INSTRUCTION OVERHEAD AND DATA LOCALITY EFFECTS IN SUPERSCALAR PROCESSORS

Murali Annavaram, Gary S. Tyson, Edward S. Davidson

Electrical Engineering and Computer Science Department
The University of Michigan, Ann Arbor
(annavara, tyson, davidson)@eecs.umich.edu

ABSTRACT

To reduce software development and maintenance costs, programmers are increasingly using object oriented programming languages, such as C++, and relying on highly flexible data structures, such as linked lists. Object oriented programming languages provide features that help manage complex software systems, but object oriented programs tend to suffer increased instruction counts due to generalized class implementations and many more calls to small functions.

The use of linked data structures increases programming flexibility by allowing easy addition and deletion of nodes, and by dynamically allocating memory to satisfy applications that use large memory space. However successive elements in linked data structures can potentially be allocated to noncontiguous memory locations, leading to poor spatial locality for list traversals, which may increase the number of cache misses and reduce performance.

This paper evaluates the impact of both the increased instruction overhead and poor spatial locality on superscalar processor performance as issue width increases. We show that underutilized resources of wide-issue processors can partially alleviate the impact of the instruction overhead. Poor locality on the other hand tends to cause more performance degradation as the processor issue width increases.

Furthermore we have improved the spatial locality of some programs that use linked data structures by using a vector representation to implement the list structures. Vectors exhibit better spatial locality during list traversals, but suffer from instruction overhead and memory copy overhead when nodes are added to and deleted from the structure. Even with this increased cost for additions and deletions in a vector, when list traversals dominate list modifications, the improved spatial locality significantly improves overall performance. By reducing cache misses, we achieved a 1000% overall speedup for *health* and 40% for *em3d*.

I. INTRODUCTION

One major consequence of increasingly complex software systems and their high development costs is the growing emphasis on code adaptability, interoperability and reusability. To meet these concerns, object oriented programming languages, such as C++, are increasingly being used in a wide range of heterogeneous environments and applications such as user-interfaces, operating systems and scientific computing. However, C++ programs tend to execute more instructions than C programs for a given task, because of generalized class implementations (e.g. templates) [CGZ94] and many more calls to small functions.

Moreover due to the increasing cost of software development, programmers optimize development time by using highly flexible data structures such as dynamically allocated linked data structures. However successive elements in linked data structures can potentially be allocated to noncontiguous memory locations, leading to poor spatial locality for list traversals. Given the increasing gap between the memory and processor core performance, increases in cache misses can severely degrade overall performance, particularly in wide-issue processors even with aggressive out-of-order engines.

In this paper we evaluate the impact of both the increased instruction overhead and poor spatial locality on the program performance, particularly in the context of today's trends toward wider issue superscalar processors. We show that although increased instruction overhead and poor locality can each degrade performance, the trend toward wider issue processors can partially alleviate the impact of the instruction overhead, but poor locality causes increased performance degradation in wider-issue processors.

To evaluate the effect of the underlying data structure representation on locality and hence performance, we changed the underlying data structure representation of a set of benchmarks from linked lists to vectors. The C++ Standard Template Library (STL) provides a convenient way of making this change, since vector and linked lists representations share a common set of STL interfaces for adding, deleting and accessing elements. Changing linked lists to vectors did indeed improve the spatial locality of list traversals, while using STL also preserved the programming flexibility provided by the original choice of linked data structures. In particular, implementing the *health* and *em3d* benchmarks in a programming style that maintains programming flexibility, while using underlying data structures that achieve good locality resulted in an overall speedup of 1000% for *health* and 40% for *em3d*.

Our results show that while software development efforts should consider the impact of both instruction overhead and poor locality, much larger performance gains can be achieved by carefully tuning programs for improved locality as the higher priority.

The rest of this paper is organized as follows. Section II

Bmark	Description(input)
Power	Solves a power system optimization problem. The main data structure is a multi-way tree with a linked list at each node of the tree. The data structure is built at the beginning of the program execution and is unmodified throughout the execution. The program iteratively computes the optimal cost of the power supply, traversing the data structure repeatedly.(10K nodes)
EM3D	Simulates the propagation of electro magnetic waves in a 3D object. The data structure used is a linked list. Each node in the linked list has a stored array of pointers that lead to its neighboring nodes. The execution phase traverses the linked list; at each node it accesses the nodes pointed to by the array.(2K nodes,1K iter)
MST	Minimum spanning tree of a graph. The data structure is a linked list with a hash table at each node that stores the computed minimum distance between the node and the neighbors reachable from this node. The execution phase traverses the list and iteratively computes the minimum spanning tree of the list of nodes.(1K nodes)
Health	Simulates a health care system. The data structure is a quadtree with three linked lists at each node of the tree: patients waiting for a doctor, being assessed for treatment and admitted into the hospital, respectively. During execution in each iteration a new patient is added to one of these lists, an existing patient is moved from one list to another, or deleted.(Tree depth is 5 levels,1000 iterations)
mlcache	A discrete event simulator for a cycle level cache simulation. For each address trace record, an event is scheduled into the event queue, which is sorted by the time stamp of that event's completion time. At each time step this queue is traversed and all the events that are scheduled for completion at that time stamp are processed.(100,000 address trace records)

Table 1: Benchmark Description

briefly describes the most relevant prior work done in analyzing the behavior of object oriented programs and in data reorganization to improve their spatial locality. Section III describes the benchmarks, simulation environment and performance analysis tools used in this study. Section IV compares the instruction counts of same benchmarks written in C and C++ and shows the impact of this difference on execution cycles. By analyzing the impact of the cache miss latencies on the overall performance we show how poor locality can cause more performance degradation than increased instruction counts. Section V shows how changing the linked list data structure to vectors improves the spatial locality of some applications, resulting in significant performance gains. We conclude and discuss future directions in Section VI.

II. RELATED WORK

Calder et. al. [CGZ94] studied the behavioral differences between C and C++ programs and observed that a well written C++ numerical library is as efficient as an optimized FORTRAN library. They noted that the C++ programs have higher instruction cache miss rates than the C programs. Unlike their study which used different sets of C and C++ programs for comparison, this paper compares C and C++ programs that execute the same task.

One approach to improving spatial locality of a program is to make algorithmic changes to the program such that the reference behavior better suits the way the data is laid out in the memory [GJG88, WL91, CMT94]. Another approach is to change the data layout to match the program's reference behavior [SD94, CKJA98, CDL99]. Recently Ding and Kennedy [DK99] proposed a hybrid approach of reorganizing both computation and data at run time to improve cache performance.

Chilimbi et al. [CDL99] proposed semi automatic tools *ccmorph* and *ccmalloc* that reorganize the data layout of tree structures so as to improve the cache performance. These tools needed programmer-provided hints to pack two closely accessed data elements into one cache block.

Shih and Davidson [SD94] proposed array grouping to improve the communication efficiency and cache utilization of parallel programs with irregular access patterns. Their method interleaves array elements from different arrays that are accessed similarly, and closely in time, to from a new array which will exhibit better spatial locality.

Ding and Kennedy [DK99] proposed locality grouping where the access sequence is reordered so that accesses to an object are clustered to improve the temporal reuse. They also suggested dynamic data packing to place objects that are accessed closely in time into the same cache line thereby improving the spatial locality of the program. Their technique relies on user directives to decide on the profitability of packing and locality grouping.

III. SIMULATION ENVIRONMENT AND BENCHMARKS

We selected the benchmarks where the main data structure used is a linked list. Hence we used only 4 of the 10 benchmarks from the *Olden* benchmark suite [RCRH95] that primarily use linked lists, as well as the *mlcache* [TRTD98] cache simulator. *Olden* benchmarks have been used previously to study the behavior of linked data structures [RS99, CDL99]. Table 1 briefly describes the benchmarks and the input sizes used in our simulations. All these benchmarks are written in C; we modified each of them into a C++ program that use STL calls to create, access and modify the linked data structures without making any algorithmic changes. These benchmarks were compiled on a Dec Alpha by the egcs-2.91.66 C++ compiler with -O2 optimizations. We used the SimpleScalar [BA97] timing simulator to simulate three different machine configurations: *old, current* and *future*. The microarchitectural parameter values used in these configurations are shown in the Table 2.

IV. COMPARISON BETWEEN C AND C++ PROGRAMS

This section presents a comparative study of the performance of C and C++ programs on wide-issue processors. We also analyze the effects of cache miss latencies on overall performance.

A Instruction overhead

	old	current	future
Issue Width	2	4	8
IFQ/LSQ Size	4	16	64
Reservation Units	16	64	256
L1 I & D cache	8KB	16KB	256KB
L1 Assoc:linesize	2way:32byte	2way:32byte	4way:64byte
Unified L2 cache	1MB	1MB	2MB
L2 Assoc:linesize	4way:64byte	4way:64byte	4way:64byte
L1 hit cycles	1	1	1
L2 hit cycles	8	16	40
Mem access cycles	16	40	100
Branch Predictor	2bit,512-entry	2-level	Perfect

Table 2: Microarchitectural parameter values

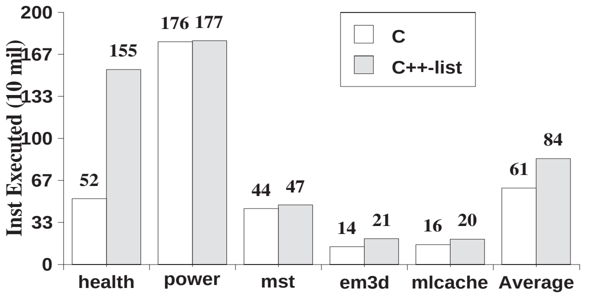

Figure 1: Instructions executed in the C and C++ versions of the benchmarks.

In our implementation although the C and C++ programs are algorithmically similar they differ in the methods used for creating, traversing and modifying the linked lists. C++ uses STL interfaces for this purpose, while C programs implement these functions as part of the user code. Since the C and C++ programs differ only in the list accessing routines, the C++ overhead will be prominent in those programs where execution is dominated by list traversals and modifications.

Figure 1 shows the number of instructions executed in the C and C++ implementations of each benchmark. Of the benchmarks used in this study, *power* has negligible C++ instruction overhead because 96% of the execution time (computed using *gprof* profiler on Dec Alpha) is spent in compute intensive routines (e.g. *sqrt, div*) which are independent of the programming language used. Only a very small fraction of the time is spent in traversing and modifying the linked data structures. On the other hand, C++ instruction overhead is significant in *health* because it extensively modifies and traverses linked lists while doing very little computation on each node. The C++ implementation of *em3d, mst* and *mlcache* have noticeable, but more modest instruction overhead. On average over these benchmarks the C++ implementation executes 38% more instructions than the C implementation.

B Execution cycles

In this section we analyze the impact of the C++ instruction overhead on the execution cycles required for program completion. In order to separate the effects of the instruction overhead from the cache effects which are discussed later in section C we use a perfect cache model in this section. In the perfect cache model both cache hits and cache misses take only one cycle to complete.

Figure 2 shows the execution cycles needed for completing the C and C++ benchmarks using perfect cache. As expected, in the absence of cache miss latencies, execution cycles decreased across all the benchmarks as the machine configuration became more aggressive (e.g. more functional units, better branch predictors). Furthermore for each configuration the increased instruction count of the C++ programs resulted in increased execution cycles. However, it is important to note that the 38% average instruction overhead of the C++ programs resulted in only a 29% increase in execution cycles on the *old* configuration, 22% on the *current* configuration, and only a 19% increase on the *future* configuration. The increased resources and lower average resource utilizations of the wider issue processors can apparently accommodate the increased instruction counts of object oriented programs, not only with fewer added execution cycles but with fewer added cycles relative to the reduced C program achieved by the wider-issue processor. Hence more aggressive machine configurations tend to reduce the impact of the increased instruction counts on overall execution cycles.

C Cache effects

This section analyzes the performance effects of replacing the perfect cache with a realistic multi-level memory hierarchy. For these simulations we used a two level cache hierarchy as specified in Table 2. Figure 3 shows the execution cycles required for completing each of the benchmarks in each of the three configurations. Each bar in the graph is split into two components. The lower component shows the execution cycles of the benchmarks using perfect cache. The upper component, labeled as *mem-overhead* in the graphs, of each bar shows the extra cycles spent by the processors due to cache miss latencies. The *mem-overhead* is the difference between the execution cycles when running a benchmark on two machine configurations that differ only in the cache miss latency: One configuration uses a multi-cycle data cache miss latency and other uses a perfect cache (as used in Section B). The third bar of each group in Figure 3 is discussed in Section V.

The first two bars in the graph for *power* show that in both the C and C++ implementations, the memory access overhead is insignificant, indicating that this program rarely suffers from cache miss penalties. In *mlcache* the memory access overhead is reduced as we go from the *old* to the *future* configuration, indicating that the increased cache sizes reduced the memory access overhead by reducing the cache misses. However, each of the other programs (C and C++ implementations of *health, em3d* and *mst*) show an increase in the *mem-overhead* component as a percentage of total run time as the configuration changes from *old* to *current* to *future.* Even the increased cache sizes of *future* compared to *current* did not help in alleviating the memory access bottleneck of these benchmarks.

D Discussion on *Health* benchmark

Surprisingly in *health* the C++ implementation has less memory access overhead than the C implementation. The reasons for this anomalous behavior are as follows. In every iteration of *health* a patient record is moved among three

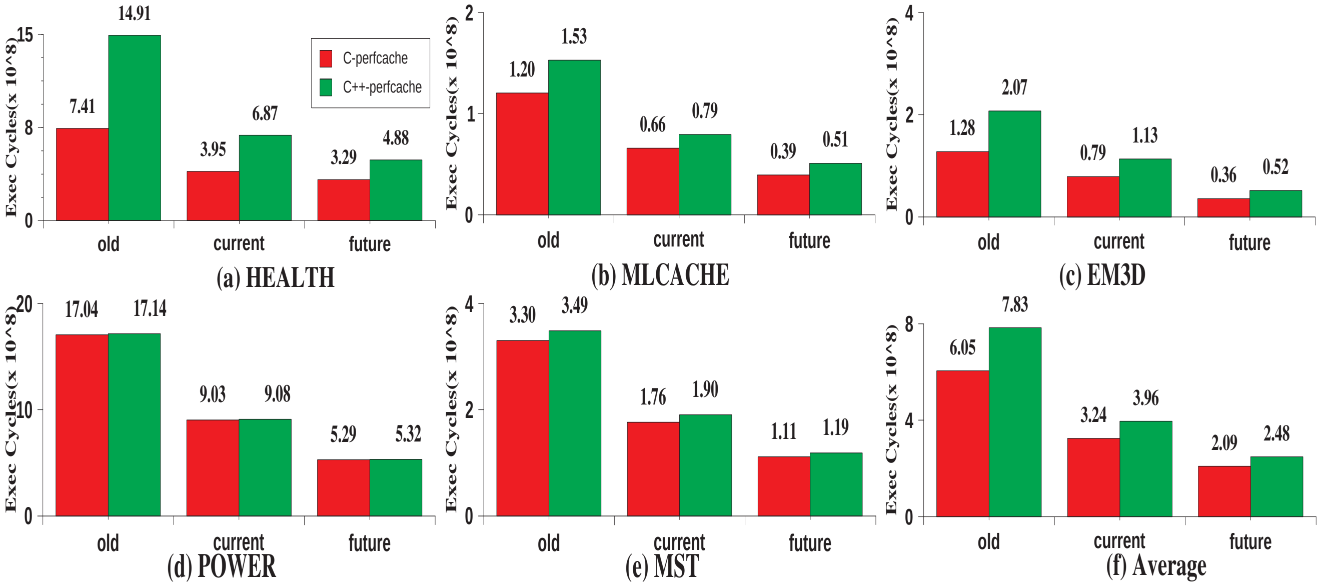

Figure 2: Execution Cycles needed for running the original C program and the C++ STL list implementations with perfect data cache

different lists – waiting list, assessed list and admitted list. In the C program each patient record is created only once and by manipulating the pointers, patient records are moved among these three lists. Due to the pointer manipulations at every iteration, after some number of iterations each of these lists becomes a sequence of widely distributed patient records with very little spatial locality for list traversals. In the C++ STL list implementation when a patient record is moved between lists, the library function *push_back()*, instead of manipulating the pointers, allocates memory for a patient record on the destination list and copies the patient data from source list. Although there is an overhead associated with this memory allocation and copying of record data, it does improve the spatial locality of list traversals as the nodes in the list are likely to be allocated nearby. In the *old* configuration, the overhead of this allocation outweighs the advantage of the improved spatial locality, but in the *future* configuration the effect of instruction overhead is reduced, and improved locality outweighs it.

E Summary

These results show that while both increased instruction overhead and poor locality can degrade the performance, the trend toward wider issue processors can partially alleviate the impact of the instruction overhead, but poor locality causes increased degradation in relative performance in wider issue processors.

V. VECTOR IMPLEMENTATION OF LINKED LISTS

We claimed in Section I that non-contiguously placed dynamic data structures tend to reduce spatial locality and degrade performance. The benchmarks used in this paper extensively use dynamically allocated linked data structures; *health*, *em3d* and *mst*, in particular, suffer from significant memory overhead while accessing these linked data structures. Since linked data structures make addition and deletion of nodes easy, restricting their use to improve performance may not be an acceptable solution from a software development perspective.

Hence we are constrained to evaluate the claim that non-contiguous data structures reduce spatial locality and degrade performance, by implementing these benchmarks in some programming style that preserves good locality without sacrificing the programming flexibility of linked data structures. Since both vector and linked list representations share the same STL interfaces for adding, deleting and accessing nodes, the list implementation of the benchmarks can be replaced by a vector implementation with minimal code modification. As we will see, vector structures do exhibit better spatial locality during list traversals, but they suffer from both instruction overhead and memory copy overhead when nodes are added and deleted from the structure.

The *Olden* benchmarks used in this study do much more traversal than modification of the linked lists and hence suffer from poor locality during list traversals. But these applications are not examples of poor programming techniques; the programmer preferred to use the linked lists to make the occasional addition and deletion of nodes easy. This choice in effect optimized development time at the expense of increased execution cycles. These benchmarks are also attractive for use in this study, since they have been used in several previous studies [RS99, CDL99] aimed at analyzing and improving the memory performance of linked data structure applications.

In Figure 3 the third bar (*c++-vect*) of each group shows the execution cycles needed by the vector implementation of the benchmarks. The vector implementation greatly reduces the memory access overhead in *health and em3d* relative to the list implementation, but *mst* does not benefit from the vector implementation. Further analysis showed that the original *mst* program uses an optimized user defined memory allocation routine that allocates the memory for all the

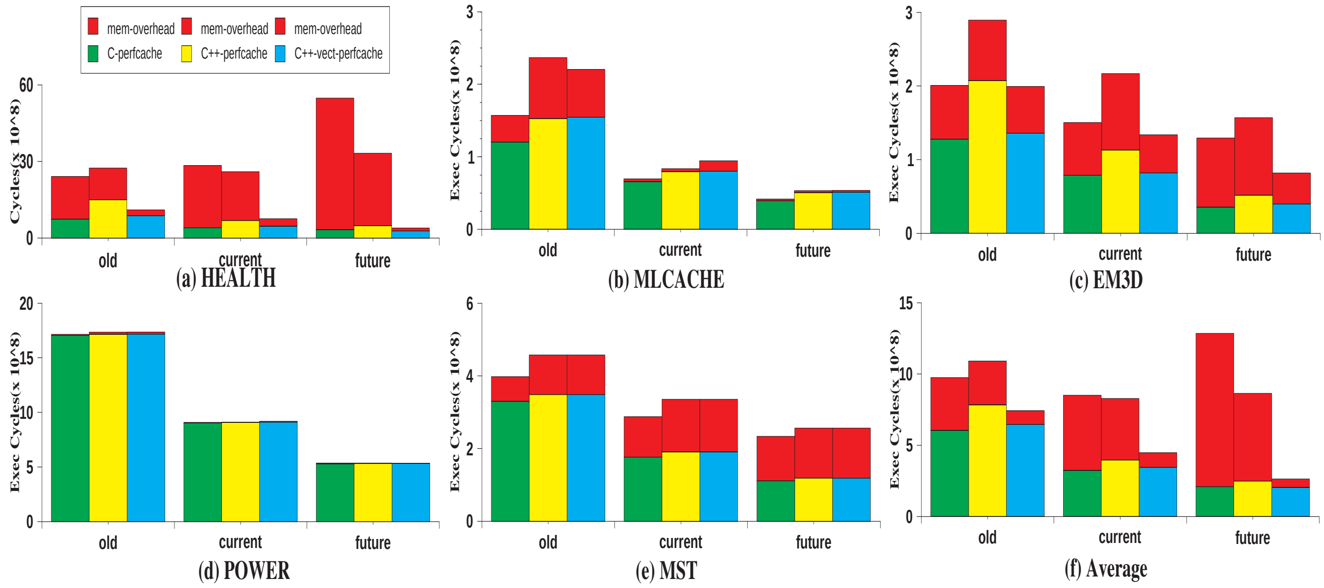

Figure 3: Execution Cycles needed for running the original C program and the C++ STL list implementations with a regular data cache

nodes in the linked list in one contiguous memory space. The programmer used knowledge of the program access patterns to carefully fine tune the linked list modification and access routines to improve the performance. In our STL implementation we used the default memory allocation methods which are not optimized specifically for the access patterns of this program.

Although the average computation cycles for the vector implementation are similar to the original C program the average memory access overhead is less in the vector implementation than the C implementation. This reorganization of data into vectors speeds up *health* by 1000%, *em3d* by 40% over the original program on the *future* configuration.

A Data cache performance

To further explain the gap in performance of vector and list implementations, we analyzed the performance of the data cache in terms of the number of cache accesses and cache misses. The graph on the left in Figure 4 shows the data cache accesses for the three implementations of each benchmark on the *future* configuration; the graph on right shows the data cache misses. Since the number of cache accesses in a program are typically proportional to the number of instructions executed, due to the extra instructions in the C++ programs the number of cache accesses in C++ are also more than C programs. The memory access overhead, however, is more dependent on the number of cache misses than the number of the cache accesses.

As shown in the cache miss graph, the improved spatial locality of the C++ vector implementation results in fewer cache misses than the C and C++ list implementations, except for *mst*. This improvement leads in turn to improved overall performance as observed in the execution cycle graphs.

B Summary

The C++ STL list implementation suffers from larger instruction counts compared to the C programs. Hence the execution cycles of the STL list implementation are higher than the original C program, except in *health*. Although STL vector implementation also suffers from increased instruction counts, whenever the vector implementation increases spatial locality the extra instruction overhead is offset by the improved memory system performance as is the case in *health* and *em3d*. But in *mlcache* and *mst*, the execution cycles of the C++ vector implementation are higher than the C program because the vector implementation did not improve the locality. These results show that while the software development process should be concerned with both instruction overhead and poor locality, much greater performance gains can be achieved by carefully tuning the program for improved locality.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we evaluated the performance impact of the increased instruction overhead of C++ programs when executed on wide issue superscalar processors. We also evaluated the effect of poor spatial locality of linked list traversals when the nodes in the list are non-contiguously allocated. Our results showed that the underutilized resources of wider issue processors can partially alleviate the impact of the instruction overhead. In our benchmarks a 38% instruction overhead of C++ programs resulted in only a 29% increase in execution cycles in a 2-wide superscalar processor configuration, 22% in a 4-wide configuration, and 19% increase in an 8-wide configuration.

While wider issue can better tolerate instruction overhead, the poor locality of non-contiguously allocated linked data structures can more seriously degrade the performance in wider issue processors. But linked data structures provide programming flexibility by allowing easy addition and deletion of nodes. Thus restricting their use so as to improve the performance may not be an acceptable solution from a software development perspective. By using C++ STL,

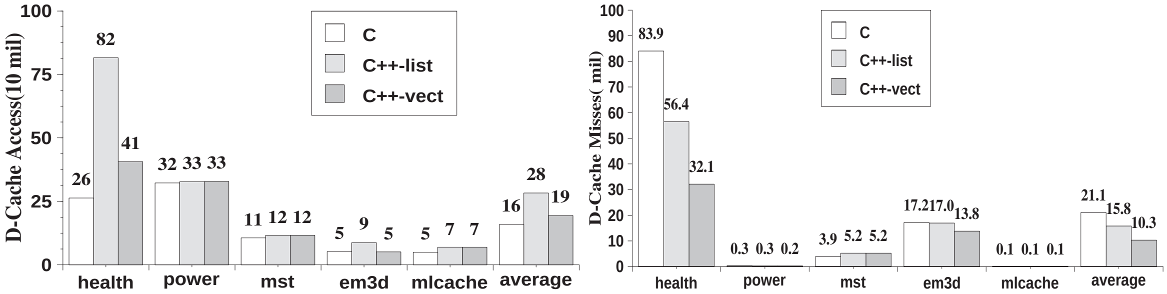

Figure 4: Data cache references and misses of the original C program, STL list, and STL vector implementations on the *future* configuration

which provides identical interfaces for adding, deleting and accessing the nodes both in vector and linked list representations, we changed the underlying data structure representation of a set of benchmarks from linked lists to vectors. While changing linked lists to vectors improved the spatial locality of list traversals, continuing the use of C++ STL also preserved the programming flexibility of using linked data structures. By using the C++ STL vector implementation that improved the locality while preserving the programming flexibility we achieved an overall speedup of 10X for *health*, 40% for *em3d*.

Although in our current work we linked the appropriate vector and list libraries manually, the appropriate linkage should be done automatically by the linker at compile time. One of the advantage of STL is that it provides standard interfaces across alternative data structure implementations which greatly simplifies the job of linking with the appropriate interface. One way to automate this process in our current implementation is to give hints to the compiler about the usage patterns of the linked lists. In this paper we showed that benchmarks are better implemented with vectors if they have a low frequency of adding and deleting linked list nodes relative to the frequency of list traversals. Using profile directed feedback, the same program can be linked with different implementations provided that they share common interface functions. Library writers can provide multiple specialized implementations that share one standard interface and the appropriate implementation can be selected and linked at run time by using the hints provided by the programmer or from the profile information. The programmer can thus be relieved of the burden of choosing which implementation is most efficient.

VII. ACKNOWLEDGMENT

The simulation facility was provided though an Intel Technology for Education 2000 grant.

VIII. REFERENCES

[BA97] D. Burger and T. M. Austin. The simplescalar tool set. Technical report, University of Wisconsin-Madison, Computer ScienceDepartment Technical Report #1342, June 1997.

[CDL99] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure layout. In *PLDI*, pages 1–12, 1999.

[CGZ94] B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between c and c++ programs. *Journal of Programming Languages*, 2(4), 1994.

[CKJA98] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *ASPLOS-VIII*, pages 40–52, 1998.

[CMT94] S. Carr, K. S. McKinley, and C. Tseng. Compiler optimizations for improving data locality. In *ASPLOS-VI*, pages 252–262, October 1994.

[DK99] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *PLDI*, pages 229–241, 1999.

[GJG88] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. In *Journal of Parallel and Distributed Computing*, pages 587–616, 1988.

[RCRH95] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. In *ACM transactions on Programming Languages and Systems*, March 1995.

[RS99] A. Roth and G. S. Sohi. Effective jump pointer prefetching for linked data structures. In *ISCA-99*, pages 111–121, June 1999.

[SD94] T. Shih and E. S. Davidson. Grouping array layouts to reduce communication and improve locality of parallel program. In *Proceedings of the 1994 International Conference on Parallel and Distributed Systems*, pages 558–566, December 1994.

[TRTD98] E. S. Tam, J. A. Rivers, G. S. Tyson, and E. S. Davidson. mlcache: A flexible multi-lateral cache simulator. Technical report, University of Michigan, Electrical Engineering and Computer Science Department Report, CSE-TR-363-98, May 1998.

[WL91] M. E. Wolf and M. S. Lam. A data locality optimization algorithm. In *PLDI*, pages 30–44, 1991.