Characterization of Repeating Data Access Patterns in Integer Benchmarks

Erik M. Nystrom* Roy Dz-ching Ju† Wen-mei W. Hwu‡
enystrom@uiuc.edu roy.ju@intel.com w-hwu@uiuc.edu

Abstract

Processor speeds continue to outpace the memory subsystem making it neccesary to proactively acquire and retain important data. Current applications have an ever increasing number of dynamically allocated data structures and these data structures occupy large footprints. A large portion of dynamically allocated data is accessed through pointers in the form of recursive data structures. Loads accessing these data structures often become a data cache bottleneck because they depend on a previous instance of themselves, scattering their accesses. For this reason, optimization techniques specific to these types of loads should be employed. One technique is to analyze the data stream and use the resulting information to guide prefetching and/or dynamic data layout.

This work characterizes the repeating data access patterns in both the Olden and SPEC CPU2000I benchmark suites. The intent is to apply the analysis result to guide later optimizations. In this paper, we discuss our findings, with some benchmarks show interesting correlation in their data accesses. We have also obtained indication that such correlation manifests itself more prominently when a program's execution is divided into phases. A preliminary attempt was also performed in which analysis results were used to guide data prefetching.

1 Introduction

As the increase in processor speeds continues to outpace that of the memory subsystem, it is necessary to proactively acquire and retain important data. Various prefetching, cache management, and data layout techniques have been proposed to alleviate some of the demand for data. Current applications have an ever increasing amount of dynamically allocated data structures and these data structures occupy larger footprints. The trend is driven by the need for flexible, dynamic memory management, and object oriented programming styles. A large portion of dynamically allocated data are accessed through pointers in the form of recursive data structures (RDS), which include inductively created objects like linked lists, trees, graphs, etc.

RDS elements tend to be accessed within loops that serially dereference a previously loaded value, typically referred to as the pointer chasing problem. Such loads, whose target address depends on a value loaded by a previous instance of itself, are called induction pointer loads (IPL). The need to traverse through a sequence of data items within an RDS can place IPLs on critical execution paths. A study has shown that ensuring a cache hit for the load feeding into an indirect load has a significant performance potential [1].

This work characterizes repeating data access patterns in both the Olden and SPEC CPU2000I benchmark suites. The intended use of these analysis results is to guide later optimization. While previous work has primarily focused on suites of small benchmarks, such as the Olden benchmarks, this work looks at the latest version of the SPEC benchmarks from the CPU2000I suite. These benchmarks are known to have significant data cache performance issues and will influence the design of microprocessors for years to come. The Olden benchmarks provide a common frame of reference with previous work while the behaviors of the CPU2000I benchmarks provide insight into larger, less understood applications.

2 Framework

Figure 1 shows a potential analysis and optimization path. The data stream is analyzed and the resulting

*Coordinated Science Lab 1308 West Main Street, MC-228 Urbana, IL 61801

†Intel Corp. Microprocessor Research Labs 3600 Juliette Lane, Santa Clara, CA 95052

‡Coordinated Science Lab 1308 West Main Street, MC-228 Urbana, IL 61801

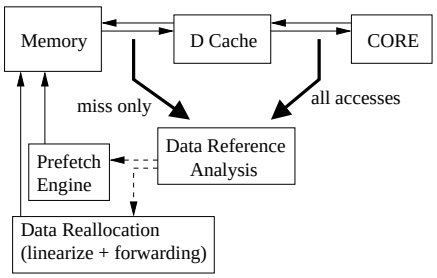

Figure 1: Potential analysis and optimization framework.

information can be used to guide prefetching and/or dynamic data relayout. There exist several different possible configurations. Every reference in the data stream can be inspected, or only those that miss at a certain cache level. A hardware and compiler collaborative approach can be used where detailed static analysis guides the hardware. Another possibility is that a simple form of the analysis could be peformed by the hardware for guiding CPU side data prefetching, memory side prefetching, or a dynamic data reallocation and layout scheme. Once information is collected, it can be used dynamically in the same run to optimize the remaining program execution or it can be directed to optimize a subsequent run.

As a step toward this goal, this paper characterizes a variety of benchmarks based on the their data access patterns. Namely, this work characterizes repeating data access patterns in both the Olden and CPU2000I benchmark suites for the purpose of driving future optimization of their data access streams.

The process flow is depicted in Figure 2. First, each benchmark was compiled using an Intel compiler during which time the IPLs were annotated using a detection algorithm similar to the work by Gerlek et al [2]. An IPL is a load whose load address depends on a value loaded by a previous instance of itself. Informally, IPLs are those pointer loads used in loops to traverse RDSs.

Second, each benchmark was executed on a functional simulator to generate a data reference trace, including the afore mentioned annotations. The propagation of the annotations allows the next phase to keep separate statistics and perform analysis on only IPL accesses. Third, a trace of the data reference stream was analyzed for both data cache performance and data reference affinity. The effects of phasing on reference affinity and of a simple prefetching heurisic on cache performance were also measured.

To help provide insight into the process, a sample code example is shown in Figure 3. The code fragment consists of four loads, *A, B, C,* and *D*. The code traverses a set of elements that contain a piece of data, loaded by *B*, and two pointers to other elements, loaded at *C* and *D*. Loads *C* and *D* are IPL loads, while *A* and *B* are not. A sample reference stream is also shown and is labeled "Trace All" and the load that generated the reference is shown above the reference. The analysis system looks at a single, filtered trace mix. "Filtered" means that only references generated by IPL loads will be examined. A single trace mix means that references from different static IPLs are mixed together. In Figure 3 the stream labeled with "IPL Only" would be the resulting trace used for analysis.

B and *C* from Figure 3 also show how an IPL and non-IPL can be strongly bound. Given their ordering and assuming they share a cache line, *B* will always miss and *C* will always hit. However, the pattern useful for preventing *B* from missing is controlled through *C*.

As previously mentioned, one of the measurements is data reference affinity. Data reference affinity refers to the tendency of certain data references to preceed or follow other data references. For example, if an access to *B* always follows an access to *A*, then *A* and *B* exhibit very strong affinity. This will be explained in detail in Section 4.2. The affinity analysis results are then used during a second benchmark run to guide prefetching.

In some cases, a benchmark's data access patterns show very limited amounts of affinity. However, the data collected is an aggregate across the entire program's execution. While this aggregate number is important, multiple phases of strong affinity, after being averaged together, can obscure each other. An experiment was performed which took two of the benchmarks that exhibited poor affinity and performed a rudimentary phased analysis of the reference stream. The amount of affinity within the phases helps determine if information is being obscured by looking at the program's execution as a whole.

The benchmarks bisort, em3d, health, treeadd, and tsp were selected from the Olden [3] bench-

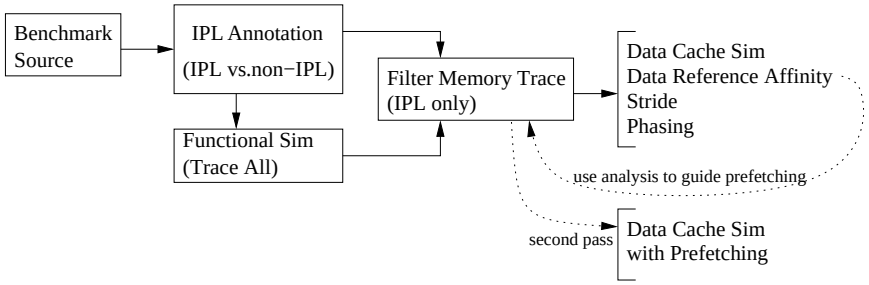

Figure 2: Experimental framework.

```
Ⓐ r4 = ld global_label      (non−IPL)
loop:
Ⓑ r1 = ld r2                (non−IPL)
   <...>
   br <cond> other
Ⓒ r2 = ld r2 + 8            (IPL)
     (e.g. p=p−>next)
   jmp loop
other:
Ⓓ r2 = ld r2 + 16           (IPL)
   jmp loop
```

```
Load     : A B  C  B  C B  C B  D   B  C
Trace All: 7 7 32 13 64 2 96 8 256 21 128

IPL Only : 32 64 96 256 128
```

Figure 3: Code fragment with annotations and an example trace stream.

mark suite and crafty, eon, vortex, vpr, mcf, parser, and twolf were selected from CPU2000I. Due to a marked slowdown from data collection, the CPU2000I benchmarks were run using the light input sets to reduce simulation time. The CPU2000I light input sets were developed proprietarily at Intel to suit the simulation-based development environment. These carefully generated light input sets exhibit roughly the same performance characteristics, such as program hot spots, cache miss rate, branch mis-prediction rate, etc. as the reference input sets. Most of the programs used in our experiments were run to completion, with the exception of three which were only partially executed due to simulation time, namely twolf (2 billion instructions), mcf (2 billion instructions), and parser (500 million instructions).

3 Related Work

Data cache performance is a bottleneck in many large integer programs and is an issue continued to be studied. For both array references and recursive data structures many methods exist for prefetching data. Reinman and Calder survey and evaluate a variety of methods including stride, context, and hybrid predictors [4]. Stride prediction uses the last seen address and a stride value to determine a future address. Context predictors use a history of previous sequence of addresses to predict the next address. Hybrid predictors simply combine stride and context prediction along with a decision mechanism. Additionally, runtime analysis of the reference stream can enable selective bypassing of infrequent accesses [5].

Luk and Mowry developed three methods to hide or remove the load latency of recusive data structures [6]. The first, greedy prefetching, prefetches along pointers pre-existing within RDS elements. By contrast, their second method called history prefetching uses a compiler annotated pointer that is dynamically set to a node recently accessed "d" steps away. Subsequent accesses to the element prefetch along the history pointer. Finally, data linearization maps RDS elements sequentially together so that they form an array-like structure, thus facilitating strided access.

Data layout techniques can result in more efficient data access. However, if data are moved dynamically care must be taken to ensure that all existing pointers to the data are updated. Luk and Mowry proposed memory forwarding which allows the safe relocation of data through the use of a forwarding address [7]. After relocating data, a forwarding address is left in the old data's location. If a reference to the old location exists, the forwarding address is automatically dereferenced by the hardware to yield the current data location.

Roth and Sohi present four classes (or idioms) of prefetching that consist of combinations of jump-pointer prefetching and chain prefetching [8]. Similar to Luk's history prefetching, jump-pointer prefetching uses pointers added to the data structure by hardware/compiler in order to facilitate prefetching. Chain prefetching uses pointers already part of the data structure for prefetching. The other two classes are queue jumping, which is just a specific instance of jump-pointer prefetching on very regular data structures, and root jumping, which consists of a series of chain prefetches for the purpose of aquiring an entire short, dynamic list.

In addition to more reactive prefetching techniques, Roth and Sohi proposed a mechanism through which data can be actively prefetched [9]. They first classified loads into two groups, recurrent loads and traversal loads. Recurrent loads produce addresses that future instances of themselves then consume. Traversal loads produce addresses that loads other than themselves consume. A correlation table is used to match producer loads with consumer loads. Using the producer-consumer information, a kernel of the execution can aggressively execute alongside and ahead of actual execution, prefetching data along the way.

To take advantage of repeating access patterns, Joseph and Grunwald designed a hardware structure that analyzes the miss stream of the processor and dynamically builds a Markov model for memory accesses [10]. The model is then used to prefetch multiple references from memory.

The relative placement of data can have a large effect on cache performance. A temporal relationship graph (TRG) was used by Calder et al. to statically guide the placement of local variables, global variables, and heap objects in a manner that improves data cache performance [11]. In a TRG the weight of an edge between two objects estimates the number of misses that could occur if the objects are placed into conflicting cache lines.

Data stucture layout tries to place temporally related elements into the same cache block and to separate frequently and infrequently accessed elements. In fact, cache conscious placement during generational garbage collection can improve cache performance for pointer code [12]. Similar to our methodology, an affinity graph, built with a history queue of length three, is used to determine relationships between objects. Since fields within a data structure element and elements within the entire data structure are rarely accessed equally, Chilimbi et al. propose techniques for both the definition and layout of a data structure [13] [14]. Both cases are motivated to improve cache performance.

Many of the previous aggressive prefetching and data layout schemes for integer benchmarks relied on strong and repeating correlation among data accesses to be effective. These works often evaluated their techniques using small benchmarks, such as the Olden suite, or non-standard benchmarks. It remains to be seen whether these techniques are still effective for larger benchmarks like CPU2000I. A key ingredient for answering this question is to identify how much affinity exists in the data accesses of these benchmarks.

4 Results

4.1 Cache Characteristics

Each benchmark's data cache performance was measured for both 64k and 128k 4-way set associative caches each using an LRU replacement policy. The hit and miss statistics were gathered and separated into IPL and non-IPL events. Figure 4 shows the cache results. The following characteristics can be readily determined for each benchmark:

1. miss ratios for 64k and 128k
2. fraction of IPL loads
3. fractions of misses due to IPL loads for 64k and 128k

From the results, the benchmarks em3d, health, crafty, mcf, and twolf all exhibit at least a 10 percent miss ratio. In a few benchmarks IPLs are more than 10 percent of all dynamic loads, namely bisort, health, mcf, parser, and twolf. By examining the combined results for each benchmark, the following characteristics can be extrapolated:

1. relative cache performance of IPL and non-IPL loads
2. IPL and non-IPL dependence on cache size

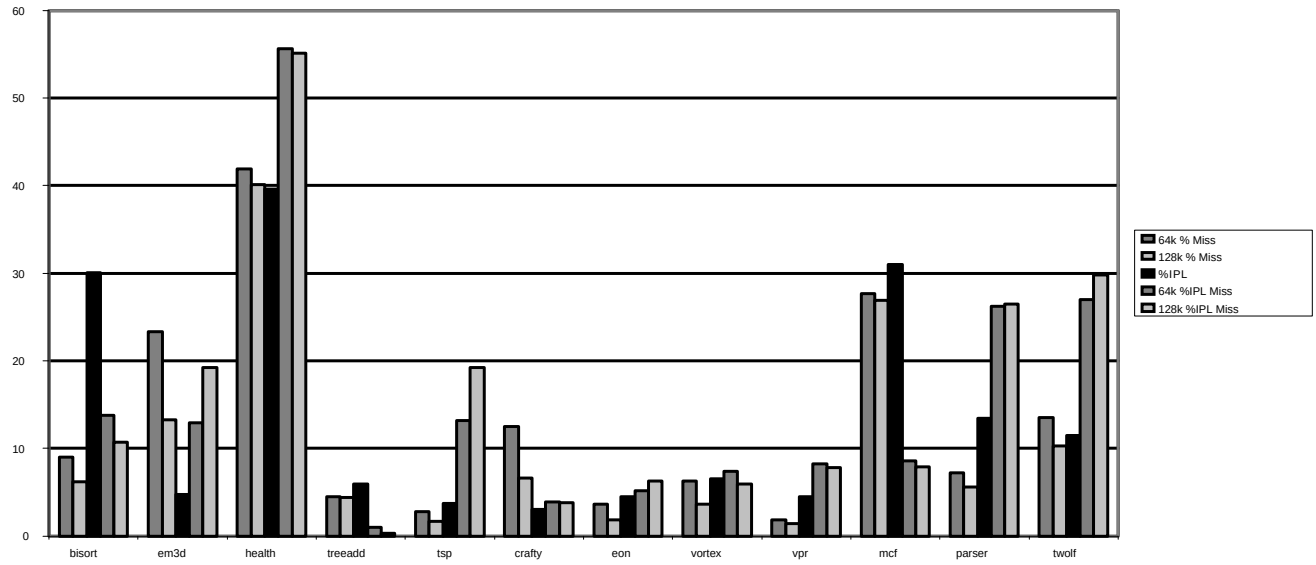

Figure 4: Cache results for Olden and CPU2000I.

If IPLs account for a large fraction of the total misses, then IPLs are a bottleneck in cache performance and techniques specific to these types of loads should be employed. If IPLs are infrequent or a large number of IPLs account for only a small fraction of the misses, then IPLs are not a bottleneck.

Taking a second look at the results in Figure 4, in a few benchmarks IPLs miss more frequently than non-IPLs on average. For example in em3d, IPLs represent only five percent of the total dynamic loads, while causing between 10 and 18 percent of the total misses. The benchmarks health, tsp, parser, and twolf also exhibit this property. The two noticeable cases where IPLs miss less frequently than non-IPLs are in treeadd and mcf.

The relative dependence of IPLs and non-IPLs on cache size can also be determined. If an increase in the cache size results in a decrease in overall misses but an increase in the fraction of these misses caused by IPL loads, then the larger cache is satisfying non-IPL loads, while having little effect on IPL-loads. This implies that IPL loads access a larger set of data or access the data in a more erratic manner. Should the fraction of IPL loads stay level, then additional cache space benefits both IPL and non-IPL loads equally. If the fraction decreases, then IPL loads benefit more from the additional space than non-IPL loads. This shows the relative effectiveness

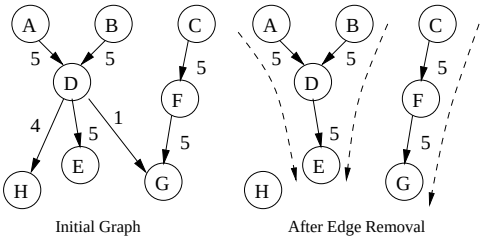

Figure 5: Affinity example.

of a cache size increase on reducing cache misses from IPLs and non-IPLs. Again referring to Figure 4, em3d, tsp, parser, and twolf show that the fraction of misses due to IPLs increases as the cache size is increased. Therefore blindly increasing the cache size may not yield much benefit to IPLs in these benchmarks.

4.2 Data Access Affinity

Data access affinity is the tendency of an access to a particular location to consistently follow an access to another location. If a sequence of accesses exhibits strong affinity, then the potential exists to take advantage of the stable sequence, for example by prefetching. A sequence of accesses exhibiting strong affinity will be referred to as a chain.

Chains are formed in the following manner. A stream of effective addresses (EA) from IPL references are generated by the functional simulator, as

Benchmark	%IPL	%Miss	%None	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
bisort-1 all	30.1	10.7	67	24	8												
bisort-1 mo	30.1	10.7	70	4	11	7	2	1									
bisort-8 all	30.1	10.7	91	6	1												
bisort-8 mo	30.1	10.7	89	5	3	1											
bisort-16 all	30.1	10.7	98	1													
bisort-16 mo	30.1	10.7	91	5	1												
em3d-1 all	4.7	19.2	14									85					
em3d-1 mo	4.7	19.2	0									99					
em3d-8 all	4.7	19.2	13						86								
em3d-8 mo	4.7	19.2	1						98								
em3d-16 all	4.7	19.2	7					85			6						
em3d-16 mo	4.7	19.2	13					86									
health-1 all	39.6	55.1	18									3	5	47	24		
health-1 mo	39.6	55.1	8							7	23	38	21				
health-8 all	39.6	55.1	27				4	33	29	3							
health-8 mo	39.6	55.1	24				48	20	4								
health-16 all	39.6	55.1	34			6	48	9									
health-16 mo	39.6	55.1	36			56	6										
treeadd-1 all	5.9	0.3	0														99
treeadd-1 mo	5.9	0.3	0											100			
treeadd-8 all	5.9	0.3	0														99
treeadd-8 mo	5.9	0.3	12								87						
treeadd-16 all	5.9	0.3	0														99
treeadd-16 mo	5.9	0.3	6							93							
tsp-1 all	3.7	19.2	35		3			1	58								
tsp-1 mo	3.7	19.2	28			2	62	1		1	1						
tsp-8 all	3.7	19.2	60	1	2	19	9	5	1								
tsp-8 mo	3.7	19.2	79	7	8	1		1									
tsp-16 all	3.7	19.2	72	3	18	3											
tsp-16 mo	3.7	19.2	92	3			1										
Benchmark	%IPL	%Miss	%None	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384

Figure 6: Affinity of data accesses in the Olden benchmarks.

mentioned in Section 2, and sent to the analysis tool. A history of the past 16 EAs is kept. Using this history, for each access to an EA the previous, 8th previous, and 16th previous EAs accessed are recorded. These distances were chosen to represent a variety of distances up to a distance usable for prefetching (i.e. 16 loads away).

Once execution completes, the collected access history data guides the formation of three graphs. Each node of the graph represents an EA. An edge from node *A* to node *D* of weight 5 means that *D* followed *A* 5 times as in Figure 5. One graph is composed of edges for the next access, one of edges to the 8th next, and one for edges to the 16th next.

The analysis is looking for the next most likely successor, thus the graph is reduced so that each node has at most one successor. To do this only the highest weight edge is kept and then only if it represents more than 30 percent of the transitions out of the node. This leaves a graph of nodes with at most a single successor, though potentially multiple predecessors. Starting from the root of the graph, a traversal through the graph forms a chain of accesses that have high affinity, thus tend to follow each other. The number of accesses represented by the remaining edges provides a good estimate of the stability and regularity of the patterns. One way to take advantage of this is to prefetch the access represented by the next node in the chain. Since three graphs, one away, eight away, and sixteen away, are formed and pruned the resulting graphs represent the stability of patterns when looking ahead by distances of one, eight, and sixteen, loads respectively.

The analysis process above was performed twice, one in which all IPL accesses were collected (*all*), and a second in which only those IPL accesses that missed the cache were collected (*mo*).

Figure 6 presents the data affinity results for the Olden benchmarks, and Figure 7 presents the results for the CPU2000I benchmarks. The right hand

Benchmark	%IPL	%Miss	%None	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
crafty-1 all	3	3.8	71		8	18											
crafty-1 mo	3	3.8	96	1	2												
crafty-8 all	3	3.8	97	1													
crafty-8 mo	3	3.8	99														
crafty-16 all	3	3.8	99														
crafty-16 mo	3	3.8	99														
eon-1 all	4.5	6.3	73	3	18	4											
eon-1 mo	4.5	6.3	90	1	8												
eon-8 all	4.5	6.3	89	5	3			1									
eon-8 mo	4.5	6.3	99														
eon-16 all	4.5	6.3	94	2	1		2										
eon-16 mo	4.5	6.3	99														
vortex-1 all	6.5	5.9	90	1	6												
vortex-1 mo	6.5	5.9	83	1	15												
vortex-8 all	6.5	5.9	94	1	3												
vortex-8 mo	6.5	5.9	92	2	4												
vortex-16 all	6.5	5.9	96	1	1												
vortex-16 mo	6.5	5.9	99														
vpr-1 all	4.5	7.8	25		5	1									65		
vpr-1 mo	4.5	7.8	68	3	4	2	3		1	3	4	1			5		
vpr-8 all	4.5	7.8	39	1										58			
vpr-8 mo	4.5	7.8	77	2	1	1	2	4	4			5					
vpr-16 all	4.5	7.8	47	1										47		1	
vpr-16 mo	4.5	7.8	78	2	1	2	4	5			4						
mcf-1 all	31	7.9	65	3	8	16	1				3						
mcf-1 mo	31	7.9	37		1	2	2	2	3	4	43						
mcf-8 mo	31	7.9	65	1	2	2	4	23									
mcf-16 mo	31	7.9	71	1	2	4	19										
parser-1 mo	13.4	26.5	57	2	6	8	7	7	2	1	4						
twolf-1 all	11.5	29.8	42		4	5	13	29	4								
twolf-1 mo	11.5	29.8	39	1	7	12	36	1									
Benchmark	%IPL	%Miss	%None	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384

Figure 7: Affinity of data accesses in the CPU2000I benchmarks.

columns running from 2 to 16384 show the percentage of all IPL accesses that fall into a chain of that length. A 10 in the column labeled 16, shows that 10 percent of all IPL accesses fall in a chain of a length of at least 16 but less than 32. The columns labeled %IPL and %Miss are the percentage of all dynamic loads that are IPLs and the percentage of all misses due to IPLs for a 128k cache. These numbers are from the cache results presented in the previous section. The %None column is the percentage of the accesses that do not exhibit affinity. In health, looking one access ahead for all IPL accesses, about 79 percent of these accesses exhibit affinity, which can be found by taking the row sum of the chain length columns. Therefore about 79 percent of the time the next access is readily predictable. The benchmark treeadd shows another important point. Its accesses exhibit very high affinity, however these IPLs rarely ever miss the cache (only 0.3 percent) so the existence of affinity is somewhat irrelevant. For Olden, em3d is another noticeable benchmark showing strong affinity. In CPU2000I, mcf, parser, and twolf show some affinity in IPLs.

4.3 Phasing and Prefetching

To gain more insight into the nature and use of the affinity data collected the data were inspected for phasing behavior and used to guide prefetching. The phasing information helps determine if stronger relationships exist than might be obvious from a view of the entire execution. The use of the data to guide prefetching shows one potential application.

As previously mentioned, affinity analysis was performed on a mixed stream of EAs and the edge weights in the graph are from the entire execution of the program. Because of this, a node with two edges each containing half the weight could be the result of

Benchmark	%None	2	4	8	16	32	64	128	256
bisort	**70**	**4**	**11**	**7**	**2**	**1**			
bisort-1	77	3	9	7	1				
bisort-1	66	2	8	9	7	2	2		
bisort-1	62	1	6	7	7	7	1	1	3
bisort	**89**	**5**	**3**	**1**					
bisort-8	86	5	5	1					
bisort-8	83	4	4	4	2				
bisort-8	81	3	5	3			2	1	
bisort	**91**	**5**	**1**						
bisort-16	87	5	5	1					
bisort-16	86	4	4	2					
bisort-16	83	4	4			2	4		

Figure 8: IPL affinity in samples for bisort.

a rapid toggle between two successors or two steady sequences each executing for half the time. Thus steady sequences might be obscured by the aggregate statistics.

It is useful to determine if, in some cases, higher affinity exists than is obvious from the aggregate numbers. To do this, bisort and vortex were chosen for phased analysis since they exhibited poor overall affinity. Data was collected for samples of every 100 million instructions and then analyzed.

Figure 8 shows the phased affinity results for bisort, and Figure 9 shows the results for vortex. In each case the original, aggregate affinity is shown in the shaded rows and the results for each sample are shown below them. In almost all cases the affinity of the samples is more than that of the aggregate. This means that there is higher affinity from which to take advantage if a more dynamic scheme is used.

The information on data affinity can be fed back from an earlier part of execution to optimize a later part in the same run or from a previous run of an application to a subsequent run. We made a preliminary attempt to study the potential of using data affinity to guide prefetching. In these experiments, the affinity chains are saved during the first execution. These chains are analyzed as described in the previous section to identify the candidates for prefetching in the next execution. During a second run of the same benchmark, the affinity data from the first run is reloaded into the simulator and used to guide prefetching. This gives an indication on the potential of this correlation-based prefetching. The current heuristic to trigger prefetching is very simple. When an EA is seen that exists in a chain, a prefetch to the successor EA is performed if one exists. EAs in a chain have at most one successor. Nothing special is currently done to weed out useless and redundant prefetches, and the prefetch, from the viewpoint of the simulator, is a normal cache read. Both of these factors could result in excessive prefetching and cache pollution. However, the simple prefetching algorithm still shows that the affinity information can be useful.

Two sets of prefetch data are presented. Figure 10 shows the results of prefetching on IPL cache performance. IPLs are the sole focus of this prefetching. Each of the benchmarks em3d, health, mcf, and twolf each shows a sharp decrease in the number of IPLs that miss the cache.

5 Conclusion and Future Work

The various analyses have yielded valuable information about the characteristics of IPL accesses and help evaluate whether and how this information can be effectively used to guide new prefetching and data layout techniques.

With regard to the data cache, IPLs sometimes trigger more than their share of misses and are not satisfied as easily by larger cache sizes. Em3d, health, tsp, parser, and twolf exhibited this property. Benchmarks in both Olden and CPU2000I showed promising amounts of data reference affinity. On average, benchmarks from CPU2000I have less affinity than in Olden. However, there is enough affinity to warrant attempts to exploit it since even a simple prefetching scheme can improve cache results. This

Benchmark	%None	2	4	8
vortex	**83**	**1**	**15**	
vortex-1	86	2	10	
vortex-1	78	2	12	6
vortex-1	79	3	13	3
vortex-1	78	3	15	2
vortex-1	79	3	8	8
vortex-1	78	4	14	2
vortex	**92**	**2**	**4**	
vortex-8	96	2	1	
vortex-8	94		5	
vortex-8	92		6	
vortex-8	94	1	4	
vortex-8	92	1	6	
vortex-8	94	1	3	
vortex	**99**			
vortex-16	98			
vortex-16	98	1		
vortex-16	96	1	1	
vortex-16	96	1	2	
vortex-16	97	1	1	
vortex-16	96	1	1	

Figure 9: IPL affinity in samples for vortex.

is especially true when the added benefits of phasing is considered.

There is certainly much more to learn about data reference characteristics which requires different analysis techniques and a more robust infrastructure. One of the challenges is to have a fast mechanism in the presence of so much data to collect. Since future tests will include longer traces, the analysis will have to be faster and use less memory. This would include filtering out reference relationships that appear to be weak.

The phased analysis was performed to gain insight into benchmarks showing low affinity. The two phasing tests yielded interesting results worthy of further investigation. This should be done by intelligently choosing phase starting and ending points instead of simple sampling, and compiler static analysis may help provide such information. Once collected, this data could then be used to drive prefetching.

Prefetching itself should be extended to have a better heuristic so that prefetches are less likely to pollute the cache. If cache pollution and the amount of prefetches do not cause performance problems, then, in certain instances, multiple nodes could be prefetched when a node has a small number of important successors.

The affinity analysis should try to analyze and group traces based on individual instructions. We conjecture higher affinity for data referenced by particular loads. Finally, the analysis could be extended to look at more than just IPLs. This would allow potentially more benefits from the analysis techniques.

6 Acknowledgments

We would like to thank Rakesh Krishnaiyer, Oren Gershon, Dan Lavery, Wei Li, and David Sehr at Intel Corp. for providing assistance on various tools that we used.

References

[1] S. Srinivasan, R. Ju, A. Lebeck, and C. Wilkerson, "Locality vs.criticality," in *To appear in: Proceedings of the 28th International Symposium on Computer Architecture*, July 2001.

[2] M. P. Gerlek, E. Stoltz, and M. Wolfe, "Beyond induction variables: detecting and classifying sequences using a demand-driven ssa form," *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 1, pp. 85–122, 1995.

[3] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren, "Supporting dynamic data structures on distributed memory

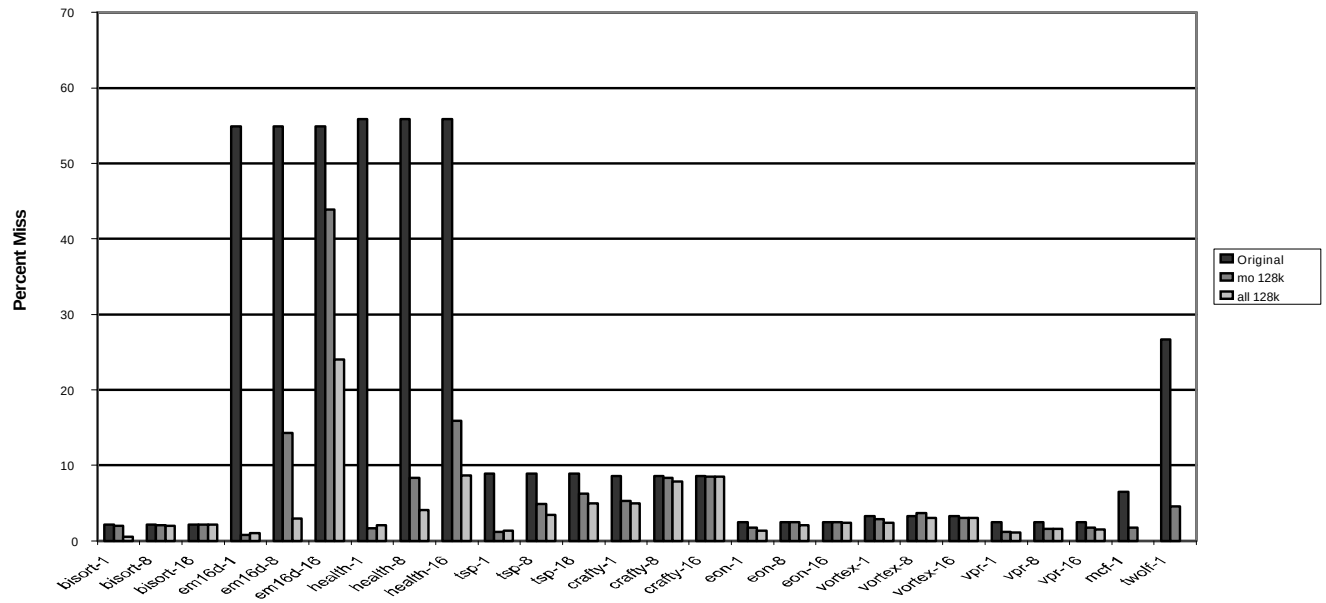

Figure 10: Cache performance of IPLs using IPL prefetching.

machines," *ACM Transactions on Programming Languages and Systems*, vol. 17, pp. 233–263, March 1995.

[4] G. Reinman and B. Calder, "Predictive techniques for aggressive load speculation," in *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, November 1998.

[5] T. L. Johnson and W. W. Hwu, "Run-time adaptive cache hierarchy management via reference analysis," in *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

[6] C. Luk and T. Mowry, "Compiler-based prefetching for recursive data structures," *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pp. 222–233, Oct 1 1996.

[7] C. Luk and T. Mowry, "Memory forwarding: enabling aggressive layout optimizations by guaranteeing the safety of data relocation," *Proceedings of the 26th International Symposium on Computer Architecture*, pp. 88–99, May 1 1999.

[8] A. Roth and G. Sohi, "Effective jump-pointer prefetching for linked data structures," in *Proceedings of the 26th International Symposium on Microarchitecture*, pp. 111–121, May 1 1999.

[9] A. Roth, A. Moshovos, and G. Sohi, "Dependence based prefetching for linked data structures," in *Proceedings of the 8th international conference on Architectural support for programming languages and operating systems*, pp. 115–126, Oct 2 1998.

[10] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *Proceedings of the 24th international symposium on Computer architecture*, pp. 252–263, 1997.

[11] B. Calder, C. Krintz, S. John, and T. Austin, "Cache-conscious data placement," in *Proceedings of the 8th International Conf. on Architectural Support for Prog. Lang. and Operating Systems*, pp. 139–149, April 1998.

[12] T. Chilimbi and J. Larus, "Using generational garbage collection to implement cache-conscious data placement," in *Proceedings of The International Symposium on Memory Management*, pp. 37–47, 1998.

[13] T. Chilimbi, M. Hill, and J. Larus, "Cache-conscious structure layout," in *Proceedings of the ACM SIGPLAN '99 conference on Programming language design and implementation*, pp. 1–12, 1999.

[14] T. Chilimbi, B. Davidson, and J. Larus, "Cache-conscious structure definition," in *Proceedings of the ACM SIGPLAN '99 conference on Programming language design and implementation*, pp. 13–24, 1999.