# Virtual Cache Line: A New Technique to Improve Cache Exploitation for Recursive Data Structures

Shai Rubin, David Bernstein, and Michael Rodeh

IBM Research Lab in Haifa and
Computer Science Department – Technion (Israel Institute of Technology)
{rubin,bernstein,rodeh}@haifa.vnet.ibm.com

**Abstract.** Recursive data structures (lists, trees, graphs, etc.) are used throughout scientific and commercial software. The common approach is to allocate storage to the individual nodes of such structures dynamically, maintaining the logical connection between them via pointers. Once such a data structure goes through a sequence of updates (inserts and deletes), it may get scattered all over memory yielding poor spatial locality, which in turn introduces many cache misses. In this paper we present the new concept of Virtual Cache Lines (VCLs). Basically, the mechanism keeps groups of consecutive nodes in close proximity–forming virtual cache lines–while allowing the groups to be stored arbitrarily far away from each other. Virtual cache lines increase the spatial locality of the given data structure resulting in better locality of references. Furthermore, since the spatial locality is improved, software prefetching becomes much more attractive. Indeed, we also present a software prefetching algorithm that can be used when dealing with VCLs, resulting in even higher data cache performance. Our results show that the average performance of linked list operations–like scan, insert, and delete–can be improved by more than 200% even in architectures that do not support prefetching. Moreover, when using prefetching, one can gain additional 100% improvement. We believe that given a program that manipulates certain recursive data structures, compilers will be able to generate VCL-based code. Until this vision becomes true, VCLs can be used to build more efficient user libraries, operating-systems, and applications programs.

## 1 Introduction

When dealing with recursive data structures the problem of high latencies in accessing memory is well recognized. Significant efforts have been directed in the past on reducing its harmful effects. Advanced memory designs have been developed [2] e.g., by way of cache memories and prefetch instructions, and offer partial remedy to this problem. Actually, Mowry [11] showed that many scientific programs spend more than half of their time waiting for data. In [7, 11] software methods and tools to overcome this problem for scientific code are proposed.

Memory latency impacts both the instruction stream and the data stream. The locality of the instruction stream may be improved by code reorganization [8,

13]. Also, code usually has natural locality by itself. Therefore, larger instruction caches reduce the instruction cache miss ratios considerably.

Data locality is more difficult to cope with as data can grow in size to magnitudes which do not fit into cache (or even into memory) for any practical cache size. Even worse, data may be scattered in memory in a rather random way unless measures are taken to cluster related pieces. A common approach, especially for scientific programs, is to reorganize the computation while leaving the data layout intact [9]. Unfortunately, this approach is only marginally applicable to scattered data since the machine stalls while the next piece of data to be processed is fetched. Prefetching offers only partial remedy – it is the data layout itself which should be optimized for better spatial locality.

Spatial locality can be achieved by storing neighboring nodes in close proximity. When seeking a solution to this problem, three measures have to be balanced: (a) The data structure operations have to be efficient. For example, if a linked list is stored in consecutive locations in memory, search is fast but insert and delete operations become inefficient. (b) Memory has to be utilized effectively. For example, by allowing gaps between nodes, update operations may become more efficient, at the expense of lower memory utilization. (c) The machine architecture should be exploited. For instance, in machines that support memory prefetch instructions, they should be used to reduce memory latency. Examples of such balance between these three measures are B-trees [1]: special data structures designed to cope with long disk latencies when using virtual memory. The technique presented in this paper takes the intuition from this basic data structure, however it deals with memory latencies rather than disk latencies.

### 1.1 Recursive Data Structures (RDS)

Recursive Data Structures (RDSs) are usually defined in terms of nodes and links connecting them. Only in rare cases do they specify the relative positioning of the nodes. It is this degree of freedom which we try to exploit.

Consider a program which manipulates an RDS. The nodes of the RDS are typically dynamically allocated on the run-time heap and are, in general, scattered in memory. Therefore, cache hit ratio is rather low, and frequent calls to the memory manager are time consuming. To improve, Luk and Mowry [10] have suggested to linearize the data, namely, to map heap-allocated nodes that are likely to be accessed closely in time into contiguous memory locations.

We extend this notion of data linearization by dynamically grouping nodes into Virtual Cache Lines (VCLs) – a software concept that is a generalization of the hardware-oriented structure of cache lines. This grouping have four major positive effects: (a) The number of cache misses decreases. (b) The number of calls to the memory manager is reduced. (c) Memory fragmentation improves. (d) Ability to use prefetching when dealing with RDS is much higher than before. One negative effect is that managing the VCLs is somewhat more complicated. The good news are that compared with the common implementation of the linked list the VCLs overall performance can be improved by 300%.

In order to confirm the fact that VCLs might be useful regardless the architecture one uses, we measured the performance for two common platforms. The first is a non-prefetching architecture of Intel's Pentium [6] and the second architecture is the IBM PowerPC [15] that supports prefetching instructions.

### 1.2 Compilers techniques for handling RDSs.

While improving spatial locality is a clear objective, finding ways to achieve this goal is quite a challenge:

1. The problem is global, namely, the entire program has to be taken into account. A data layout, which is good for one segment of the code, may be suboptimal for another portion of the program. This introduces heavy dependency on interprocedural analysis.
2. Discovering the data structure that a program uses is a very hard problem. Moreover, what we really need is to discover not only the nature of the data structure, but also the code segments which implement the data structure operations. For example, not only do we have to find that a program manipulates a binary tree, but we also have to identify the code sections that implement the insert and delete operations.

In view of recent research results in the area of shape analysis [14, 16] we do believe that the above mentioned difficulties may be circumvented for many programs. We hope that this paper will encourage shape-analysis researchers to focus not only on discovering the data structures layout the program manipulates, but also to find the places (in the program) where certain operations are performed. For example, in order to fully automate the process of replacing the user-defined data structures with more sophisticated structures (like ours), we need to know where in the program the user does an insert or a delete operation.

However, this paper takes a more pragmatic approach: the method is simply to implement data structures in a cache-aware way. The paper presents this new idea from two different aspects. First, it suggests a new software data layout technique that exploits a given memory hierarchy regardless the processor one uses. Second, it points to a more sophisticated design to highly optimize the original basic technique, by using prefetching instructions that exist in some of the more common architectures such as PowerPC [15]. Several compiler related applications might gain from using the proposed technique:

1. User libraries. Recently user libraries become part of the official C++ language [12]. Our technique for cache optimization can be easily integrated into specific structures (e.g. singly-linked lists, doubly-linked lists) of these libraries.
2. Memory allocators and garbage collectors. A lot of research work was conducted investigating the way to improve the cache conscious data placement of memory allocators. Additionally the importance of garbage collectors is increasing in the recent years as Java becomes more popular. Since lists are basic part of these mechanisms, it might be very useful to take into account

cache consideration when building these parts of the compiler. Actually first signs of such research can be found in [4, 5].

The rest of this paper concentrates on presenting our novel data layout technique and the exploitation of prefetching in the context of this data layout. It is our plan to carry this research all the way to automatic generation of efficient data layouts of recursive data structures by optimizing compilers.

### 1.3 Outline of the paper

Section 2 reviews the VCL model in more details and presents a linked list implementation that uses it. The section also presents an evaluation of the VCL technique when it was tested on Intel's Pentium. Section 3 discusses and presents a prefetching algorithm that further improves the cache behavior of the linked list data structure. This time the evaluation is done by using an existing prefetching architecture – the IBM PowerPC. This section also presents a comparison between the known Greedy-Prefetching [10] and our new proposed technique. Section 4 concludes the paper, presents future work, and introduces the new research opportunities the VCL idea opened for the compiler and program analysis researchers.

## 2 Virtual Cache Lines - A Cache-Aware Data Structure

This section presents the concepts behind the Virtual Cache Lines model. First, it discusses the relation between the configuration of the physical cache lines (currently only the $L1$ cache) on a given system and the virtual cache lines arrangement. The last sub-section presents performance evaluation of the proposed model implemented on the popular Intel's Pentium architecture.

### 2.1 Aggregating nodes of a linked list

Consider a linked list with nodes which span 8 bytes[1] and assume that the machine has cache lines of size 64 bytes. Let us start with a VCL size which is equal to that of a physical cache line. Therefore, each VCL can contain 8 or fewer nodes. If we break the linked list into sublists each of which contains 8 consecutive nodes (except possibly, for the last sublist), the cache miss ratio is reduced by up to a factor of 8 (Fig. 1). Moreover, prefetching becomes very attractive since in this data layout. It is rather easy to find the address of the next VCL prior to processing the current one, thereby leaving as many as 8 nodes to be visited and processed before accessing the next VCL.

[1] Throughout this paper we limit ourselves to relatively small nodes. The motivation behind this assumption is that large nodes can, in many cases, be split into two sections: (a) the key section which contains the node's identifier. (b) The data section. If we store the second part separately from the first, then the first becomes rather small. It is the first section which is visited more often. This method is visited more often.

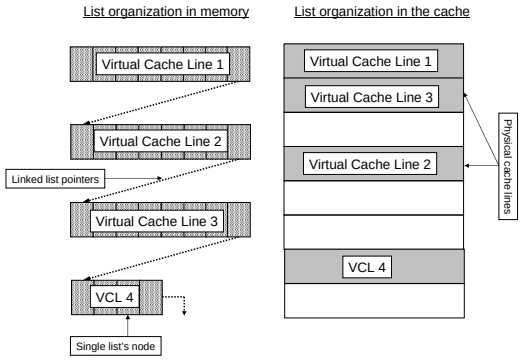



**Fig. 1.** Organization of nodes of a link list in the VCL mechanism. Each VCL contains 8 (or fewer) nodes. The figure presents the mapping of VCLs to physical cache lines. Such mapping may reduce the list cache miss-rate by a factor of 8. (The links between nodes inside the VCL are omitted for readability).

**Relaxing Memory Density.** While a dense layout such as the one shown in Fig. 1 is indeed very effective for traversal operations, it performs rather poorly when it comes to update operations such as insert and delete, since every such operation will invoke massive re-organization of the entire list. A way out of this trap is to trade memory density for performance, namely, allow gaps in the data layout. In the case of linked lists, let us allow the number of nodes per VCL to vary between $min$ and $max$ (Fig. 2), except maybe the last VCL. The next section shows that the mechanism to keep the number of nodes between these limits is fairly simple and efficient.

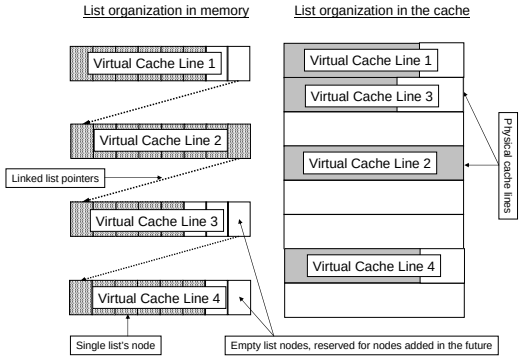



**Fig. 2.** Practical implementation of the VCL model. The number of nodes in each VCL varies between $min$ and $max$ (in this case $min = 5, max = 8$).

**Supporting Insert/Delete Operations Efficiently.** Two situations may arise: Either an update operation can be done within the $min/max$ imposed limits, or multiple VCLs have to be involved to resolve either a VCL *overflow* situation or VCL *underflow* situation. In case of a VCL overflow ($max$ is exceeded), either some nodes of the current VCL are spilled into the next VCL(s), or a new VCL (or maybe more) is allocated dynamically. Similarly, in cases of underflow – either we re-balance or we delete a VCL by invoking the memory manager (Fig. 3).

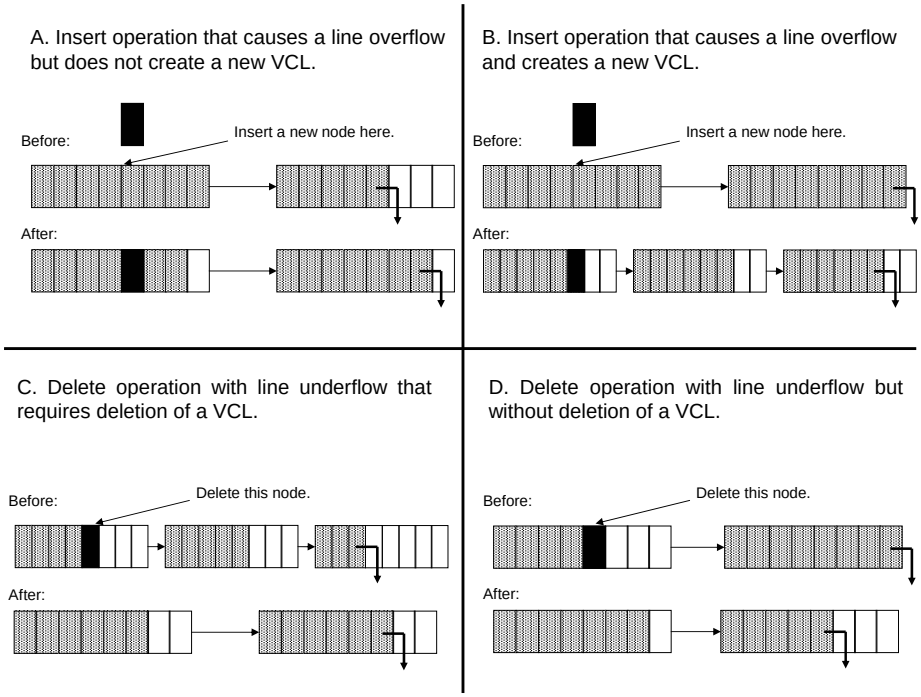



**Fig. 3.** Cases of insert/delete operations (in all cases $min = 5, max = 8$).

The insert[2] algorithm of a new node to the list is shown in Fig. 4. This procedure is being invoked each time a node should be inserted to a line and the line is full: the VCL already contains $max$ nodes. We will save from the reader the formal details proving inductively that after performing the code derived from Fig. 4 each VCL (except, maybe, the last one) contains between $min$ and $max$ nodes. However, intuitively it is easy to see that in all cases this inductive invariant holds. In Case 1 we found a VCL with at least one empty place so, shifting the nodes towards this empty place will enable us to insert the new node.

[2] The algorithm for the delete operation is completely analog. In order to increase readability we will save the details from the reader.

Since we did not add any new line it is obvious that each line contains between $min$ and $max$ nodes. In Case 2 we scan $min$ lines, each contains $max$ nodes. This means we have seen $min \times max$ nodes. Since we created $(max - min)$ new lines, it is clear that dividing the $min \times max$ original nodes to $max$ lines yields $min$ nodes in each line. Clearly in the first line we will have an extra node – the one we intended to insert. Case 3 is an end case. Since we allowed the last line to contain less then min nodes, it is possible to divide the original nodes in the desired fashion.

Scan the list from the overflowed line till one of the following is true:

Case 1: You reached a line that contains less then $max$ nodes.
Case 2: You scanned $min$ lines and all of them contain $max$ nodes.
Case 3: You reached the end of the list.

In Case 1: Reorganize the nodes in the lines you have just scanned (including the one that has less than $max$ nodes) such that each line will have at least min nodes

In Case 2: Create $(max - min)$ new VCLs. Reorganize the nodes in the lines you have just scanned and the new lines such that each line will have at least min nodes.

In Case 3: Reorganize the nodes in the lines you have just scan such that each line will have at least min nodes except the last one – if necessary create a new VCL.

**Fig. 4.** The insert algorithm in case of line overflow.

### 2.2 Performance Evaluation of the VCL Model in Non-Prefetching Architecture

To assess the performance of the VCL scheme, the efficiency of a simple scan on the list was evaluated. It is easy to realize the importance of a highly efficient scan operation on the list. Since the linked list is a very basic data structure, most of its common operations—like insert (after a specific item), delete (a specific item), and find—require at least partial scanning of the list. We have measured the performance of three implementations of linked lists:

1. Scattered lists – the standard implementation of linked lists, where each node is individually allocated by the memory manager. We were careful and constructed these lists by doing a large number of random insert and delete operations. In this way we promised that the nodes are truly scattered in memory – a typical situation to programs that manipulate large number of heap objects. This model is widely used in programs and user libraries [12].
2. Smart lists – a linked list that uses the VCL mechanism.

3. Compressed lists – a linked list that is mapped to an array. This can serve as a reference model since it offers optimal spatial locality. While this model is of no interest when it comes to update operations, it may be used as a lower bound when assessing performance of scan operations.

For convenience, we assumed that the linked lists to be studied are sorted by a specific field called *key*. The Scattered and Smart implementations present the same interface to the invoking applications. As a representative of non-prefetching architectures we chose the Intel Pentium. Moreover, testing the proposed model on such a popular processor is tempting. Intel's Pentium processor has an on-chip 8Kb data cache and a 256Kb L2 cache (instruction and data). Both caches have line size of 32 bytes. The operating system that we used was Windows 95. The Pentium architecture does not support prefetch instructions.

One last remark should be noted. Our main purpose is to show the potential improvement when using VCLs. We performed our experiments on lists that are constructed from nodes that span 8 bytes. It should be cleared that the improvement percentage (relatively to the Scattered list) one might gain when dealing with larger nodes is smaller. However, this improvement percentage is not the right way to light out results. The correct view is measuring the performance difference between the Smart list and the best performance one might achieve - the Compressed list. This difference is small regardless the nodes size.

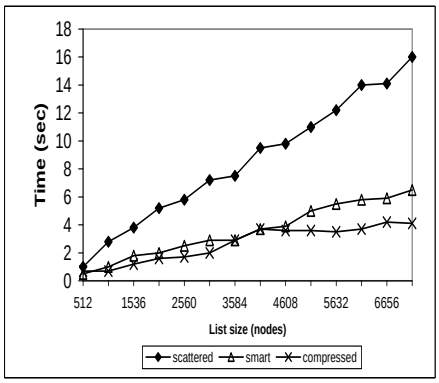
**Performance of List Scanning Operation.** To assess the performance of scanning the list we ran 15,000 experiments. In each such experiment we randomly selected a value to be searched for, and then performed the search on the three list implementations. Since the search is a linear process, searching is actually scanning of the list. The code that we ran in all three cases is shown in Fig. 5. In all cases lists with the same number of nodes are identical, meaning the contents of the nodes and the order between them is the same. The only difference between lists with the same length is their memory layout.

```
1. current = list→head_ address();
2. while (current) {
3.     if (current→key == key_to_find)
4.         return current; // End of the current search
5.     current = current→next;
6. }
```

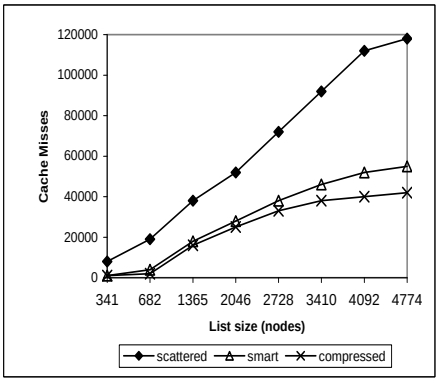
**Fig. 5.** The code for a single search operation. This code was performed for each of the lists' models.

We measured the time it took to complete the 15,000 searches in each of the implementation of the linked list. The test was repeated for several list sizes. We

used a VCL with $min = 8$ and $max = 10$. The size of a single node (including the next pointer) was 8 bytes. Therefore, each VCL consumed 80 bytes out of which at least 64 were used. On the Pentium processor these numbers mean that the "busy" part of each VCL consumes 2-3 physical cache lines.




(a) Time measured when performing 15,000 continuous searches on various sizes of lists using the three implementations of the linked list (Scattered, Compressed, and VCLs).




(b) Number of cache misses which occurred during a search operation using the three implementations of the linked list.

**Fig. 6.** List Scan Benchmark Performance.

Fig. 6(a) shows the performance results that we obtained. On average, Smart lists perform 2.48 times better than Scattered lists. For comparison reasons, observe that Scattered lists perform 3.4 times worse than Compressed lists. It is clear that when the list is smaller than the cache size, say 512 nodes which require 4Kb, the performance of the repeated searches through the list in the three implementations is similar.

Fig. 6(b) presents the total number of cache misses which occurred during the run. The cache misses where measured by using Vtune – commercial tool shipped by Intel [17]. Two main observations are worth mentioning:

**Observation 1:** There is a high correlation between run-time performance and cache misses in the three implementations of the linked list.

**Observation 2:** The performance (time and cache misses) of Smart lists is close to that of Compressed lists.

The last observation is somewhat surprising. We see that the Smart list achieves a very close performance to the compressed list. However, the smart list is still scattered around in memory. In this case the nodes of the Smart list are allocated in groups of tens, which means we have a large number of

independent groups (between 50 and 710 in the case of Fig 6(a)). It seems that in order to get an 'array' performance small groups are enough.

**Performance Evaluation of Insert/Delete Operations** Next, let us see how does insert and delete perform. The following experiment shows that maintaining linked lists is more efficient in the VCL model. In this experiment we compared the performance of Scattered and Smart lists ($min = 8$, $max = 10$). The two lists were built by performing the same sequence of insert and delete operations. Obviously, the number of inserts was bigger than the number of deletes. Fig. 7 presents the time it takes to build these lists with various sizes. The most important observation from this experiment is:

**Observation 3:** It is more efficient to maintain linked list in the VCL model than in the scattered implementation due to reduced number of memory manager invocations as well as lower number of cache misses (because of the implicit search operation).

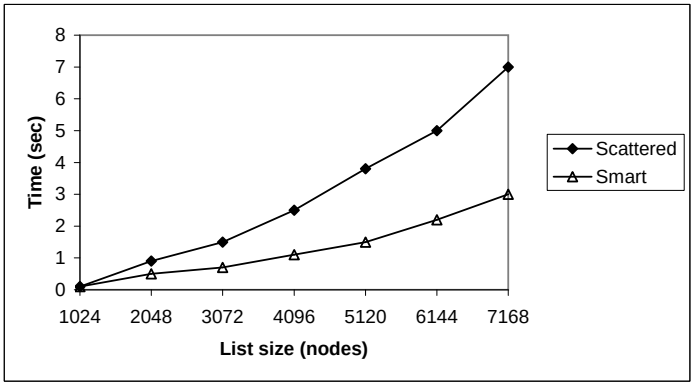



**Fig. 7.** Comparison of time it takes to maintain (i.e., *insert* & *delete* operations) Scattered and Smart lists.

One point should be cleared. The relation between $min$ and $max$ values will influence the results of this experiment. If the difference between $min$ and $max$ will be large, then the number of node reorganizations (as described in Fig. 3) will decrease and the performance gap between the smart and the scattered list will widen. We choose these specific values for $min$ and $max$ for two main reasons:

1. These values were discovered as the appropriate values when trying to optimize the scan operation.
2. We want to demonstrate that even when the gap between $min$ and $max$ is small, the improvement in the performance of the insert and the delete operation is still measurable.

# 3 Prefetching Techniques for the VCL Model

This section presents an algorithm that improves the performance of the VCL scheme by using various aspects of the software prefetch mechanism. As a representative for architectures that support prefetching we choose the IBM PowerPC processor (model 604e) that implements the 'dcbt' (Data Cache Byte Touch) non-interrupted prefetch instruction. An advantage of the PowerPC 604e processor is the fact that it has a prefetch queue that can tolerate up to four consecutive prefetch operations. The processor has an on-chip 32Kb data cache and a 512Kb L2 cache (instruction and data). Both caches have line size of 32 bytes. The operation system is AIX version 4.1.

## 3.1 The Prefetch Algorithm

The most popular prefetch technique in the case of recursive data structures is the Greedy-Prefetching [10] . The basic idea is very simple: when traversing the data structure, prefetch the nodes that are directly pointed by the current node. Fig. 8 illustrates Greedy-Prefetching when traversing a linked list.

```
1. current = list→head_address();
2. while (current != null) {
3.     PREFETCH(current→next);
4.     WORK(current)
5.     current=current→next
6. }
```

**Fig. 8.** Greedy-Prefetching on a linked list.

Our prefetch approach takes a similar direction. However, instead of performing inter-prefetching between the nodes of the data structure, we perform inter prefetching between the VCLs. The prefetch method uses the fact that the VCLs themselves can be arranged in a linked list data structure. Therefore, when traversing the current VCL, the algorithm performs a prefetch to the next VCL. VCL prefetching means not only bringing a specific physical cache line into the cache, but using an arbitrary number of bytes starting from a specific address. This can be achieved by the new prefetch queue implemented on the PowerPC processor model 604e. This model supports a prefetch queue of up to four prefetch instructions, where each instruction refers to a single physical cache line. Therefore, prefetching the next VCL can be achieved by several sequential prefetches that refer to a small linear part of memory (our VCL). Fig. 9 shows how to achieve a prefetch of a single VCL that is 128 bytes long using four 'regular' prefetch instructions.

```
1. address = start_of_VCL;
2. prefetch(address+0);
3. prefetch(address+32);
4. prefetch(address+64);
5. prefetch(address+96);
```

**Fig. 9.** Prefetching a VCL with length of 128 bytes by applying 4 regular prefetch instructions.

By prefetching VCLs instead of the original nodes of the data structure, the proposed approach should overcome the main disadvantage of Greedy-Prefetching. As mentioned above, the Greedy-Prefetching performs prefetch to the 'closest' nodes. In some cases this prefetch will be done too late, so it fails to hide the memory latency caused by a cache miss. Actually, our results (in the next section), show that the Greedy-Prefetching can degrade performance in cases where a small amount of work is done on each node. By prefetching VCLs instead of nodes, the algorithm increases the gap between the prefetch instruction and the data usage. For example, assume that a single VCL consists of 12 original nodes, we can prefetch 12 nodes ahead instead of one. Fig. 10 presents the code that traverses the list of VCLs and performs prefetch one VCL ahead.

```
1. current_VCL = list→first_VCL();
2. current = list→head_ address();
3.   while (current_VCL) {
4.      prefetch_next_VCL(current_VCL→next)
5.      num = current_VCL→num_of_items();
6.      for (i=0 ; i < num ; i++){
7.          WORK(current);
8.          current = current→next;
9.      }
10.     current_VCL = current_VCL→next;
11. }
```

**Fig. 10.** Prefetch of VCL instead of original nodes.

Determining the length of the VCL is done by taking into account several architectural features. The number of nodes in each VCL is determined as follows: the time it takes to perform lines 6-9 in Fig. 10 should be (approximately the same as) the time it takes to bring the following VCL into the cache. Hence, the optimal number of nodes in the VCL is directly influenced by an architectural feature: the time it takes to bring data into the cache. The second architectural component that we should consider when building the VCLs is the depth of the

prefetch queue. As mentioned, prefetch of a single VCL is performed by several sequential physical cache line prefetches (Fig. 9). Hence, the length of a VCL (in bytes) should be smaller than the maximal number of bytes that can be simultaneously prefetched by the prefetch queue.

### 3.2 Performance Evaluation of the VCL Model in Prefetching Architecture

To evaluate the suggested prefetching mechanism, we repeated the scan test on the three types of lists. We also compared our prefetch algorithm with the known Greedy-Prefetching (Fig. 8). We repeated the comparison in several cases where the program performed a different amount of work in each node when traversing the list. The amount of work is measured in clock cycles and determined in the function call WORK (See Fig. 8 and Fig. 10). Again, each node is 8 bytes long and since the 604e processor supports up to four sequential prefetches, we built the VCLs with a maximum length of 128 bytes. This way we could simulate a VCL prefetch using up to four prefetch instructions.

Fig. 11 presents the results we got. Some major points should be noted:

1. The Smart list with prefetching behaves better (approximately 25% improvement) than the Smart list without prefetching.
2. When the amount of work is small (less than 16 cycles per node) Greedy-Prefetching degrades performance.
3. The proposed prefetch method is better than Greedy-Prefetching in most cases and performs equally to it when we perform considerable amount of work on each node. This means that one can use the proposed method and achieve performance at least as good as the Greedy-Prefetching.

From these last points we can conclude the following main observation:

**Observation 4:** The combination of the VCL data layout and the prefetching technique improves the cache behavior of the Scattered linked list between 100% to 300%.

## 4 Future work and conclusions

Based on the lessons we learned from these preliminary results, we intend to continue our research in the following directions. First, we intend to find the precise connections between the length of the physical cache line and the virtual cache line. This connection is related to correctly choosing the values for the *min* and *max* parameters. Second, this research might be extended higher in the memory hierarchy and to more sophisticated data structures such as trees. However, we believe that the promising research direction is found in the shape analysis area. The opportunity to automatically transform a naive data structure and its implementation programmed by the user to more sophisticated structures and methods would enable us to insert the cache-aware data structures by the compiler without any assistance from the programmer.

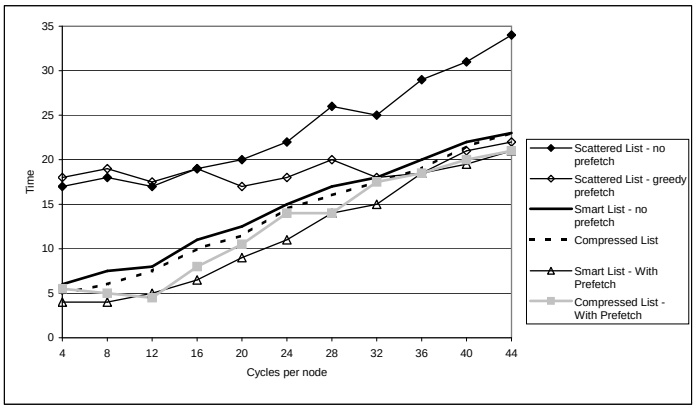



**Fig. 11.** Evaluation of the prefetching algorithm.

Programs that use Recursive Data Structures (RDSs) usually suffer from poor cache behavior due to the lack of locality in their data layout. One of the most common ways to deal with memory latency is to use prefetching. While this approach wins a success in numeric applications [3, 11]. in a pointer intensive environment the prefetch solution is more complex and not widely applicable [10]. This paper presents a new method to deal with this challenging problem. The Virtual Cache Line mechanism groups together sequential nodes of a linked list. This grouping concept results in a notably higher spatial locality of the linked list and therefore improves its cache performance. Our results show that not only it is possible to handle the linked list in the VCL manner, it is even more efficient to do so. The insert/delete operation becomes more efficient, but the main result is that even without prefetching scanning the list improves by 200%. Combining the VCL method with our new prefetching algorithm further improves the performance gain by an average factor of 300%. We believe that these preliminary results provide motivation to keep and develop prefetching methods and mainly to make compilers to use them automatically.

## References

1. Rudolf Bayer and Edward M. McCreight, *Organization and maintenance of large ordered indices*, Acta Informatica **1** (1972), 173–189.
2. James E. Bennett and Michael J. Flynn, *Reducing cache miss rates using prediction caches*, Tech. Report CSL-TR-96-707, Stanford University, 1996.
3. David Bernstein, Doren Cohen, Ari Freund, and Dror E. Maydan, *Compiler techniques for data fetching on the PowerPC*, Proceedings of PACT'95, Conference on Parallel Architectures and Compilation Techniques (Limassol, Cyprus), ACM Press, June 27–29, 1995, pp. 19–26.
4. B. Calder, K. Chandra, S. John, and T. Austin, *Cache-conscious data placement*, Proceedings of the Eighth International Conference on Architectural Support for

Programming Languages and Operating Systems (ASPLOS'VIII) (San Jose, CA), October 1998, pp. 139–149.
5. Trishul M. Chilimbi and James R. Larus, *Using generational garbage collection to implement cache-conscious data placement*, Proceedings of the International Symposium on Memory Management (ISMM-98), ACM SIGPLAN Notices, vol. 34, 3, 1999, pp. 37–48.
6. Intel Corp., *Pentium processor family developer manual*, 1995 ed., vol. 3, Intel, 1995.
7. Anoop Gupta, John L. Hennessy, Kourosh Gharachorloo, Todd C. Mowry, and Wolf-Dietrich Weber, *Comparative evaluation of latency reducing and tolerating techniques*, Proceedings of the 18th Annual International Symposium on Computer Architecture (Toronto, Canada), IEEE Computer Society Press, May 1991, pp. 254–263.
8. W. W. Hwu and P. P. Chang, *Achieving high instruction cache performance with an optimizing compiler*, Proceedings of the 16th Annual International Symposium on Computer Architecture (Jerusalem, Israel) (Michael Yoeli and Gabriel Silberman, eds.), IEEE Computer Society Press, June 1989, pp. 242–251.
9. Monica Lam, Edward E. Rothberg, and Michael E. Wolf, *The cache performance and optimizations of blocked algorithms*, Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'IV) (Santa Clara, California), April 1991, Stanford University.
10. Chi-Keung Luk and Todd C. Mowry, *Compiler-based prefetching for recursive data structures*, Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'VII) (Cambridge, MA), 1996, pp. 222–233.
11. Todd C. Mowry, *Tolerating latency through software-controlled data prefetching*, Ph.D. thesis, Stanford University, Computer Systems Laboratory, 1996.
12. David R. Musser and Atul Saini, *STL tutorial and reference guide*, Addison-Wesley, Reading, 1996.
13. Karl Pettis and Robert C. Hansen, *Profile guided code positioning*, Proceedings of PLDI'90 Conference on Programming Languages Design and Implementation, 1990, pp. 16–27.
14. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm, *Parametric shape analysis via 3-valued logic*, Proceedings of POPL'99 Conference on Principles of Programming Languages (San Antonio, TX), January 1999, pp. 105–118.
15. Tom Shanley, *PowerPC system architecture*, PC system architecture series, Addison-Wesley, Reading, MA, USA, 1995.
16. Marc Shapiro and Susan Horwitz, *Fast and accurate flow-insensitive points-to analysis*, Conference Record of POPL.97 Conference on Principles of Programming Languages (Paris, France), January 1997, pp. 1–14.
17. Ron van der Wal, *Programmer's toolchest: Source-code profilers for Win32*, Dr. Dobb's Journal of Software Tools **23** (1998), no. 3, 78, 80, 82–88.