Branch-directed and Pointer-based Data Cache Prefetching

Yue Liu
Mona Dimitri
David R. Kaeli
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA

Abstract

The design of the on-chip cache memory and branch prediction logic has become an integral part of a microprocessor implementation. Branch predictors reduce the effects of control hazards on pipeline performance. Branch prediction implementations have been proposed which eliminate a majority of the pipeline stalls associated with branches. Caches are commonly used to reduce the performance gap between microprocessor and memory technology. Caches can only provide this benefit if the requested address is in the cache when requested. To increase the probability that addresses are present when requested, prefetching can be employed. Prefetching attempts to prime the cache with instructions and data which will be accessed in the near future.

The work presented here describes two new prefetching algorithms. The first ties data cache prefetching to branches in the instruction stream. History of the data stream is incorporated into a Branch Target Buffer (BTB). Since branch behavior determines program control flow, data access patterns are also dependent upon this behavior. Results indicate that combining this strategy with tagged prefetching can significantly improve cache hit ratios. While improving cache hit rates is important, the resulting memory bus traffic introduced can be an issue. Our prefetching technique can also significantly reduce the overall memory bus traffic.

The second prefetching scheme dynamically identifies pointer-based data references in the data stream, and records these references in order to successfully prime the cache with the next element in the linked data structure in advance of its access. Results indicate that using this form of prefetching, coupled with tagged prefetching, can reduce cache misses for programs containing pointer linked data structures.

keywords: *data caches, hardware and software prefetching, history-based branch prediction, stride prefetching, pointer-based prefetching*

1 Introduction

As microprocessor speeds continue to outpace memory technology speeds, the design of the cache becomes increasingly important. Even when properly designed, caches can not always contain all of the addresses requested by the microprocessor. As a result, cache misses will occur.

Hill divides cache misses into three classes: *conflict, capacity* and *compulsory* [1]. Conflict misses occur when multiple memory lines map to a single cache line. Many conflict misses can be avoided by utilizing set-associative caches, or by carefully remapping a program onto the available memory space [2]. A number of schemes attempt to reduce the number of cache conflicts while still maintaining low-orders of associativity [3, 4, 5].

Capacity misses occur because the cache is not large enough to hold the entire program. Thus, replacements occur and misses result when replaced lines are re-referenced. Compulsory or cold-start misses are those occurring in all cache designs (in the absence of prefetching). They occur because a memory line is being referenced for the first time. Compulsory misses are generally addressed by adjusting the line size, though increasing the line size may introduce more capacity misses.

Prefetching has been shown to reduce the frequency of compulsory and capacity misses, and can work in parallel with instruction execution. Prefetching hides (part of) the memory latency by exploiting the overlap of processor computations with memory accesses. Since the characteristics of access patterns for instructions and data are typically different, a split-cache organization which decouples data access from instruction access has been used. In order to take advantage of the particular characteristics of the access patterns and design prefetching schemes accordingly, prefetching research is divided into two areas: 1) instruction prefetching, and data prefetching.

Instruction prefetching speculatively loads instruction sequences into an instruction cache or instruction window. One form of instruction prefetching which has attracted a lot of attention recently is dynamic branch prediction. These mechanisms try to predict the outcome of branches, prior to their issue. Branches, which constitute a relatively small percentage of all executed instructions (10-20%), are responsible for a majority of the delays in a microprocessor pipeline [6]. Various branch prediction and target instruction prefetching schemes are proposed in the literature, such as *branch correlation* [7] and Yeh and Patt's *two-level table* [8, 9] schemes.

While much research has focused on instruction prefetching, much of the remaining delay experienced in the pipeline is due to data cache accesses [10]. In particular, data fetches resulting in cache misses were found to be the dominating cause of these delays [11].

Both hardware and software data cache prefetching techniques have been proposed [12]. Hardware-based data cache prefetching schemes include: always prefetching, stride prefetching, tagged prefetching [13]. Among them, tagged prefetching is most effective for prefetching address streams which exhibit spatial locality. However, data references have lower spatial locality than instruction references. This makes it difficult to predict future data reference patterns.

Branch directed prefetching attempts to incorporate data prefetching with instruction prefetching techniques. It associates data references with branch instructions. When a branch instruction is found, the branch prediction mechanism will predict which path the branch will take and the data references at the target location of the predicted path are prefetched accordingly. The obvious advantage of branch directed prefetching is that because branch behavior can be predicted very accurately, data references should also follow a predictable pattern. Still, there remain a significant number of data references which do not maintain a fixed or strided address pattern.

Pointer-based prefetching attempts to identify when accesses are being made to dynamically (i.e., heap) allocated data structures (e.g., linked lists). By recording the access patterns to these structures, we can effectively prefetch the next element prior to its impending access.

This paper is organized as follows. In Section 2 we briefly review past work on data cache prefetching. In Section 3 we present our branch directed and pointer based prefetching schemes. In Section 4, trace-driven results are presented for each scheme. In Section 5 we discuss a number of issues related to this work and in Section 6 we summarize the contributions of this paper.

2 Data Cache Prefetching

Next we review a number of prefetching strategies which have been proposed. *Always prefetch* takes advantage of the *spatial locality* commonly found in computer programs [14]. That is, when a processor accesses a cache line, it will likely access the next sequential line soon thereafter; therefore, prefetching the next sequential line into the cache will likely reduce the number of cache misses. Using always prefetch, on every reference to a cache line, the next sequential line is prefetched [13]. A cache access is performed on every memory access. Since data prefetching may interfere with demand data cache accessing (e.g., demand cache accesses may be delayed due to a prefetch request), special design considerations must be made to limit this interference.

The *prefetch on miss* technique considers issuing a prefetch only if the current reference results in a cache miss [13]. *Tagged prefetching* prefetches the next sequential line when either one of the following occurs:

1) the current reference results in a cache miss, or

2) the current reference, say *@Y*, accesses a cache line which was prefetched into the cache and the reference *@Y* is the first reference to this prefetched line [15].

The idea is very similar to prefetch on miss except that a reference to a cache line which did not produce a miss because it was recently prefetched also initiates a prefetch (i.e., had there not been a prefetch, there would have been a miss to this line).

Unlike always prefetch, prefetch on miss and tagged prefetch only initiate prefetches on specific events (that do not occur on every memory access in typical programs). Thus they require fewer memory cycles than always prefetch. Smith compared always prefetch, prefetch on miss and tagged prefetch [13]. His results showed that always prefetching cut the (demand) miss ratio by 50 to 90 % for most cache sizes, and that tagged prefetching was almost equally effective. Prefetching only on miss is less than half as effective (in terms of cache misses) as always prefetching or tagged prefetch in reducing the miss ratio.

In addition to sequential prefetching, some hardware-controlled data prefetching techniques also allow for predicting data accesses in vector code. Stride prefetching tries to capture the stride of data references. Fu and Patel [16] describe a prefetching method which uses a Stride Prediction Table (SPT) to calculate the stride distance for data references. The SPT is a cache of data addresses previously referenced. Each entry of the SPT consists of an instruction address, a data address, and a valid bit. The instruction address acts as a tag for the SPT entry. The valid bit indicates whether or not this SPT entry is valid. The stored address is the memory address of the data previously referenced by this instruction.

When a memory accessing instruction is issued, the SPT is searched to find the previously referenced data address. If such a data address is found, the stride of the data references for this instruction is then calculated by subtracting this previously referenced data address from the data address referenced by the current instruction. Finally, a data prefetch, whose address is the sum of the currently referenced data address and the stride distance, is issued.

Chen and Baer [17, 18] proposed a number of methods to efficiently prefetch the data stream. In their *Reference Prediction Table* (RPT) they associate data cache prefetches with memory accessing instructions. The idea is to connect certain data references with memory referencing instructions. In order to prefetch data as early as possible, the RPT is accessed with the Look-ahead Program Counter (LA-PC) instead of the regular program counter. The LA-PC stays ahead of the regular program counter

with the help of branch prediction. That is, when a branch instruction is encountered, the LA-PC is set based on a dynamic branch prediction; otherwise, LA-PC is simply incremented.

In a modification to the RPT with LA-PC, Chen and Baer proposed to use Correlated Branch Prediction (CBR) to capture looping-ending conditional branches, and use CBR to drive the RPT. While they use a similar approach to the work presented here, we are concerned with prefetching for all data accesses, not just those contained within loops.

Joseph and Grunwald investigate *Markov Predictors* to drive prefetching [19]. This scheme prefetches between on-chip and off-chip caches, based on the past miss stream pattern. This is a form of correlation-based prefetching, similar to the techniques which we propose in this work (though different in that we pair branches with prefetch requests).

Charney and Reeves describe a prefetching mechanism which uses parent-child key pairs to initiate prefetching [20]. They also incorporate stride prefetching into their form of correlation-based prefetching, much in the same way we use stride prefetching in concert with our branch directed prefetching approach.

3 Branch Directed Prefetching

Branch directed prefetching was first presented by Chang et al [21] and again by Liu et al [22]. This algorithm associates data prefetching with branch instructions. Data prefetching is considered when a branch instruction is fetched from the memory. We have extended past work in this paper to include how to effectively combine tagged prefetching with branch-directed prefetching, as well how to chase pointer-based data structures.

Since branch instructions determine which instruction path is followed, data access patterns are also dependent upon branch instructions. For example, the branch instruction at the beginning of an `If-Then-Else` code segment determines the execution of the `Then` body or the `Else` body. Thus, if there exist data accessing instructions within both the `Then` body and the `Else` body, the branch instruction will determine which set of memory accesses will be executed. By associating data addresses with branch instructions, data prefetches can be issued well in advance of when the data will be needed. Furthermore, data prefetches can be issued independent of when the branch is executed (branch prediction is performed many basic blocks ahead of instruction issue). With a highly accurate branch predictor, this prefetching algorithm will prefetch useful data.

Some advantages of branch directed prefetching are:

1) No extra table is needed since all the information about the prefetched data is stored in the Branch Target Buffer (BTB). It is possible to map several prefetch data addresses to a single entry in the BTB.

2) With branch directed prefetching scheme, data access patterns are associated with unique branches or unique branch targets. Since the number of branches or branch targets for an application is typically less than the number of LOAD instructions (see Table 3 later in this paper), the BTB can have fewer unique entries than an RPT. This should save some tag area.

3) Branch directed prefetching can consider both conditional and unconditional branches.

3.1 Using a BTB for Branch Directed Prefetching

Many hardware-controlled branch predictors use past branch behavior (i.e., branch history) to predict future behavior. These branch execution histories are stored in a BTB. One possible organization of these

buffers is shown in Figure 1. Each BTB entry contains an instruction address and a history field. The branch instruction address is used as a tag for the BTB. The history field is a small state machine which stores the execution history of the branch instruction. For example, Smith's hardware predictor [23], which has been implemented in several contemporary processors, has a two bit counter in each history field to record the past history for each branch instruction. When the branch completes execution the actual outcome is used to update the counter and branch target. A more elaborate branch predictor could be used here [8, 9]. We choose not to, so that we can devote our attention to data prefetching instead of describing elaborate branch prediction mechanisms. A more accurate branch predictor (e.g., gshare, PAS, GAS) should further benefit branch directed prefetching.

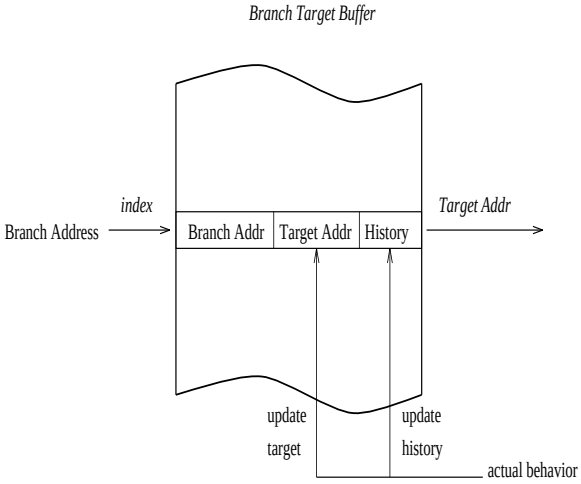

Figure 1: A typical organization of a Branch Target Buffer (BTB)

To implement branch directed prefetching, data buffer fields are added to each entry in the BTB, as shown in Figure 2. The *D-buffer* fields contain several data entries. Each data entry consists of three items:

1) data address

2) stride

3) stride state

The *data address* field stores the data address of the operand accesses associated with the instructions located at the branch target stream. The *stride* field stores the stride between the previous and the current data addresses associated with the same data access located at the branch target stream. The 2-bit *stride state* field provides information indicating whether or not stride prefetching should be invoked.

A 2-bit counter produces an 85% correct branch prediction rate for the benchmark programs we have studied (this does not include unconditional branches and jumps). The counter is always updated according to the branch's actual behavior.

Our objective is to prefetch only useful data into the data cache. To do this, we use the finite state machine (FSM) shown in Figure 3 to decide whether stride prefetching should be used. This FSM uses the information stored in the 2-bit *stride state* field of the BTB. The FSM is initialized to zero. The value is incremented when the stride is the same as that stored in the *stride* field and reset to zero otherwise.

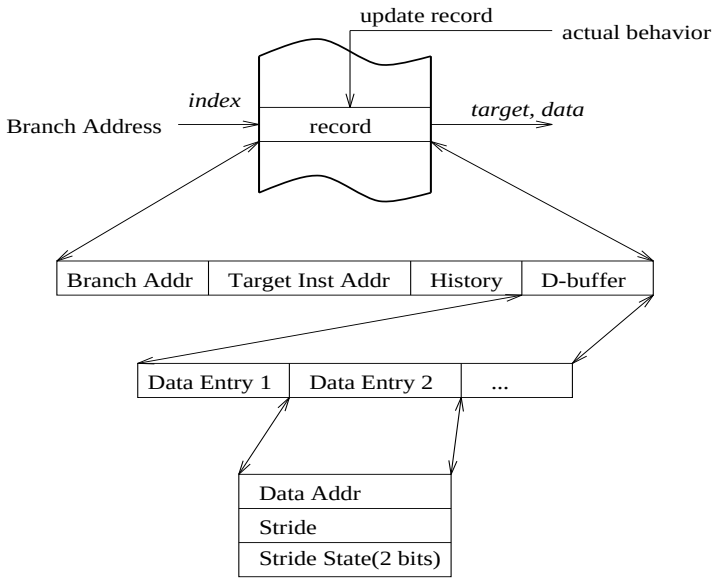

Figure 2: The organization of one entry in the BTB modified for branch directed prefetching

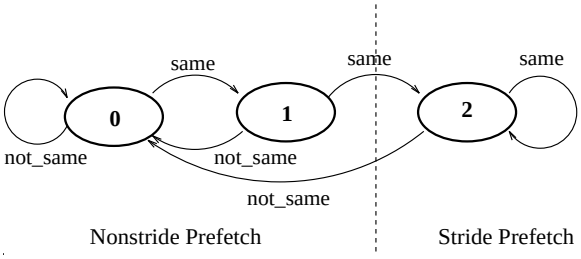

Figure 3: FSM of 2 bit stride state counter

state	decision
0	initial state, non-stride prefetching
1	intermediate state, non-stride prefetching
2	final state, stride prefetching

Table 1: Prefetching issued of state state counter

This FSM favors data references which have the same stride. A stride prefetch is issued when the value is 2, which means that the same stride was used for the last two executions of that memory reference. Table 1 summarizes the three possible states.

Our BTB design has enough information for efficient branch directed data cache prefetching, and will be making decisions to prefetch the LOAD addresses far in advance of the issuance of the LOADs. If a branch instruction has an entry in the BTB, the history bits are first checked to find out how to predict this branch. The stride state is checked to see whether there needs to be a stride prefetch issued. If a vector stride is detected, then the data address to prefetch is calculated by adding the stride distance to the data address in the data buffer; otherwise the data address in the data buffer of the BTB is prefetched directly into the data cache. An example is shown in Figure 4. The branch shown in the example exhibits predictable behavior, that is, it is always taken, always goes to the same target and has a regular data reference pattern in its target stream.

In order to prefetch useful data into the data cache, information stored in the BTB is dynamically updated each time the same branch instruction is accessed. The data address in the BTB is always replaced by the most recently accessed data address. This will require the BTB to be updated on every LOAD instruction, though multiple LOADs could be folded into a single update to the BTB.

Before updating the data stride and the stride state counter, the difference between the currently accessed data address and the data address stored in the corresponding data entry field in the BTB is calculated. If the result is the same as the value of the stride in the BTB, the stride state counter is incremented; otherwise, the data stride is overwritten by the new value and the stride state counter is reset to zero. We have also studied whether data references on the not-taken path should be stored in the BTB. We will address this question in our simulation results.

It is noteworthy that the above BTB design is vulnerable to moving target branches [24]. If a branch's target changes, the data cached in the BTB for this branch is flushed out. Figure 5 shows an example of a changing target branch. Note that the branch in the figure is taken three times but points to three different targets. Since the branch is taken twice (this increases the branch predictor from 0 to 2), a wrong prefetch will be issued each time the branch is predicted. By adding a CALL/RETURN stack [24] to the BTB, this problem is remedied. We model a CALL/RETURN stack in this work.

3.2 Pointer-based Prefetching

Pointer-based prefetching is designed to handle data accesses to heap allocated, pointer linked, data structures (e.g., linked lists, trees). The goal is to successfully prefetch the next element in a linked structure, in advance of the request for that element (the prefetch distance can be adjusted by prefetching multiple links ahead of the current reference).

List based data structures are quite prevalent in integer code, and are commonly used to maintain lists of records when the exact size (i.e., number of elements) of the list is not known statically (i.e., at compile time). The consecutive elements of the list are commonly found at nonsequential memory addresses. This poses problems to most spatially-based prefetching techniques.

Mehrotra describes a data prefetch device called the *Indirect Reference Buffer* (IRB) which detects pointer chasing within the pipeline, and then communicates this to the prefetch unit [26]. While this design will identify a variety of different types of pointer chasing, it does not address how to get far enough ahead of the pipeline to hide the long latencies associated with accessing the upper levels of the memory hierarchy.

Luk and Mowry evaluate a number of compiler-based solutions to this problem [25]. Their work

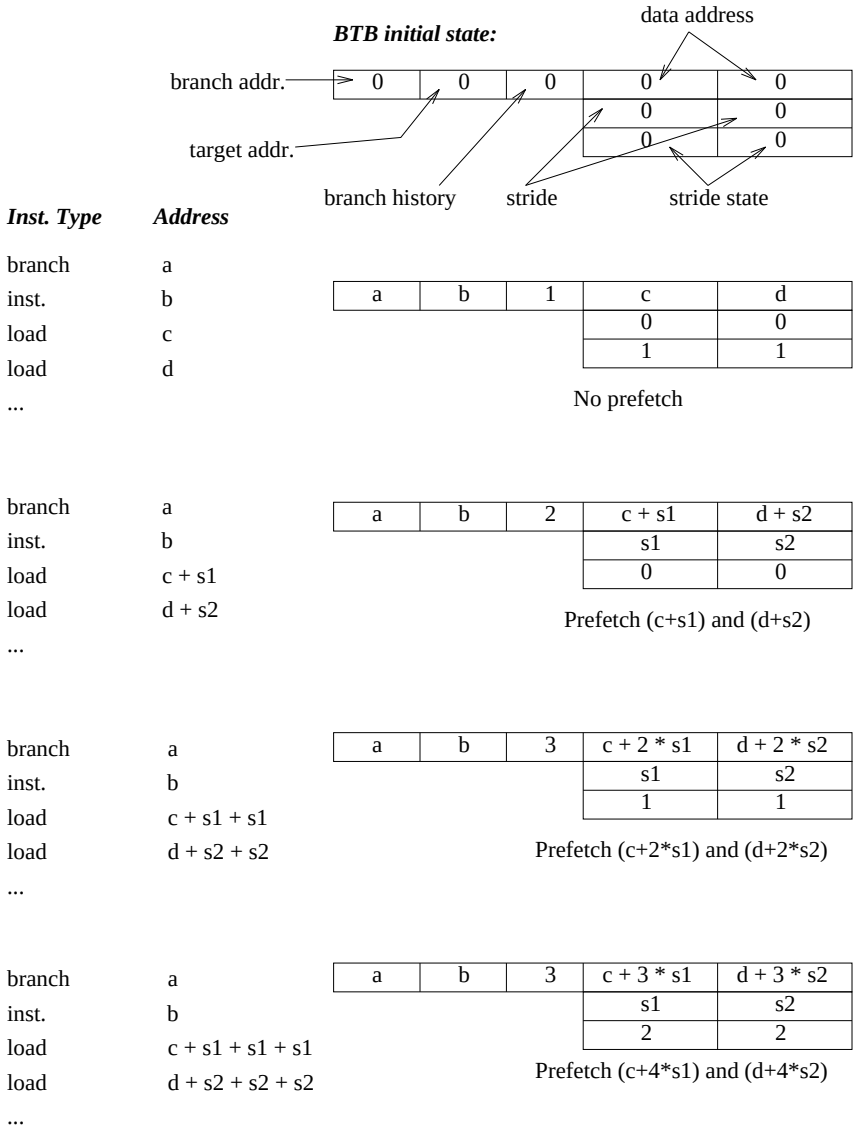

Figure 4: An example of BTB maintenance and branch directed prefetching. The BTB has two data entries and branch *a* is well behaved.

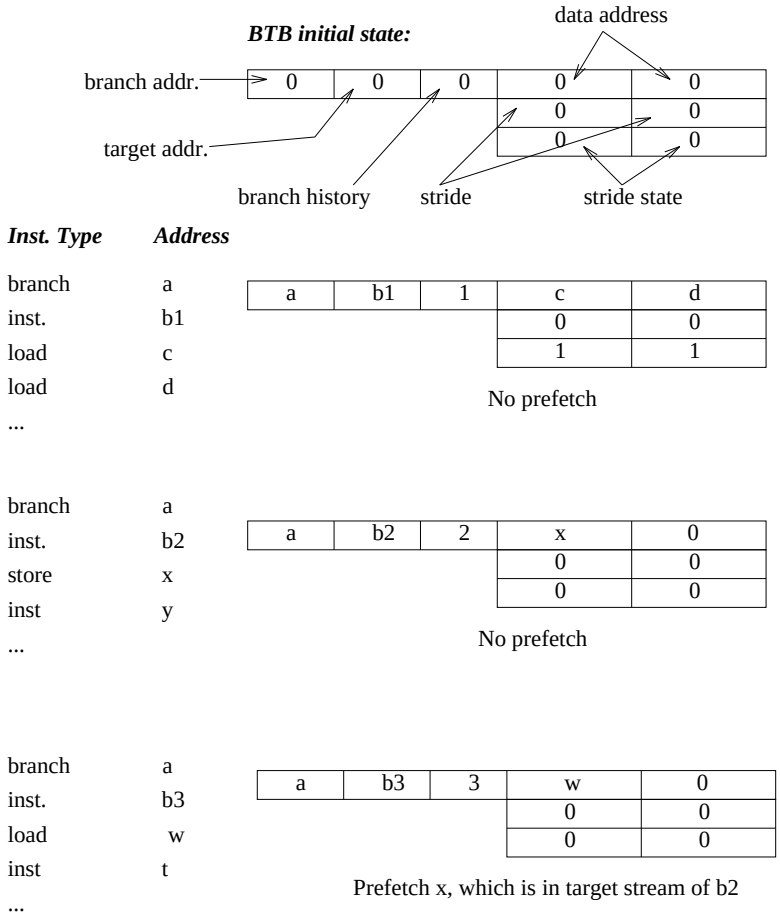

Figure 5: An example of moving target branch and a wrong prefetch

evaluates three schemes: 1) *Greedy Prefetching*, 2) *History-Pointer Prefetching*, and 3) *Data-Linearization Prefetching*. Greedy Prefetching attempts to prefetch down all next pointer paths (for k-ary recursive data structures this will result in k prefetches per node transversal). History-Pointer Prefetching attempts to be more accurate by recording information on past pointer traversal through the linked data structure. Data-Linearization Prefetching tried to improve spatial locality of the linked structure by reorganizing where in memory this structure is dynamically allocated. The results from their work showed that Greedy Prefetching can improve overall execution time, but can also lead to a large number of unnecessary prefetches. History-Pointer Prefetching can further reduce execution time, while significantly reducing the number of prefetches. Data Linearization will further improve the effectiveness of prefetching, while reducing overall execution time. All of these results were based on analysis of programs rich in pointer-based data structures.

While Luk and Mowry's work has some similarities to ours, they only suggest a software implementation of their technique. We suggest that our mechanism can be implemented in either hardware or software. We also differ in that our mechanism identifies the allocation of memory for the next sequential element in the list and records the corresponding *next pointer* link. This link is stored in a table which is indexed using a linear index (i.e., the element's position in the list). When a pointer-based data address is referenced, the table will initiate a prefetch for the next consecutive element in the list (we predict that the list is accessed sequentially).

Lipatsi et al. describe a mechanism called *SPAID* which issues a single prefetch upon calls which contain pointer-based arguments [27]. The thought is that these pointer will be immediately dereferenced in the called procedure. Luk and Mowry compare SPAID to Greedy Prefetching and found Greedy to slightly outperform SPAID. Our work here resembles Greedy Prefetching, though we only maintain a single next pointer address.

To implement our scheme, we have designed a mechanism which detects calls to the dynamic memory allocator (e.g., malloc). We build a table of pointers during the allocation of the list. Each table is indexed by the instruction address of the calling instruction to the dynamic memory allocator. The address of the head of the dynamically allocated list structure is used as an index. Each list is managed in a separate table. We also can handle deallocations (e.g., frees), updating the prefetching table accordingly.

This mechanism can be implemented using hardware or software prefetching. If hardware prefetching is employed, once the head of the list is accessed, a portion of the pointer-based prefetch table is loaded into a hardware prefetch mechanism and the table is used to issue prefetches accordingly. We also have the option of prefetching a number of nodes ahead in order to match the inherent latency in the memory hierarchy with the amount of execution overlap. This issue is discussed later in this paper.

Similarly, if software prefetching is used, the table will contain a list of offsets which are used by non-binding prefetch instructions generated by the compiler [28]. An index register is used to sequence through the list of prefetch index values. The prefetch instruction performs an indirect prefetch. Next we will evaluate the merits of these two techniques using trace-driven simulation.

4 Simulation Results

4.1 Modeling Branch Directed Prefetching

Simulation models for four different branch directed prefetching implementations are studied. We label these models BDPM#1, BDPM#2, BDPM#3 and BDPM#4. In these models, various data prefetching techniques are combined with branch directed prefetching.

As previously stated, our BTB structure takes advantage of stride prefetching. BDPM#1, BDPM#2 and BDPM#3 use the same BTB as illustrated above. BDPM#4 has a different BTB structure to cache prefetching data for both taken and not taken branches.

BDPM#1 is the simplest branch directed prefetching protocol, and only maintains a BTB which caches data from the taken branch target stream. The BTB is interrogated to determine if a branch has been found in the instruction address stream. When a branch instruction is found, the decision of what to prefetch is made based on the information stored in the data buffer fields, the stride fields and the stride state counter.

BDPM#2 uses tagged prefetching as well as branch directed prefetching. This protocol investigates whether the two prefetching schemes can work together without degrading each other's performance. To implement tagged prefetching, an 1-bit tag is added to each data cache entry. The decision of whether to prefetch is made depending upon the results of a data cache access. In BDPM#2, both tagged prefetching and branch directed prefetching work autonomously without sharing information with each other.

BDPM#3 is an attempted optimization. It also uses tagged prefetching and branch directed prefetching together. But this time, these two prefetching techniques share information. The shared information is the first data address in the branch directed buffers. The motivation to correlate these two prefetching techniques is because of the following considerations:

1. Data references before and after a taken branch may be in different memory blocks. Therefore the *principle of locality* may not hold when a branch is taken. Continuing to perform tagged prefetching is not efficient when a branch is taken, because the prefetched cache line will likely to be useless.

2. Assuming correct branch prediction, when a branch is taken, the data reference which is most likely to be accessed in the near future is the one cached in the BTB. Letting tagged prefetching share this information should allow it to work more efficiently.

Like BDPM#3, the BDPM#4 model includes having the BTB collaborate with tagged prefetching. Moreover, BDPM#4 maintains data references located in both taken and not-taken target streams (but only one stream at any one time). If a branch is predicted taken, the data addresses on the taken path are prefetched into data cache; otherwise, the data addresses on the not-taken path are prefetched.

4.2 Execution Driven Simulation

Cache simulation is performed on a DEC Alpha platform using Digital's ATOM toolset [29]. Five benchmark programs from the SPEC benchmark suite are selected for our analysis. They are all integer benchmarks written in C. The rest of the benchmark suite has been studied, but we provide results for only a subset of the benchmarks which possess interesting branching patterns (i.e., patterns which are more difficult to predict). We do not want to downplay the dependency of our technique on the predictability of branches. The input files used for the benchmarks are listed in Table 2. Some characteristics of the traces used in this work are shown in Table 3.

4.3 Simulation Results and Analysis

We model an 8 KB data cache (similar in size to the DEC AXP 21164). While SPEC programs provide a small footprint, we are able to see the benefits of our mechanism even with these well-behaved programs. Operating system laden workloads should further benefit from our strategy. The data cache associativity

	Gcc	Eqntott	Espresso	Sc	Xlisp
input file	gcc.i	int_pri_3.eqn	bca.in	loada1	li-input.lsp

Table 2: Input files for the benchmark programs during the simulation

	Gcc	Eqntott	Espresso	Sc	Xlisp
inst	57152583	1802796455	409359049	6483489	277861948
condb	6797723	194374477	74002505	772727	32497316
uncb	474918	5264880	1027756	117834	3880126
jump	1052281	9707393	3622816	128745	12586509
bj	9287	681	2361	1207	1032
data	25175138	255534781	104935034	2502122	122091928
load	18028603	231145547	94196403	1813299	80095156
store	7146535	24389234	10738631	688823	41996772

inst : the number of instruction in the trace
condb : dynamic number of conditional branches
uncb : dynamic number of unconditional branches
jump : dynamic number of jumps
bj : static number branches and jumps
data : number of data references
load : number of loads
store : number of stores

Table 3: Trace statistics for the 5 SPEC benchmarks

was varied from direct-mapped to 8-way. We model a 16 byte block, use LRU replacement and a write-allocate policy. While we could have modeled a longer block size (the DEC AXP 21164 uses 32 bytes), a smaller block size coupled with our aggressive prefetcher should allow us to be more precise without suffering extra demand misses.

We also model a 1024-entry, single-level, 2-way set associative BTB. Each entry consists of the full tag, two-bit counter field, and associated branch-directed prefetching information. Replacement in the BTB uses LRU. Branches are only entered in the BTB when the branch is taken.

Table 4 shows the prediction rate for conditional branches. The prediction rate for taken branches is the ratio of the number of branches which were predicted taken and were actually taken, divided by the number of branches which were predicted taken. The prediction rate for taken and not-taken branches is derived in the same manner.

	Correct Branch Prediction Rate	
Benchmark	Taken Branches	Taken and Not-taken Branches
Gcc	93%	85%
Eqntott	99%	97%
Espresso	91%	89%
Sc	92%	85%
Xlisp	99%	85%

Table 4: Correct branch prediction rate for conditional branches using a 2-bit branch prediction counter and a 2-way set associative, 512 entry per set BTB

The reported data cache hit ratios are the total number of data read hits for the entire execution of the program, divided by the total number of read references. Since the entire execution of the program is traced, the cold start misses are included in the miss ratio. Some abbreviations used in the graphs include: *(TP)*-tagged prefetching, *(BDP)*-branch directed prefetching, and *(BDP-TP)*-branch directed prefetching combined with tagged prefetching.

4.4 Comparison of Branch Directed Prefetching and Tagged Prefetching

Figures 6-10 compare data cache hit rates for tagged prefetching and our branch directed prefetching scheme. The results are consistent for the five benchmark programs (Gcc, Xlisp, Eqntott, Sc and Espresso). Both tagged prefetching and branch directed prefetching improve the data cache hit ratio. Branch directed prefetching is generally better than or as good as tagged prefetching. The only exception is Xlisp, where tagged prefetching has a slightly higher hit rate. This is because the data accesses for Xlisp exhibited greater spatial locality, which will especially favor tagged prefetching.

Although branch directed prefetching and tagged prefetching both attempt to hide the same latency, there are several distinguishing features. The major advantage of tagged prefetching is that it does not need a BTB to store extra data. Prefetching decisions are only based on the most recent data access result. With branch directed prefetching, prefetching decisions are dependent upon branch history. A BTB is needed (though most microprocessors today contain some form of on-chip branch prediction). The major advantages of branch directed prefetching are that: 1) it can handle changing stride-based

access patterns which can not be prefetched efficiently using tagged prefetching, and 2) prefetches can be issued well in advance of data use.

For all five benchmarks, branch directed prefetching shows considerable improvement in the data cache hit ratio when using a direct mapped data cache over a data cache without prefetching. The improvements in the data cache hit ratio when using branch directed prefetching with one data entry in the BTB over no prefetching at all are: 1.9% for Sc, 9.6% for Espresso, 1.1% for Xlisp, 0.3% for Gcc and 2.6% for Eqntott. For a direct mapped data cache using data prefetching, cache conflicts [5] can produce extra cache misses and reduce the benefits provided by the prefetching mechanism. This is because prefetching attempts to prime the cache with data which will most probably be needed in the near future. When the prefetching mechanism makes a mistake and prefetches instructions or data which will not be used, a cache line is needlessly replaced by the prefetched line. If the replaced line was going to be used in the near future, the erroneous prefetch will have introduced a cache miss which would normally not have occurred. This is called *cache pollution* [30]. However, for branch directed prefetching, there is less cache pollution produced. In most cases, branch directed prefetching brings useful data into the data cache. Only when the branch prediction is wrong, or when the data address behaves erratically, does branch directed prefetching produce pollution.

For 2-way, 4-way and 8-way set associative data caches, branch directed prefetching and tagged prefetching provide nearly the same data cache hit ratio. Both obtained higher data cache hit rates than caches without prefetching. In our branch directed prefetching scheme, the performance also depends upon the accuracy of the branch prediction policy and data access pattern prediction (i.e., the stride state counter). For the five benchmark programs, the correct branch prediction rates (shown in Table 4) are low (except for Eqntott, which spends a lot of time in a small portion of the code). If the correct prediction rate is increased by using more sophisticated prediction mechanisms (e.g., two-level predictors [8] or correlated branch predictors [7]), the accuracy of branch directed prefetching should further improve.

From the results of the BTB's with 1, 2, and 3 data entries, it can be seen that a BTB with 2 data entries has a better performance than a BTB with 1 data entry (especially for Eqntott, Gcc and Xlisp), and a BTB with 3 data entries only slightly outperforms a BTB with 2 entries.

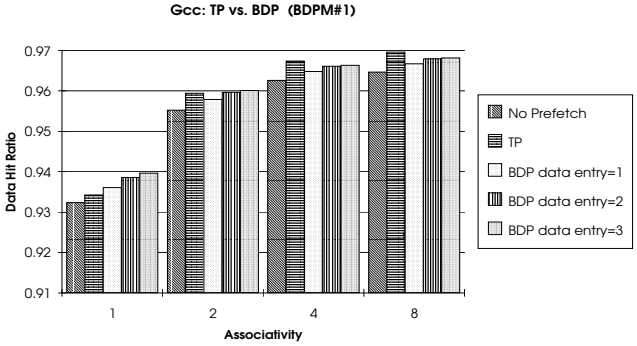

Figure 6: Gcc's result of tagged prefetching versus branch directed prefetching

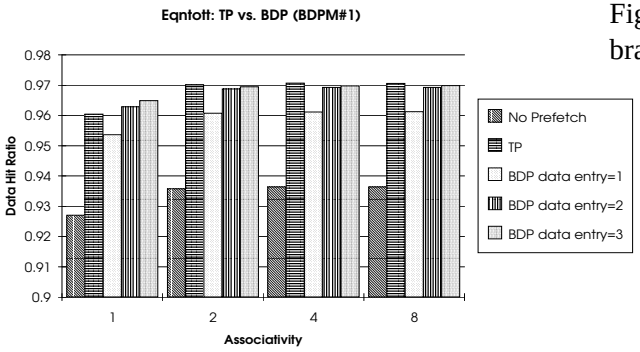

Figure 7: Eqntott's result of tagged prefetching versus branch directed prefetching

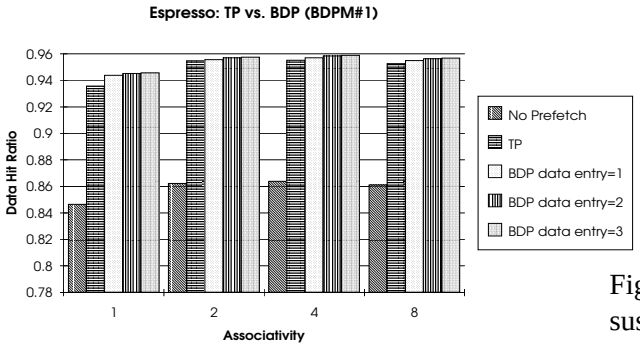

Figure 8: Espresso's result of tagged prefetching versus branch directed prefetching

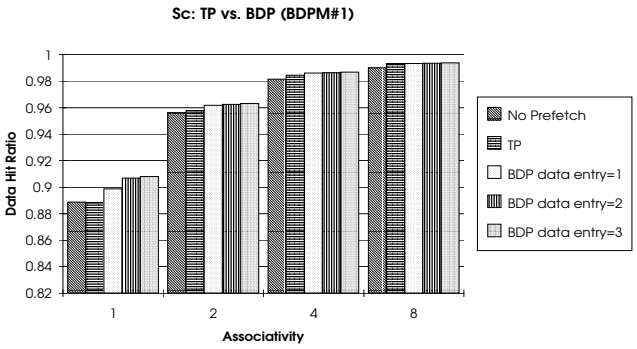

Figure 9: Sc's result of tagged prefetching versus branch directed prefetching

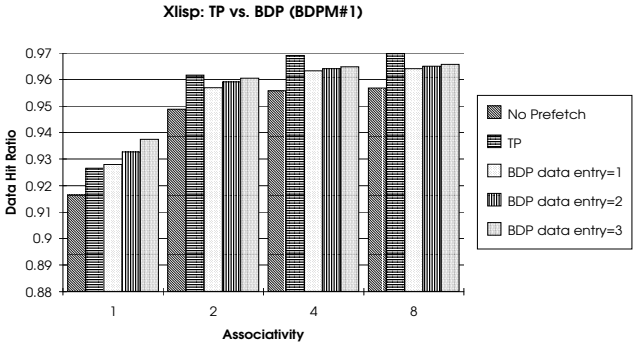

Figure 10: Xlisp's result of tagged prefetching versus branch directed prefetching

In Figure 11 we compare the number of bus accesses generated for tagged prefetching and branch directed prefetching for the five applications. The results shown are for a direct-mapped cache with 1 data buffer being used in the branch directed prefetch scheme. While the number of misses is reduced for all but the Eqntott application, the number of prefetch accesses is dramatically reduced for all programs. While tagged prefetching is providing improved cache hit rates by assuming a sequential data stream, the amount of bus traffic introduced is substantial. We can still enjoy the same cache hit rates using branch directed prefetching, but we can considerably reduce the amount of bus traffic (from 20% for Espresso to over 50% for sc).

When we increase the number of prefetch data buffers stored in the BTB, the number of prefetch accesses increases, but the number of misses is reduced to offset the increased bus traffic. For 2 and 4-way set associative caches, less bus traffic occurs due to misses, but branch directed prefetching still produces fewer bus accesses. This is because branch directed prefetching is more accurate than tagged prefetching.

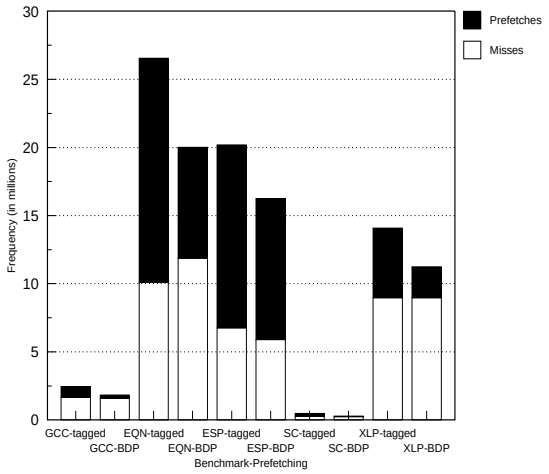

Figure 11: The bus traffic of tagged prefetch versus branch directed prefetch (1 data address from target stream, direct-mapped data cache.)

4.5 Branch Directed Prefetching Working with Tagged Prefetching

We have seen that branch directed prefetching can increase cache hit rates substantially. An interesting question is whether branch directed prefetching can gain any extra advantage when combined with tagged prefetching. Figures 12-16 show the results for the five benchmarks. As is apparent from the results, there are potential benefits to allow branch directed prefetching to work in concert with tagged prefetching.

First, let us consider the combined effect of branch directed prefetching and tagged prefetching. The main benefit of combining the two, of course, is that each scheme can compensate for the other scheme's weakness. Tagged prefetching takes advantage of spatial locality by prefetching the next sequential cache line in the memory address space, and thus can ensure that the sequential data streams are in the cache. Branch directed prefetching exploits both spatial (stride) and temporal (loop-related) locality. It can prefetch data in a branch target stream. This primes the cache with useful data when the branch is predicted correctly. Allowing these two to work together is theoretically more advantageous for complex data reference behavior (i.e., operating system code, general purpose applications).

The results in Figures 12-16 show that adding tagged prefetching helps achieve higher cache hit rates than when either tagged prefetching or branch directed prefetching is used alone. Again, it can be seen that the most substantial improvements are obtained for direct mapped data caches. For set associative data caches, combining branch directed prefetching and tagged prefetching also gives better hit rates than only using tagged prefetching. This indicates that branch directed prefetching can prefetch useful data which tagged prefetch is not able to. Allowing these two prefetching mechanisms to work together can let them solve the other's deficiency. Also remember that these results are based on a very simple branch prediction model which uses a simple 2-bit up/down branch prediction counter. If a more sophisticated scheme is used, branch directed prefetching can be more accurate.

For Espresso, a significant improvement (more than 2.5%) is achieved (see Figure 14). The main computations Espresso performs are set operations (e.g. union), with the sets being represented as arrays of bits. The set operations are then implemented as logical operations on these arrays. Our branch directed prefetching simulation models incorporate stride prefetching which is especially good for prefetching array data. Therefore, the branch directed prefetching scheme that we implemented provides considerable improvement for Espresso. Note that when working separately, both branch directed prefetching and tagged prefetching have improved hit ratios by 10% (see Figure 8). Therefore the 2.5% increase in the data cache hit ratio of an 8 KB produces a cache hit ratio of 98%.

4.6 Branch Directed Prefetching Collaborating with Tagged Prefetching

Having branch directed prefetching collaborate with tagged prefetching is an attempt to further enhance the branch directed prefetching scheme. The results in Figures 17-21 break down into three groups:

1) Tagged prefetching

2) Branch directed prefetching collaborating with tagged prefetching, using a BTB which only enters branches when first taken. Only the data addresses in the taken target stream are prefetched. This protocol is labeled as BDPM#3.

3) Branch directed prefetching collaborating with tagged prefetching, using a BTB which enters both taken and not taken branches and stores data for both taken and not taken (fall through) target streams. If the branch is predicted taken, the data for the taken target stream are prefetched into the data cache; otherwise, the data for the fall through stream are prefetched. This protocol is labeled as BDPM#4.

The results of both BDPM#3 and BDPM#4 are better than for tagged prefetching. The data shows that prefetching data from both taken and not taken branch target streams can produce higher cache hit rates. These results also suggest that using branch directed prefetching without a tagged prefetching scheme which stores data for both taken and not taken target streams is a better policy than only storing data for the taken target stream.

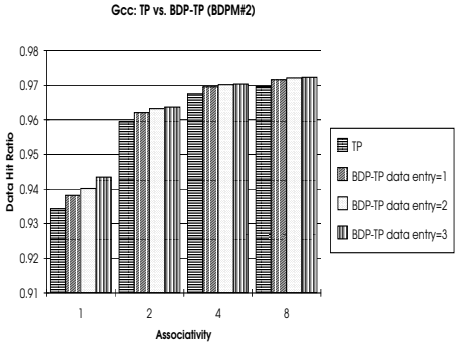

Figure 12: Gcc's result of tagged prefetching and branch directed prefetching working autonomously

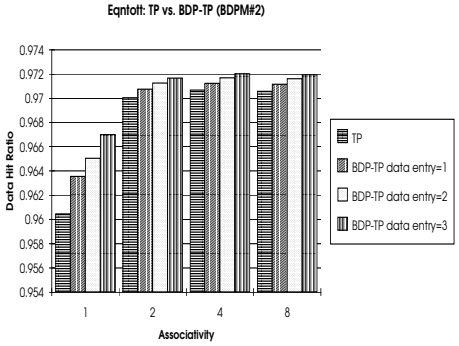

Figure 13: Eqntott's result of tagged prefetching and branch directed prefetching working autonomously

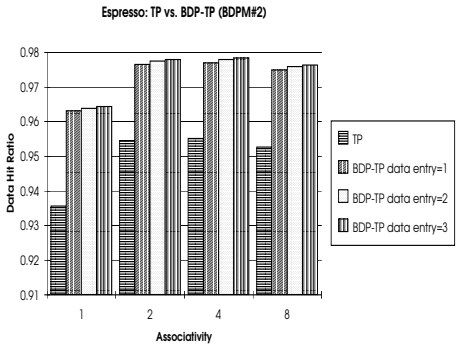

Figure 14: Espresso's result of tagged prefetching and branch directed prefetching working autonomously

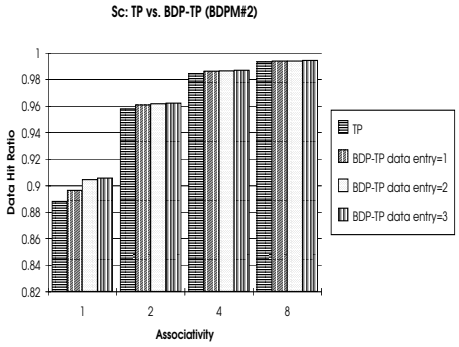

Figure 15: Sc's result of tagged prefetching and branch directed prefetching working autonomously

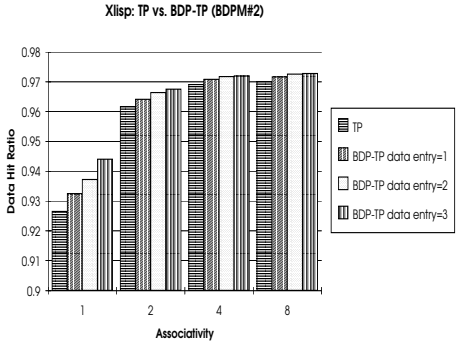

Figure 16: Xlisp's result of tagged prefetching and branch directed prefetching working autonomously

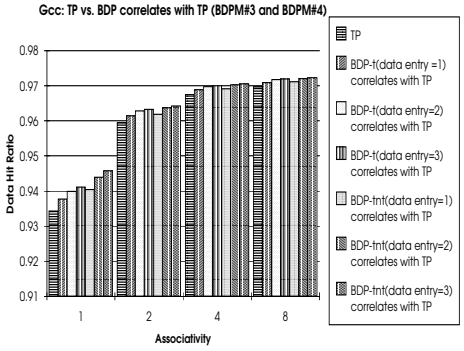

Figure 17: Gcc's result of correlating branch directed prefetching with tagged prefetching

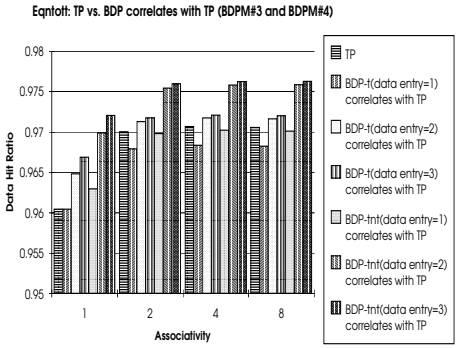

Figure 18: Eqntott's result of correlating branch directed prefetching with tagged prefetching

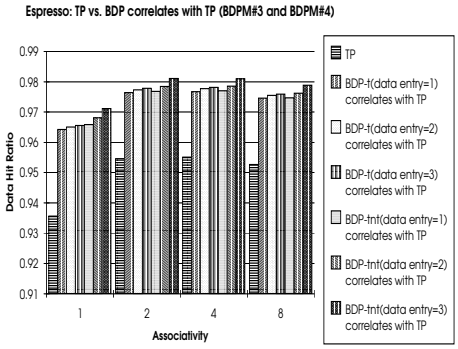

Figure 19: Espresso's result of correlating branch directed prefetching with tagged prefetching

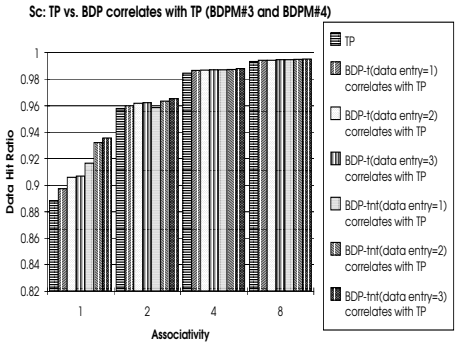

Figure 20: Sc's result of branch directed prefetching with tagged prefetching

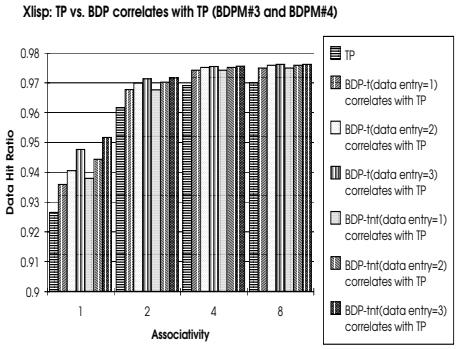

Figure 21: Xlisp's result of branch directed prefetching with tagged prefetching

Although BDPM#3 works better than tagged prefetching, it produces results quite similar to BDPM#2 (see the right portion of the graphs in Figures 12-16). In some circumstances (i.e., correct branch prediction and the data located in the branch target stream are not in the data cache), BDPM#3 would have one more data hit per taken branch than BDPM#2. This is because when a branch is taken, without recording the information of branch target stream, tagged prefetching will prefetch the old sequential stream. Since the hit rates for the benchmarks are already very high and the branch prediction scheme is very simple, the benefits of correlation are diminished.

4.7 Pointer-based Prefetching Results

Next we evaluate the merits of Pointer-based Prefetching using trace-driven simulation. We will focus our discussion around a single program, Gcc, since it contains the largest number of pointer-based accesses (in the final version of this paper we will provide results for all of the SPEC 95 benchmarks). Table 5 summaries some of the characteristics with respect to dynamic memory allocation for the Gcc benchmark.

benchmark	frequency of calls	avg bytes per call
Gcc	11,363	846

Table 5: Frequency of calls to malloc and average number of bytes allocated per call

We modeled an infinite pointer-based prefetcher in order to illustrate the potential for such a mechanism. We also use a second index register to track the offset into the data structure, indicating what field is currently being referenced (note that the average size of a single element is 846 bytes, and our cache is configured with 16 byte lines). We only issue a single prefetch per list element traversal. We are handling only the most common case of a forward traversal of a linked list. Tagged prefetching should take care of prefetching the next pointer field (we have found that next pointer fields tend to be located at the end of the data structure).

We present the reduction in read miss rate for the same data cache configurations as presented above. Results appear in Figure 22. As we can see, even using a simple greedy pointer-based prefetching scheme, we are able to significantly reduce the number of data cache misses for the Gcc benchmark. As we increase associativity, we find that pointer-based prefetching continues to provide an advantage. This is due to the increased associativity being able to tolerate the cache pollution introduced by pointer-based prefetching.

5 Discussion

One problem we have not rigorously addressed in this paper is the issue of when to prefetch. Ideally we would like to prime the cache such that the needed data arrives just before it is requested. For array accesses present in looping control structures, this can be done with a high level of accuracy. For data accesses contained in less predictable control structures (e.g., `If-Then-Else` blocks), we are at the mercy of the accuracy of control flow predictor. With an accurate multiple-block branch predictor [31], the prefetch distance could then span a number of basic blocks.

Prefetching across multiple basic blocks without the aid of hardware branch prediction may lead to a large number of wrong prefetches and a large amount of cache pollution. The History-Pointer Prefetching

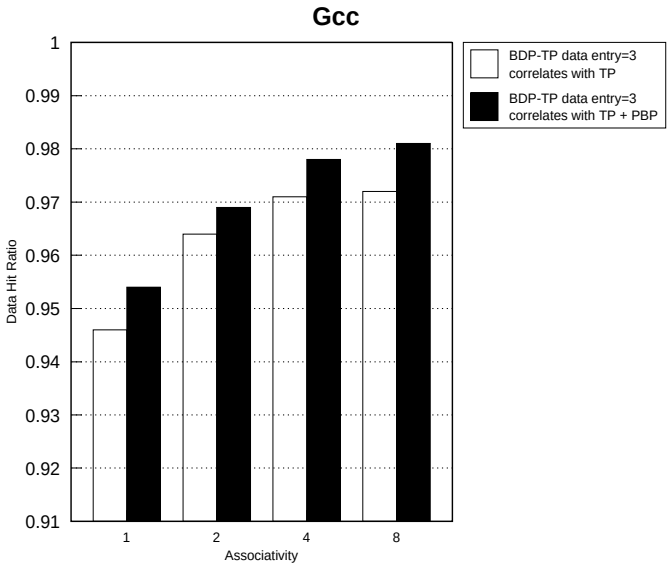

Figure 22: Gcc result of branch directed prefetching with tagged prefetching compared to adding pointer-based prefetching

approach suggested by Luk and Mowry may overcome this problem, though we may have to add more history in order to prefetch far enough in advance. Value-prediction may also be useful in order to produce the best prefetch stream [32]. We plan to pursue these questions in future work.

6 Conclusion

Cache memories are commonly used to reduce the performance gap between microprocessor and memory speeds. Prefetching can be employed to increase the chances that a memory line will reside in the cache when it is requested by the microprocessor. Prefetching attempts to prime the cache with memory lines which will be needed in the near future.

In this paper we have presented two prefetching schemes for the data stream. Branch directed prefetching utilizes a Branch Target Buffer to prefetch the data stream while prefetching the instruction stream. We compared our scheme against traditional prefetching techniques and found that we could improve the data cache read miss rate by as much as 2.5%. It was also demonstrated that when used alone, branch directed can provide similar hit rates as tagged prefetching, while producing far less memory bus traffic (reduced by as much as 50%).

Further investigation of combining a branch directed scheme with tagged prefetching showed that cache hit rates could be substantially improved. Tagged prefetching captures the spatial locality present in the data stream. Branch directed prefetching captures the temporal locality present in the data stream. Combining the two allows each mechanism to compensate for the weakness of the other.

Since most high-performance microprocessors today provide on-chip branch prediction, branch directed prefetching would seem to be a natural implementation. Branch directed prefetching should provide even more of an advantage as the accuracy of branch prediction algorithm is improved.

We also described a new prefetching for pointer-based data references. We found that by recording

the sequence of heap allocated data structures, that we could effectively reduce the number of data cache misses over using branch directed prefetching (we reduce the cache miss rate by as much as .9% for Gcc). Our mechanism can be implemented using either hardware or software, and can be adjusted to provide the appropriate prefetch distance.

The authors would like to acknowledge the contributions of Po-Yung Chang and Nancy Perugini on this work, and would like to thank Alan Eustace for providing the ATOM tools used in this work.

References

[1] M. D. Hill, "Aspects of Cache Memory and Instruction Buffer Performance," *PhD dissertation, University of California Berkeley*, 1987.

[2] A.H. Hashemi, D.R. Kaeli and B. Calder, "Efficient Procedure Mapping using Cache Line Coloring," *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation*, June 1997, pp. 171-182.

[3] A. Seznec, "A Case for Two-Way Skewed-Associative Caches," *Proceedings of the 23 Annual International Symposium on Computer Architecture*, May 1996.

[4] A. Seznec, "Skewed Associative Caches," *Proceedings of PARLE'93*, May 1993.

[5] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proceedings of the 17th International Symposium on Computer Architecture*, May 1990, pp. 364-373.

[6] N. Gloy, C. Young, B. Chen and M. Smith, "An Analysis of Dynamic Branch Prediction Schemes on System Workloads," *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996.

[7] C. Young, N. Gloy, and M. Smith, "A Comparative Analysis of Schemes for Correlated Branch Prediction," *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[8] T. Y. Yeh, Y. N. Patt, "Alternative Implementations of Two-level Adaptive Branch Predictions," *Proceedings of the 19th International Symposium of Computer Architecture*, May 1992, pp. 124-134.

[9] T. Y. Yeh, Y. N. Patt, "A Comparison of Dynamic Branch Predictors That Use Two Levels of Branch History," *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993, pp. 257-266.

[10] D. Kaeli, L. Fong, D. Renfrew, R. Booth, and K. Imming, *IBM Journal of Research and Development*, special issue on performance tools, to appear 1997.

[11] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W. D. Weber, "Comparative Evaluation of Latency Reducing and Tolerating Techniques," *Proceedings of the 18th Annual International Symposium on Computer Architecture*, May 1991, pp. 254-263.

[12] A. C. Klaiber, H. M. Levy, "An Architecture for Software-Controlled Data Prefetching," *Proceeding of the 18th Annual International Symposium on Computer Architecture*, May 1991, pp. 43-53.

[13] A. J. Smith, "Cache Memory," *ACM Computing Surveys*, Vol. 14, No. 3, Sept. 1982, pp. 473-530.

[14] P.J. Denning and S.C. Schwartz, "Properties of the Working-Set Model," *Communications of the ACM*, Vol. 15, No. 3, March 1972, pp. 191-198.

[15] J. D. Gindele, "Buffer Block Prefetching Method," *IBM Tech. Disclosure Bulletin*, 20, 2, July 1977, pp 696-697.

[16] J. W. C. Fu and J. H. Patel, "Stride Directed Prefetching in Scalar Processors," *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 1992, pp. 102-110.

[17] T. F. Chen and J. L. Baer, "Reducing Memory Latency via Non-blocking and Prefetching Caches," *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992, pp. 51-61.

[18] T. F. Chen, *Data Prefetching for High-Performance Processors*, PhD. Thesis, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA, July 1993, also appearing in a technical report, 93-07-01.

[19] D. Joseph and D. Grunwald, "Prefetching using Markov Predictors," *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997, pp. 252-263.

[20] M.J. Charney and A.P. Reeves, "Generalized Correlation Based Hardware Prefetching," Technical Report EE-CEG-95-1, Cornell University, Feb. 1995.

[21] P. Y. Chang, D. R. Kaeli and Y. Liu, "Branch-Directed Data Cache Prefetching," *Proceedings of the 2nd Annual Workshop on Shared-Memory Multiprocessor Systems*, Chicago, IL, 1994.

[22] Y. Liu and D. R. Kaeli, "Using a BTB to Guide Data Cache Prefetching," *Proc. of the 1996 IEEE International Conference on Computer Design*, Austin, Tx., Oct. 1996.

[23] J. E. Smith, "A Study of Branch Prediction Strategies," *Proceedings of the 8th Annual International Symposium on Computer Architecture*, June 1981, pp. 135-147.

[24] D. R. Kaeli, P. G. Emma, "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns," *Proceedings of the 18th Annual International Symposium on Computer Architecture*, May 1991, pp. 34-41.

[25] C.-K. Luk and T.C. Mowry, "Compiler-Based Prefetching for Recursive Data Structures," *Proceedings of the 7th Annual Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996, pp. 222-233.

[26] S. Mehrotra, *Data Prefetch Mechanisms for Accelerating Symbolic and Numeric Computation*, PhD Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1996.

[27] M.H. Lipatsi, W.J. Schmidt, S.R. Kunkel, and R.R. Roediger, "SPAID: Software Prefetching in Pointer and Call-Intensive Environments," *Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1995.

[28] T.C. Mowry, *Tolerating Latency Through Software-Controlled Data Prefetching*, PhD Thesis, Stanford University, March 1994, Technical Report CSL-TR-94-626.

[29] "ATOM User Manual," *Digital Equipment Corporation*, Maynard, MA., March 1994.

[30] J. P. Casmira, D. R. Kaeli, "Modeling Cache Pollution," *Proceedings of the 1995 IASTED Modeling and Simulation Conference*, Pittsburgh, PA, April 1995, pp. 123-126.

[31] S. Wallace and N. Bagherzadeh, "Multiple Branch and Block Prediction," *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, Feb. 1997, pp. 94-103.

[32] M.H. Lipatsi, C.B. Wilkerson, and J.P. Shen, "Value Locality and Load Value Prediction," *Proceedings of the 7th Annual Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996, pp. 138-149.