Memory Access Analysis and Optimization Approaches On Splay Trees

Wei Jiang Chen Ding Roland Cheng
Computer Science Department
University of Rochester
Rochester, New York
{wjiang,cding,rcheng}@cs.rochester.edu

ABSTRACT

Splay trees, a type of self-adjusting search tree, are introduced and analyzed. Since they have been widely used in search problems, any performance improvements will yield great benefits. First, the paper introduces some background about splay trees and memory hierarchies. Then, it presents two heuristic algorithms, based on research in reference affinity and data regrouping. These algorithms have good locality, reduce memory transfers, and decrease cache miss rates on different architectures. Finally, the paper evaluates the performance gain of these algorithms on random inputs and Spec2K benchmark programs.

Keywords

splay tree, memory hierarchy, structure splitting, reference affinity, cache-oblivious algorithm, Van Emde Boas layout

1. INTRODUCTION

Splay trees are a self-adjusting form of binary search trees. In an amortized sense (i.e. the time per operation averaged over a worst-case sequence of operations), splay trees are not much worse than dynamic balanced trees [3] and static optimal trees [14]. They use a heuristic restructuring called splaying to move a specified node to the root of the tree via a sequence of rotations along the path from that node to the root. Thus, future calls to this node will be accessed faster. Splay trees even enjoy a constant operation time when the access pattern is uniform and has good locality [2].

The key to a splay tree is, of course, the restructuring splay heuristic. Specifically, when a node x is searched, it repeats the following splay step until x is the root of the tree [2]:

- Case 1 (zig): if $p(x)$, the parent of x, is the root, rotate the edge joining x with $p(x)$.
- Case 2 (zig-zig): If $p(x)$ is not the root, and x and $p(x)$ are both left or both right children, rotate the edge joining $p(x)$ with its grandparent $g(x)$ and then rotate the edge joining x with $p(x)$.
- Case 3(zig-zag): If $p(x)$ is not the root and x is a left child and $p(x)$ a right child, or vice versa, rotate the edge joining x with $p(x)$ and then rotate the edge joining x with the new $p(x)$ [2].

Figure 1 demonstrates the splay step through examples. Observe the move-to-root heuristic, which moves the target item to the root of the tree for each search operation. Although the cost is high for an individual operation, a benefit is the rough halving of the depth along the access path.

Aside from being easy to program and efficient in practice, splay trees also use less space because no balance information is stored. However, its disadvantages are its requiring adjustments at every tree search, resulting in potentially expensive individual operations. Despite the possible $O(N)$ time required in the worst case of an individual operation, splay trees have $O(\log N)$ amortized time cost, proved by Sleator & Tarjan [2].

A splay tree has two variants. The algorithm introduced above applies to a bottom-up splay tree. The second kind of splay tree is a top-down splay tree. It searches down from the root, looking for x two nodes at a time, breaking links along the access path, and adding each detached subtree to the bottom right of the left tree or the bottom left of the right tree. Figure 2 illustrates this process. Finally, an assembling step is needed to finish the work. It moves the accessed node x to the root and sets the left and right children of x to the L and R in figure 2.

The goal of our work is to reduce memory transfer using reference affinity analysis and data regrouping techniques, and to analyze the memory access pattern of a splay tree. In our experiments, the two kinds of splay tress are compared. Also, two heuristic algorithms are designed to improve the locality of the splay tree algorithm. They are evaluated with randomly generated input and Spec2K benchmarks.

2. MEMORY OPTIMIZATION TECHNIQUES

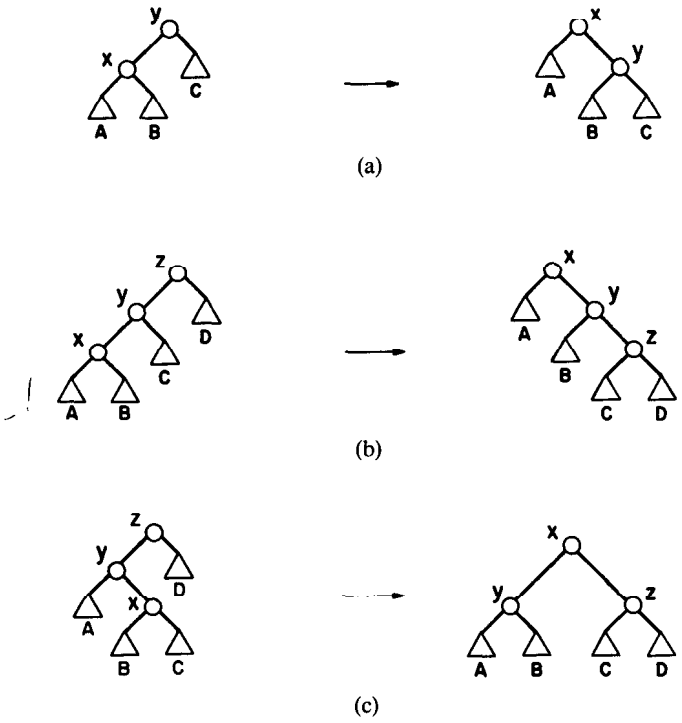


Figure 1: Bottom-up splaying steps. The node accessed is x. (a) Zig: single rotation. (b) Zig-zig: two single rotation.(c) Zig-zag: double rotation.

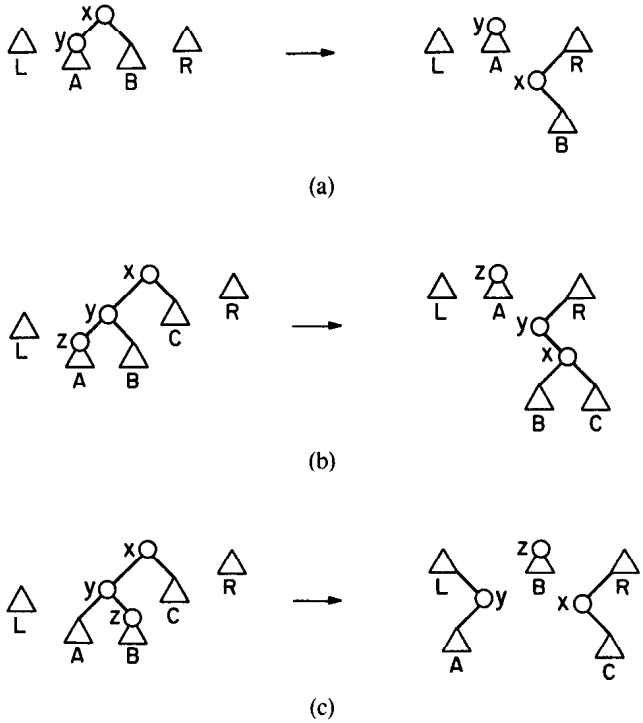


Figure 2: Top-down splaying step. The node accessed is x, Initial left tree is L, right tree is R. (a) Zig: node y contains the accessed item. (b)Zig-zig.(c) Zig-zag.

```
struct N {
    int val;
    struct N* left;
    struct N* right;
};
```

(a) before splitting

```
struct N_fragm0 {
    int val;
    unsigned int left;
};
struct N_fragm1 {
    unsigned int right;
};
struct  N_fragm0 f0_store[RESERVED];
struct  N_fragm1 f1_store[RESERVED];
```

(b) after splitting

Figure 3: structure splitting example

Because CPU speeds grow faster than memory speeds, modern computers use a memory hierarchy that includes registers, cache, disks and even networks. Even for a program with a theoretically good time bound, we may not get the ideal result if we ignore the impact of memory transfers. Consider the simple case of matrix multiplication: simply interchanging the loop levels will dramatically improve the performance without affecting time complexity.

Spatial locality is one of the most important factors to be considered when programming. Data locality is easier to maintain in problems having regular or static data. But for dynamic data, such as occurring in splay trees, maintaining locality requires more effort. In dynamic problems, the design of data structure is more crucial to moving data efficiently.

Recent research has revealed a possibility that some algorithms may work well at all levels of the memory hierarchy. We introduce these so called cache-oblivious algorithms in the following sections.

2.1 Structure Splitting

Current computers usually use cache blocks of at least 64 bytes, making data grouping very important. Before grouping data, we must first define a relation. Zhong, et al [10] defined a relation called reference affinity, which measures how close a group of data is accessed together within an execution. They measure togetherness with a *stack distance*, which is defined as the amount of distinct data accessed between two memory references in an execution trace. Note that stack distance is bounded even for long running programs.

Also, previous work by Ding & Zhong [11] has analyzed *reuse signatures*, histograms of the reuse distance of all program accesses. The reuse signature has a consistent pattern across data in complex programs. This suggests that we can analyze the affinity of data by looking at its reuse signatures from training executions. *K-distance* analysis is a simplified method of measuring reference affinity. The parameter k means elements in the same group are almost always used within a distance of k data elements. So by using k-analysis, we can get hierarchical affinity relations among data arrays and structures fields.

After getting the reference affinity of structure fields, we can apply structure splitting to increase locality. Structure splitting is illustrated in figure 3 above.

This splitting can increase cache utilization, such as after accessing a node x, when one of $left(x)$ or $right(x)$ is accessed. Previously, in the original structure, if we accessed node x, then the whole structure would be loaded, even though $right(x)$ may not be needed in subsequent steps. However, after structure splitting, more nodes can be loaded in one cache-memory transfer; thus decreasing memory-cache transfer. If $right(x)$ is accessed, an additional load appears to be necessary to load it into cache. But in fact, we see later that we more frequently access the right children of other nodes, which may be preloaded in this transfer. So this load is often useful.

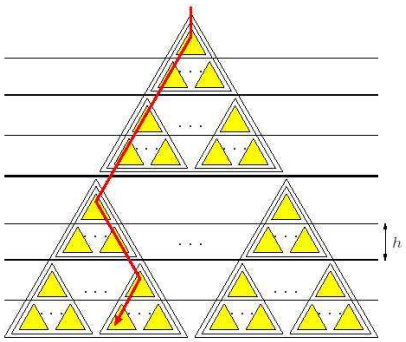


Figure 4: Van Emde Boas Layout

2.2 Cache-oblivious algorithms

Cache-oblivious means almost cache independent, i.e. the algorithm has good efficiency and compatibility on all kinds of memory hierarchies. The theory of cache-oblivious algorithms is based on the ideal-cache model. In this model, there are two levels in the memory hierarchy, called cache and disk. The disk is partitioned into memory blocks each consisting of a fixed number of B consecutive cells. The cache has room for C memory blocks. Although this model seems unrealistic, Frigo and Leiserson showed that it can simulate any memory hierarchy with a small constant-factor overhead. Many cache-oblivious algorithms including search trees have been developed to run well on all memory hierarchies without knowing the exact value of B and C. [4]

Current cache-oblivious tree algorithms include the Van Emde Boas layout and Packed-Memory Model [4]. Intuitively, placing a node into consecutive memory positions with its children increases spatial locality because, frequently, accessing a node's children occurs after accessing the node itself. This is precisely the idea behind the Van Emde Boas layout, illustrated in figure 4. By splitting a tree at mid-height, the tree breaks into top recursive and bottom recursive subtrees. This layout incorporates the recursive nature of each subtree. It achieves the optimal $O(\log N)$ memory transfers for a static optimal search tree.

For a dynamic search tree, we must further consider the destruction of the tree layout. Itai, Konheim and Rodeh have developed a technique called packed memory structure, which requires only $O(\log^2 N)$ memory transfers on B-trees [4]. The idea is to use a sparse method for storing useful elements. Each time an insertion or deletion occurs in the tree, we do it directly because the sparse

	Algorithm M1	Algorithm M2
Regrouping methods	*Van Emde Boas layout*	*Sort by frequency of access*
Structure splitting	*No*	*Yes*
Regrouping time	*noise* $>$ *threshold*	*Search operations* $>$ *threshold*

Table 1: Algorithms M1 & M2

Length of Search Sequence	Top-down (sec)	Bottom-up (sec)
10000000	3.36	2.67
10000000	3.35	2.57
10000000	3.38	2.62
20000000	6.65	5.21
20000000	6.56	5.18
20000000	6.76	5.13

Table 2: bottom-up Vs. top-down

array provides space to do it. There is also an associated threshold to determine when an array is too full or too sparse. When that threshold is reached, we redistribute the entire tree in a Van Emde Boas layout.

A splay tree is trickier because after each insertion or deletion, it transforms the entire path from the node to the root. We discuss a simplified algorithm next.

2.3 Memory Optimization algorithms

Based on the memory optimization techniques above, an ideal algorithm should have the following components: a dynamic analyzer that provides accurate information about the reference affinity of tree nodes; a regrouping routine that adjusts the memory layout efficiently, which can speed up the tree search at the expense of overhead; and a decision-maker which decides when to adjust the memory layout to get the minimal overhead and maximal speedup.

However, in practice, the algorithm has excessive overhead, especially in the analyzer. Getting accurate affinity information requires run-time monitoring, which costs an unacceptable amount of computing resource, even with the most efficient known techniques. Instead, we use heuristics to approximate the optimal algorithm. These heuristic rules are straightforward, e.g. a tree node should be placed close to its children. Two algorithms are developed using different heuristics summarized in table 1.

The regrouping routine in $M1$ will first get the Van Emde Boas layout of the entire tree and sequentially place all the nodes in one small memory block. This strategy provides some benefit on spatial locality by grouping the children and their parents in nearby locations. But as discussed, splaying can destroy these localities. So we use a noise flag to measure how far the data has been shuffled. To simplify the implementation, we can consider the noise as the level along the access path because rotation operations are proportional to the levels and are the main reasons of disturbances. If the noise reaches the threshold, then the redistribute function will completely rearrange the tree. The amortized time bound of $M1$ is $O(\log N)$ because it has only the overhead of redistribution. But with careful selection of a threshold, the frequency of redistribution can be reduced. Still, there is a trade-off in selecting a threshold. In next sections different threshold values are evaluated to find the best.

Algorithm $M2$ has three major differences from $M1$. First, it uses structure splitting. Secondly, its regrouping routine places the nodes according to the times it has been accessed recently. Doing this requires adding a new node member to record access time. Thirdly, it does not use noise to determine when to regroup. The rationale is that, if a node is accessed very often, it should be placed near the root, so that the cost of finding this element is less.

3. EVALUATION

3.1 Bottom-up Vs. Top-down

To compare the performance of bottom-up and top-down splaying, a bottom-up splay tree is implemented using exactly the same data structure as the original except that a pointer is added to its parent at each tree node. Test environment:

Machine: *PC*
CPU: *P4 2.0 GHz*
Memory: *512M*
OS: *Linux kernel 2.4.20-19.9*
Inputs: *random generated keys in [1, 10000]*

Comparing the two programs, we found that the bottom-up version averaged 22% percent faster than the top-down version. Both methods used a top-down procedure to find the searched node. The bottom-up algorithm has good locality because it acts like a stack, which first visits the last accessed node. So the reuse distance of bottom-up is about $N/2$ while top-down is N. The algorithm was run against a random memory access generator and compiled on a Linux system with gcc. Results appear in table 2.

3.2 Heuristic algorithm $M1$ with random inputs

As discussed above, the $M1$ algorithm regroups the tree nodes by sequentially placing all the nodes in a Van Emde Boas layout when noise $>$ threshold. Since the regrouping routine has overhead, the efficiency of $M1$ depends on two factors: the subtree size in the Van Emde Boas layout and the threshold to regroup. Figure 5 shows the test results under the following environment:

Machine: *PC*
CPU: *P4 2.0 GHz*
Memory: *512M*
OS: *Linux kernel 2.4.20-19.9*
Inputs: *random generated keys in [1, 10000]*
length of input sequence is 50,000,000.

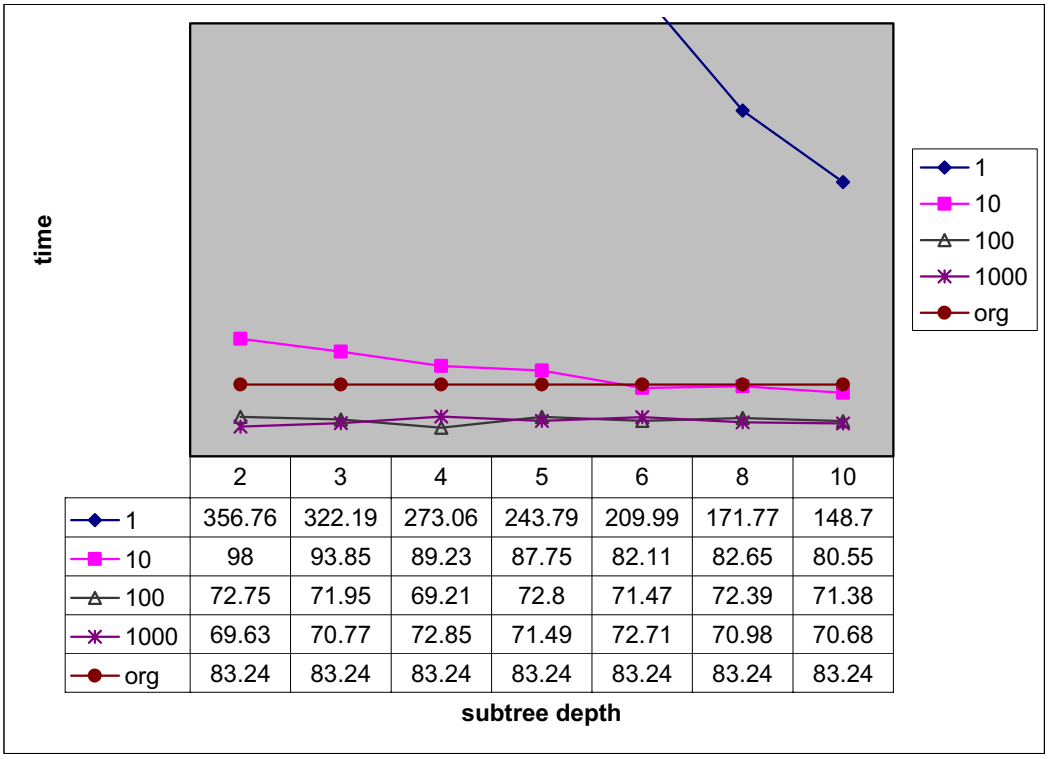
We can see from the result that the overhead of regrouping cannot be ignored, because with a very low threshold, the running time increases dramatically. Another observation is that regrouping speeds up the execution, e.g. (100, 4) is faster than (1000, 4), although it did more redistribution than the latter one. By carefully choosing the proper values for a threshold and subtree depth, we can increase the performance of the splay tree by 10% for randomly generated inputs.

3.3 Heuristic algorithms M1 and M2 on Spec2K benchmarks

Platforms. The experiments use $DEC-Alpha$ machines with OSF1 V4.0 cc compiler.

Tools. A source-level instrumentation and a run-time monitor profile accesses to individual data. The instrumentation program is called atom, which is only available on $DEC-Alpha$ machines. The profiling tool was developed by Ding and Zhong to measure the reuse signatures of programs. [11]

The test result is displayed in Figure 6. Compared to random inputs, $M1$ does not get the same performance improvement: only



	2	3	4	5	6	8	10
1	356.76	322.19	273.06	243.79	209.99	171.77	148.7
10	98	93.85	89.23	87.75	82.11	82.65	80.55
100	72.75	71.95	69.21	72.8	71.47	72.39	71.38
1000	69.63	70.77	72.85	71.49	72.71	70.98	70.68
org	83.24	83.24	83.24	83.24	83.24	83.24	83.24

Figure 5: $M1$ **with random search operations.** X**-axis is the depth of the Van Emde Boas recursive subtree,** y**-axis is the execution time. Different lines correspond to different threshold,** 10 **means *if (noise¿10*number of nodes in this tree) then regroup;*** org **is the original splay tree program.**

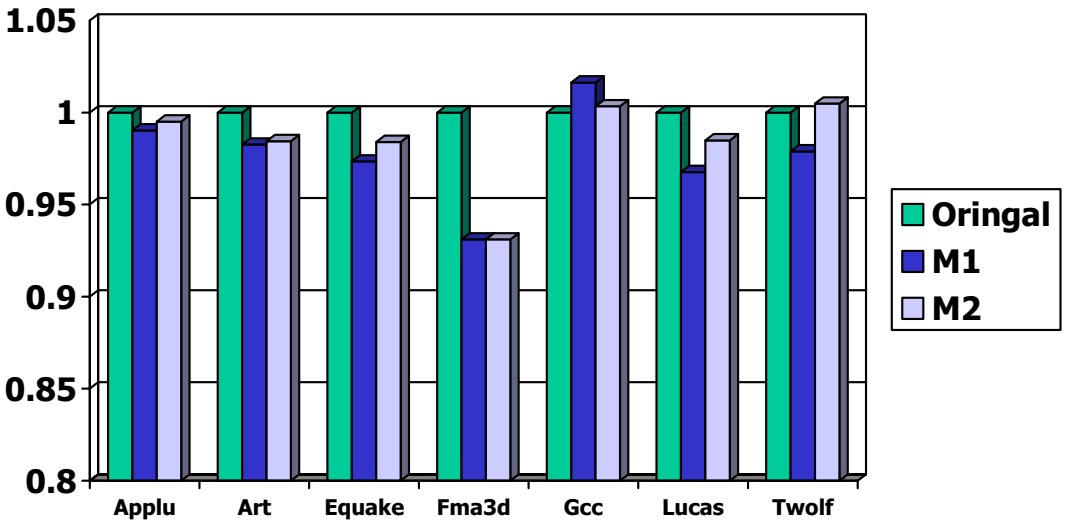


Figure 6: The test results of $M1$ **and** $M2$ **algorithm on Spec2K benchmarks. The test is on** $DEC - Alpha$ **machines by using the system program atom, which is a source-level instrumentation and run-time monitor. The** y **axis is scaled value of execution time; the original is normalized to 1.**

1∼3%. The reason is that the affinity assumption works well for random tree searches but gains little improvement for program traces, where the possibility of accessing left and right children may differ significantly. Also the layout of the splay tree changes too frequently, which reduces the benefits of regrouping.

It is unexpected that $M2$ works almost as well as $M1$ although its heuristic is very naive; namely, sort by access frequency. This shows that only a small part of the total tree is visited often. Moving them to the root will improve the performance, albeit not by much.

4. CONCLUSIONS

In practice, splay trees show very complex memory access patterns, which make simple grouping strategies not applicable in many cases. However, if we assume that the possibility of accessing left and right children are nearly equal, then the $M1$ algorithm may reduce the execution time up to 10%. Of course, in real application programs, due to the existence of inherited access patterns, usually it is not possible to gain such dramatic improvements. To the best of our knowledge, this is the first study to compare top-down splay trees with the bottom-up splay trees.

5. ACKNOWLEDGMENTS

The original splay-tree implementation was provided by Daniel Sleator. Our experiments were mainly based on local machines purchased by several NSF Infrastructure grants and equipment grants from Compaq/HP and Intel. We also used the reuse distance analyzer developed by Ding and Zhong.

6. REFERENCES

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, 2001.

[2] Daniel Dominic Sleator, Robert Endre Tarjan. Self-Adjusting Binary Search Trees. In *Journal of the Association for Computing Machinery*,Vol.32, No.3, July 1985, pp.652-686.

[3] Adelson-Velskii G.M., and Landis E.M An algorithm for the organization of information. In Sov. Math. Dokl. 3 (1962), 1259-1262..

[4] Michael A.Bender, Erik D.Demaine, and Marine Farach-Cloton. Cache Oblivious B-Trees. In *Proc. 13th ACM-SIAM Symp.* On Discrete Algorithms (SODA), 2002.

[5] A.Itai, A.G.Konheim, and M.Rodeh. A sparse table implementation of priority queues. In *Proc. 8th Colloquium on Automata, Languages and Programming, LNCS 115*, pp.417-431, July 1981.

[6] P.van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proc.16th IEEE Symposium Found. Comp. Sci.*, pp. 75-84, Berkeley, California, 1975.

[7] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, 2001.

[8] C. Ding. *Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse*. PhD thesis, Dept. of Computer Science, Rice University, January 2000.

[9] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[10] Yutao Zhong, Maksim Orlovich, Xipeng Shen, and Chen Ding. Array Regrouping and Structure Splitting Using Whole-Program Reference Affinity. To be appear on *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Washinton DC, June 2004.

[11] Chen Ding and Yutao Zhong. Predicting Whole-Program Locality through Reuse-Distance Analysis. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, California, June 2003.

[12] Y. Zhong, C. Ding, and K. Kennedy. Reuse distance analysis for scientific programs. In *Proceedings of Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Washington DC, March 2002.

[13] Gerth Stlting Brodal. External Memory Algorithms In *IT-C Course on Advanced Algorithms*,Department of Computer Science, University of Aarhus.

[14] Hu, T.C., and Tucker, A.C. Optimal computer-search trees and variable-length alphabetic codes. In *SIAM J. Appl. Math. 37*,pp. 246-256, 1979.