prefetching multiple-pass programs that lend themselves to queue or chain jumping. Cooperative JPP also has slightly lower effect on bandwidth consumption.

Performance is not the only criterion by which to evaluate JPP implementations. Other important factors are software and hardware component complexities and costs, and any requirements such as ISA changes that may be met with resistance. While software JPP loses the performance battle, it has the advantage of requiring only a programming investment. Hardware-only solutions free the programmer from implementation details but require special processor extensions. Although the implementation we present requires ISA changes and delivers performance improvements only for programs that are recompiled, we believe that these requirements can be avoided at the cost of additional hardware complexity. Cooperative JPP seems to have an overall advantage when we consider performance and cost together. It provides performance that compares favorably with hardware JPP, while requiring fewer processor resources and less significant interface changes. While cooperative JPP does require some programmer effort, it eliminates the most tedious portion of the software requirement: scheduling chained prefetches. For now, cooperative JPP seems to be the best choice for combining high performance with low implementation cost. That may change as some of the challenges associated with hardware implementations are overcome.

5 Related Work

While prefetching literature is abundant, prefetching directed at the special requirements and challenges of LDS is less extensive. Early work on improving the spatial locality in LDS reference streams was done in the context of LISP machines [5, 7]. This work aimed to increase page reference density and used runtime techniques implemented in either the memory allocator or garbage collector. Recently, Seidl and Zorn [17] and Calder et. al. [2] have shifted focus to cache-conscious allocation and added profile feedback to this process. Chilimbi et. al. implemented cache specific techniques such as compression and line coloring for LDS nodes in a memory allocator [3] and a generational garbage collector [4].

One of the earliest software-controlled LDS prefetching scheme was SPAID of Lipasti et. al. [10] which heuristically dereferenced pointers passed into procedures. Luk and Mowry [11] discussed several software techniques, including compiler-based greedy prefetching, programmer-controlled history pointer prefetching (essentially software full jumping), and data linearization. On the hardware side, Mehrotra and Harrison [13, 8] introduced a detection and prefetch scheme for loads that, in isolation, exhibited one of a number of preset access patterns, self recurrence being one. Most recently, Roth et. al. [16] described a dependence based mechanism that dynamically isolates the LDS access kernel in a program and prefetches by speculatively pre-executing that kernel.

Pugh introduced skiplists [14], a jump-pointer-based sorted list implementation with search and manipulation statistics similar to those of a balanced search tree. Jump-pointers have been used to represent set data structures efficiently [6] and to parallelize searches and reductions on lists [9]. Discussions of maintaining recursion-avoiding traversal *threads* in non-linear data structures can be found in data structures literature [18]. As noted earlier, Luk and Mowry [11] suggested the use of programmer controlled jump-pointers for prefetching. We are not aware of any implementations, actual or proposed, of hardware or cooperative jump-pointer prefetching.

6 Summary and Future Directions

In this paper, we describe the general technique of jump-pointer prefetching (JPP) for tolerating linked structure (LDS) access latency. JPP is effective when limited work is available between successive dependent accesses (e.g., a tight pointer chasing loop) to enable aggressive scheduling techniques to prefetch effectively. We present, evaluate, and compare three JPP implementations. Our technical contributions are summarized as follows:

- We present JPP as a general purpose technique for tolerating serialized latencies that result from LDS traversal. By storing explicit jump-pointers to nodes several hops away, JPP overcomes the pointer-chasing problem. It is able to generate prefetch addresses directly, rather than in a serial fashion, and is effective even in situations where not enough work is available to hide latencies by scheduling.
- We present two basic prefetching techniques: jump-pointer prefetching and chained prefetching, which can be combined to form four prefetching idioms: *queue*, *full*, *chain* and *root* jumping. Drawing from their component prefetching blocks, each idiom has certain advantages and disadvantages. We provide a high-level program characterization that can be used to select a suitable idiom for a given program.
- We describe three JPP implementations: software-only, hardware-only, and cooperative. For those programs with appreciable memory latency components, these implementations reduce overall observed memory latency by 72%, 55%, and 83%, respectively and achieve speedups of 15%, 22%, and 20%.

Several directions for future work exist, beginning with a systematic study of the design space of hardware JPP. Our simulated implementation used a fixed queueing interval of 8 nodes without regard to the trade-offs in latency tolerance and predictive accuracy. A more detailed study of this spectrum is needed, with a better mechanism adapting the interval on a case by case basis. We also assumed one method for detecting and exploiting the unused memory that pads allocated blocks. Other methods for detecting and using this padding, or maybe some other unused part of memory, may be better. Our rationale for using memory to store jump-pointers was predicated on the sheer number of pointers that would be needed. Advances in compression or prediction could make processor storage a viable option. Finally, jump-pointer prefetching may be generalized to other classes of data structures with serialized access idioms, like sparse matrices and database trees.

Acknowledgements

The authors thank Milo Martin, Marci McCoy, and Craig Zilles for their comments on several drafts of this paper, and the anonymous referees for their suggestions. This work was supported in part by NSF grant MIP-9505853, by U.S. Army Intelligence Center and Fort Huachuca under contract DABT63-95-C-0127 and ARPA order D346, and by an equipment donation from Intel. Amir Roth is also supported by a Cooperative Graduate Fellowship from IBM. The views and conclusions presented are those of the authors and do not necessarily represent the official policies or endorsements, either expressed or implied, of the U.S. Army Intelligence Center and Fort Huachuca or the U.S. Government.

References

[1] D.C. Burger and T.M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, Jun. 1997.

[2] B. Calder, C. Krintz, S. John, and T.M. Austin. Cache Concious Data Placement. In *Proc. 8th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, Oct. 1998.

[3] T.M. Chilimbi, M.D. Hill, and J.R. Larus. Cache-Conscious Structure Layout. In *Proc. SIGPLAN'99 Conference on Programming Language Design and Implementation*, May 1999.

[4] T.M. Chilimbi and J.R. Larus. Using Generational Garbage Collection to Implement Cache Concious Data Placement. In *Proc. International Symposium on Memory Management*, Oct. 1998.

[5] C.J.Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[6] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*, chapter 22: Data Structures for Disjoint Sets. The MIT Press, 1990.

[7] R. Fenichel and J. Yochelson. A LISP garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, 1969.

[8] W.L. Harrison and S. Mehrotra. Prefetch system applicable to complex memory access schemes. US Patent 5694568, Dec. 1997.

[9] W.D. Hillis and G.L. Steele. Data Parallel Algorithms. *Communications of the ACM*, 29(12), Dec. 1986.

[10] M.H. Lipasti, W.J. Schmidt, S.R. Kunkel, and R.R. Roediger. SPAID: Software Prefetching in Pointer and Call Intensive Environments. In *Proc. 28th International Symposium on Microarchitecture*, pages 231–236, Nov. 1995.

[11] C-K. Luk and T.C. Mowry. Compiler Based Prefetching for Recursive Data Structures. In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Oct. 1996.

[12] M.M. Martin, A. Roth, and C.N. Fischer. Exploiting Dead Value Information. In *Proc. 30th International Symposium on Microarchitecture*, pages 125–135, Dec. 1997.

[13] S. Mehrotra and L. Harrison. Examination of a Memory Access Classification Scheme for Pointer-Intensive and Numeric Program. In *Proc. 10th International Conference on Supercomputing*, pages 133–139, May 1996.

[14] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668, Jun. 1990.

[15] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting Dynamic Data Structures on Distributed Memory Machines. *ACM Transactions on Programming Languages and Systems*, Mar. 1995.

[16] A. Roth, A. Moshovos, and G.S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proc. 8th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, Oct. 1998.

[17] M.L. Seidl and B.G. Zorn. Segragating Heap Objects by Reference Behavior and Lifetime. In *Proc. 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, Oct. 1998.

[18] T.A. Standish. *Data Structure Techniques*. Addison Wesley, 1980.