

Restructuring Computations for Temporal Data Cache Locality

Venkata K. Pingali

Information Sciences Institute
University of Southern California
Los Angeles, CA 90292
pingali@isi.edu

Sally A. McKee

Electrical and Computer Engineering
Cornell University
Ithaca, NY 14853
sam@csl.cornell.edu

Wilson C. Hsieh

School of Computing
50S Central Campus Drive, Room 3190
University of Utah
Salt Lake City, UT 84112
wilson@cs.utah.edu

John B. Carter

School of Computing
50S Central Campus Drive, Room 3190
University of Utah
Salt Lake City, UT 84112
retrac@cs.utah.edu

This research was sponsored in part by National Science Foundation award 9806043, and in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number F30602-98-1-0101 and DARPA Order Numbers F393/00-01 and F376/00. The views and conclusions contained herein are those of the authors, and should not be interpreted as representing the official policies or endorsements, either express or implied, of NSF, DARPA, AFRL, or the US Government. This research was conducted while the first author was a student at the University of Utah.

Abstract

Data access costs contribute significantly to the execution time of applications with complex data structures. As the latency of memory accesses becomes high relative to processor cycle times, application performance is increasingly limited by memory performance. In some situations it is useful to trade increased computation costs for reduced memory costs. The contributions of this paper are three-fold: we provide a detailed analysis of the memory performance of seven memory-intensive benchmarks; we describe *Computation Regrouping*, a source-level approach to improving the performance of memory-bound applications by increasing temporal locality to eliminate cache and TLB misses; and, we demonstrate significant performance improvements by applying Computation Regrouping to our suite of seven benchmarks. Using Computation Regrouping, we observe a geometric mean speedup of 1.90, with individual speedups ranging from 1.26 to 3.03. Most of this improvement comes from eliminating memory stall time.

Keywords: Memory performance, Data structures, Optimization

1 Introduction

As memory latency grows relative to processor cycle times, the performance of many applications is increasingly limited by memory performance. Applications that suffer the most are characterized by complex data structures, large working sets, and access patterns with poor memory reference locality. Examples of such codes can be found in myriad domains. For instance, memory stalls account for 58-99% of the execution time of applications we studied in the domains of signal processing, databases, CAD tools, and high performance graphics ⁽¹⁾. Worse, the contribution of memory stall time as a fraction of total execution time is increasing as the performance gap between processors and memory grows. Reducing memory stall time improves processor resource utilization and allows an application's performance to scale with improvements in processor speed and technology.

Many hardware, software, and hybrid approaches have been investigated for improving the memory performance of applications ^(2, 3, 4, 5, 6, 7, 8). Effective optimizations exist for modest-sized applications with fairly regular data structures ^(3, 9, 10, 8), but the memory performance issues of large pointer-based applications have received less attention. Most approaches to improve the memory performance of such applications try to improve spatial locality through controlled data layouts ^(11, 12, 13), but optimal layouts are hard to achieve in applications with complex or dynamic data structures. Furthermore, the complexity of the access patterns often renders data restructuring attempts ineffective.

In this paper, we present *Computation Regrouping*, an optimization technique that trades off increased computation for reduced memory overheads; in some cases it even reduces both. We first characterize the memory access patterns of seven benchmark programs. We observe that the benchmarks can be characterized as a series of *logical operations*, short streams of computation that represent a single conceptual operation from the point of the view of the programmer. For example, a logical operation in the R-Tree

benchmark consists of performing a single insert, delete, or update on the spatial database; a logical operation in the Ray Trace benchmark consists of evaluating a single ray’s path through the scene. Many data objects are accessed within each logical operation, and there is little or no reuse. In contrast, there is significant reuse across logical operations. The key insight behind Computation Regrouping is that memory performance can be improved significantly by modifying the order in which logical operations (or portions of logical operations) are performed so that operations with similar memory footprints are executed together.

Computation Regrouping identifies a set of data objects that fit into the L2 cache and reorganizes the computations from multiple logical operations to best utilize these cached objects. This reorganization typically involves extending application data structures and/or modifying application control flow to group computations that access common data objects. The “batched” computations can be executed with higher temporal data locality, which improves cache performance. We identify two classes of regrouping optimizations: *early execution* and *deferred execution*. Early execution brings forward future operations that touch data accessed by the current operation. Tiling is an example of applying early execution to dense array-based computations⁽¹⁴⁾. Deferred execution explicitly or implicitly delays operations until enough operations are pending that access the same data to amortize the associated data accesses. Operations can be deferred directly (e.g., by explicitly queuing operations) or indirectly (e.g., by making repeated passes over a large indirection structure and only performing operations on entries that meet some filter requirement⁽¹⁵⁾). Unlike most previous approaches to improving the memory performance of applications with complex data access patterns, we focus on improving *temporal* locality rather than *spatial* locality. Thus, our approach is complementary to most previous approaches. Section 6 discusses related work in more detail.

Applications that can benefit from this work are those to which the logical operation paradigm applies, and for which processor-memory bottlenecks

account for a large portion of the runtime. Consider an application that performs one million tree lookup operations on a tree that is much larger than can fit in the L2 cache. Such an application will suffer frequent cache and TLB misses due to poor locality between the independent lookups. One way to employ Computation Regrouping to improve performance in this case would be to introduce queues at various levels of the tree, and “batch” together groups of lookups to the same subtree. This optimization would greatly improve memory performance and increase operation throughput, at the cost of extra computation to manage the queues and increased latency of operations that get deferred. We discuss a detailed example of such a regrouping optimization for the R-Tree benchmark in Section 4.2.

In this paper we describe how we apply Computation Regrouping techniques to seven memory-bound benchmarks of varying complexity that span a number of application domains. We apply the optimizations by hand, which we show requires only modest programmer effort. For these applications, regrouping results in a geometric mean application application speedup of 1.90, with individual speedups ranging from 1.26 to 3.03. These speedups are the direct result of reducing memory stall time by 26-85%, and often come despite increasing the number of instructions executed. These results demonstrate the value of increasing computation to reduce memory stalls.

Our results demonstrate the potential value of reorganizing the logical operations within an application. It remains an open question how best to apply Computation Regrouping to a particular program. The key lies in identifying what constitutes a “logical operation” for each application: once they are identified, it is a fairly simple matter to modify the application’s control and data structures to implement regrouping. Although we hand-optimized the codes for our results, it should be possible to automate regrouping partially. We discuss some techniques for doing so in Section 5.

The remainder of this paper is organized as follows. Section 2 describes in detail our benchmarks and their high-level characteristics. Section 3 discusses our proposed transformations, and Section 4 presents results obtained

on applying these transformations. Section 5 discusses our approach to automating the computation regrouping optimization. Section 6 discusses related approaches. Section 7 summarizes our results and identifies some open problems. This paper is a shortened version of the first author’s thesis; more details are given there ⁽¹⁾.

2 Benchmark Analysis

For our experiments we select seven publicly available benchmark applications that spend 25% or more of their execution time servicing secondary data cache and TLB misses. We use a carefully optimized version of each benchmark as a baseline and implement function-call and layout-related optimizations where appropriate. We eliminate as candidates many popular applications that do not meet the 25% threshold after these optimizations, e.g., Barnes-Hut ⁽¹⁶⁾ and MST ⁽¹⁶⁾. We experiment with a range of inputs to determine the sensitivity of each application to input parameters, and we examine the source code to ensure that poor memory performance is due to access patterns (such as indirect accesses and strided accesses), and not an artifact of our experimental setup.

We perform all experiments, including our initial benchmark selection, on a 14-processor SGI Power Onyx with 194MHz MIPS R10000 processors and 4GB of main memory. Each processor has 32KB L1 instruction and data caches, and a 2MB unified, write-back, two-way set-associative L2 cache. The R10000 processor is four-way superscalar and implements dynamic instruction scheduling, out-of-order execution, and non-blocking cache access. We use the SpeedShop performance tool suite ⁽¹⁷⁾ and the perfex ⁽¹⁸⁾ command line interface to access processor performance counters. All programs are compiled using the MIPSpro C compiler with -O3 optimization.

[Table 1 about here.]

The seven benchmarks cover a range of application domains (scientific codes, hardware design tools, graphics, and databases), code sizes, and complexities. Table I presents cache hit ratios for sample inputs and shows how the execution times scale with input sizes. The table also presents the estimated memory stall times provided by perfex. Each benchmark is built around a core data structure, and we find that the bottleneck for each benchmark is due to specific logical operations over this data structure. Source code sizes vary from 280 lines in IRREG to 60K lines in CUDD. Data structure complexity varies from a simple array in IRREG to a complex, directed acyclic graph (DAG) based on hash tables in CUDD. We next describe each benchmark and its baseline performance in detail.

R-Tree: An R-Tree ⁽¹⁹⁾ stores spatial information about objects such as airports and restaurants. Queries can include spatial constraints such as “all printers in this building” and “landmarks within five miles of the airport”. To execute operations efficiently, an R-Tree groups objects based on their spatial attributes. R-Trees are known to perform well when the number of data dimensions is low. Our R-Tree reference implementation is the DARPA DIS Data Management benchmark ⁽²⁰⁾.

[Figure 1 about here.]

Figure 1 illustrates the data structure used to implement the R-Tree in our benchmark: a height-balanced tree with a branching factor between two and fifteen. Each subtree is associated with a bounding box in four dimensional space, or a *hypercube key*, that encompasses the bounding boxes of all the nodes within the subtree. Each internal node contains a set of linked list entries, where each list member is a (*child, hypercube key*) pair corresponding to one child subtree of the node. Leaf nodes entries point to the data objects. Simple tree balancing mechanisms are invoked during node merging and splitting.

Query and delete tree operations require searching, whereas insert operations access nodes along a single path from root to leaf. Valid matches

include all objects whose hypercube keys overlap the search hypercube key. A search key can potentially overlap with multiple objects' boxes, and hence may trigger multiple tree traversals along different paths. Poor performance arises from the many internal nodes touched per search: often up to 50% of all nodes in the tree. For a sample run with 12K query operations on a tree of 67K nodes, the L1D and L2 cache hit ratios are 97.11% and 57.63%, respectively. Execution time scales almost linearly with the size of the tree and the number of tree operations.

[Figure 2 about here.]

IRREG: The IRREG kernel is an irregular-mesh PDE solver taken from computational fluid dynamics (CFD) codes ⁽²¹⁾. The input mesh's nodes correspond to bodies (molecules) and its edges to interactions, represented by arrays x and y , respectively, in Figure 2. The forces felt by the bodies are updated repeatedly throughout execution. The largest standard input mesh, *MOL2*, has 442,368 nodes and 3,981,312 edges. With this input, running the loop in Figure 2 for 40 iterations yields L1D and L2 hit ratios of 81.81% and 53.86%. As in EM3D (described later in this section), indirect accesses to remote nodes (array x) are expensive. For large data arrays with sufficiently random index arrays, execution time scales with the number of edges and iterations.

Ray Trace: In Ray Trace (a.k.a. RayTray), part of the DIS benchmark suite ⁽²⁰⁾, rays are emitted from a viewing window into a 3D scene. A ray is defined by an origin (position) in the viewing window and a direction, and each ray is traced through its reflections. The algorithm checks each ray for intersection with all scene surfaces and determines the surface closest to the emission source. Scene objects are allocated dynamically and stored in a three-level hierarchy. The number of objects is fixed per scene, whereas viewing window size, and hence the number of rays, is specified by the user.

When the object structure is significantly larger than the L2 cache size, data objects are evicted before reuse, and successive rays suffer the same

(large) number of misses. We report on the *balls* input, which has 9,828 surfaces, 29,484 vertices, and a memory footprint of about 6.5MB. The L1D and L2 cache hit ratios for a window size of 1024×1024 are 97.6% and 70.15%, respectively. Processing each ray requires that nearly the entire object data structure be accessed, with strided accesses and pointer chasing being the dominant access patterns. Execution time scales almost linearly with the viewing window size (i.e., the number of emitted rays) and input scene size.

FFTW: The highly tuned FFTW benchmark ⁽²⁰⁾ outperforms most other known FFT implementations ⁽²²⁾. The core computation consists of alternating passes along the X, Y, and Z dimensions of a large array. Accesses along the Z dimension account for 50-90% of the total execution time of FFT, depending on the input size. The Z stride is typically much larger than a virtual memory page, and when this dimension is sufficiently large, there is little or no reuse of cache lines before they are evicted. Accesses during this phase of the program therefore suffer very high cache and TLB miss ratios. For an input array of size $10240 \times 32 \times 32$, the memory footprint is 320MB, and the L1D and L2 hit ratios are 97.01% and 64.62%, respectively. Execution time scales linearly with increasing input dimensions.

CUDD: The CUDD Binary Decision Diagram (BDD) manipulation package ⁽²³⁾ is widely used in the hardware design community. Based on Shannon’s boolean-expression evaluation rules, BDDs compactly represent and manipulate logic expressions. The core structure is a large DAG whose internal nodes represent boolean expressions corresponding to the subgraphs of lower nodes. Each DAG level is associated with a boolean variable, and the large structure and the nature of the manipulations requires that each level of internal nodes be stored in a separate hash table. The DAG depth is the number of boolean variables. Since variable ordering is important to obtaining a compact structure, the package provides several methods for dynamically changing the ordering. The main operation among all these methods is a variable swap with three steps: extracting nodes from a hash table, updating a second hash table, and garbage-collecting nodes. Each of these steps

touches many nodes, which causes cache behavior similar to that of FFT. For a sample input circuit, *C3540.blif*, with random dynamic reordering of variables, the L1D and L2 hit ratios are 94% and 45%, respectively.

EM3D: EM3D models the propagation of electromagnetic waves through objects in three dimensions ⁽²⁴⁾. The core data structure is a bipartite graph in which nodes represent objects and edges represent the paths through which the waves propagate. The user specifies the number of nodes and degree at each node, and the program randomly generates graph edges to fit these constraints. The primary computation updates each node’s state based on those of its neighbors. We use a slightly constrained version of EM3D in which the number of nodes is a power of two and the out-degree is fixed (arbitrarily) at 10. For an input size of 128K nodes and degree 10, the memory footprint is about 30MB, with corresponding L1D and L2 hit ratios of 94.96% and 47.77%. Our experimentation confirms Culler *et al.*’s observation that the primary performance determinant is the cost of remote node accesses from nodes in one half of the bipartite graph to nodes in the other ⁽²⁴⁾. Execution time scales linearly with the number of nodes, and scales sub-linearly with increasing degree.

Health. From the Olden benchmark suite ⁽¹⁶⁾, Health simulates the Columbian health care system. The core data structure is a quad-tree with nodes representing hospitals at various logical levels of capacity and importance. Each node is associated with personnel and three lists of patients in various stages of treatment. Patients are first added to the waiting list, where waiting time depends on the number of personnel and patients. When hospital personnel are available, a patient enters the treatment phase, which takes a fixed amount of time. After treatment, the patient exits the system or goes to a bigger hospital where the process repeats. Each simulation cycle visits every quad-tree node, updating patient lists. After 500 simulation cycles on a sample input, the L1D and L2 hit ratios are 87.47% and 68.65%, respectively. Execution time increases linearly with tree size and number of simulation cycles.

3 Computation Regrouping

This section begins with a discussion of *logical operations* in our benchmark applications, including a detailed discussion of R-Tree’s logical operations. We then discuss four simple optimizations that employ Computation Regrouping techniques, summarize their properties, and discuss the tradeoffs involved.

3.1 Logical Operations

To identify the logical operations in an application, we look for operations performed many times on different parts of the data set. For most of the benchmark applications, these logical operations are trivial to identify, as they are the primary, programmer-visible operations performed within the application (e.g., inserting a record in R-Tree, performing FFTs over a single dimension in FFTW, or performing a single variable swap in CUDD). Each logical operation touches a potentially large number of data objects.

We define the *critical working set* to be the average size of the data objects accessed by a single logical operation. Table II presents the list of logical operations and the associated critical working sets for our benchmark applications. The *estimated threshold* in Table II represents the minimum problem or parameter size that causes a single logical operation’s working set to exceed the 2MB L2 cache size of our experimental platform.

[Table 2 about here.]

To illustrate our concept of a logical operation, we consider the R-Tree benchmark in detail. In R-Tree, insert, delete and query operations require tree traversals whose accesses are data-dependent and unpredictable at compile time. Each traversal potentially performs many hypercube-key comparisons. In particular, query and delete operations traverse multiple tree paths and have large working sets, often over 50% of the R-Tree.

[Figure 3 about here.]

Figure 3 illustrates memory access patterns for R-Tree operations. In Figure 3(a), each point represents an access made to the node associated with each x -coordinate by the query identified by the y -coordinate. Each row of points identifies all accesses by a single query, and each column identifies queries accessing a given node. This simple plot identifies two important characteristics of the R-Tree benchmark. First, the set of nodes accessed by successive queries rarely overlap, since there are significant gaps along the vertical axis. Second, the set of unique nodes accessed between successive visits to a given node is large: it is the union of nodes accessed by any query lying between the two successive queries to the given node. Figure 3(b) shows the distribution of this union’s size with query number. The size of this union is typically large, sometimes including half the total tree nodes. In general, queries for which there is significant data overlap are widely separated in time. Thus, for large trees, there is a high likelihood that a given node will be evicted from cache between successive operations.

R-Tree’s access behavior is representative of the applications we study, all of which exhibit several conditions that lead to poor temporal locality. For instance, the critical working set sizes typically exceed cache size. The applications consist of many logical operations, where each operation potentially accesses many data objects, but where there are few accesses to any given object with little computation performed per object.

3.2 Transformations

We identify two classes of regrouping optimizations: *early execution* and *deferred execution*. In this section we present a detailed discussion of these two sets of optimizations, as well as two special cases of deferred execution, *computation summation* and *filtered execution*. The ideas behind each transformation are similar, but the implementations, tradeoffs, and effects on application source code are sufficiently different that we treat each separately.

Applying the techniques requires identifying logical operations suitable for regrouping, choosing application extensions to implement the regrouping, and choosing mechanisms to integrate results from regrouped computations and non-regrouped computations. Table III summarizes the general cost and applicability of each technique.

[Table 3 about here.]

Early Execution: Early execution brings forward future computations that touch data accessed by the current computation. In some instances, early execution can be viewed as a generalization of traditional tiling approaches. For example, FFT’s Z-dimension walks elements represented as 16-byte complex numbers. Assuming 128-byte cache lines, each line loaded during one walk contains data accessed by exactly seven other Z-dimension walks. Serially executing the walks fails to exploit this overlap. Early execution performs computations from all eight walks on a resident cache line. Identifying future computations and storing their results is straightforward. The optimized CUDD moves the reference counting for garbage collection forward in time, and performs it along with data-object creation and manipulation. The ideas bear some similarity to incremental garbage collection, as in Appel, Ellis, and Li’s virtual memory-based approach ⁽²⁵⁾, but we optimize for a different level of the memory hierarchy, and do not strive for real-time performance. In both FFT and CUDD, code changes are largely isolated to control structures. Data structure changes are minimal, and integration of partial results is straightforward.

Deferred Execution: When there is uncertainty in the number and/or order of logical operations to be performed, computations can be explicitly or implicitly delayed (as in *lazy evaluation*) until sufficiently many accumulate to amortize the costs of fetching their data. The deferred computations can then be executed with high temporal locality. In the R-Tree from Figure 1, deferring queries, inserts, and deletes requires changes to both control and data structures: we associate a small operation queue with a subset of

tree nodes, and we enqueue tree operations until some dependence or timing constraint must be satisfied, at which time we “execute” the queues. Since all queued queries access nodes within the subtree, their accesses have good cache locality. An operation might reside in multiple queues over the period of its execution, and so we trade increased individual operation latency for increased throughput. Synchronization overhead for queries is insignificant, since they are largely read-only. Supporting deletes and inserts gives rise to larger overheads, due to additional consistency checks. Deferred execution can also be applied to CUDD; code changes are largely to its control structure, and partial result integration is thus simpler.

Computation Summation: Computation summation is a special case of deferred execution in which the deferred computations are both cumulative and associative. Two or more deferred computations are replaced by one achieving the aggregated effect. For instance, the most common operation in Health increments a counter for each patient in a waiting list at each node in the tree. These increments can be deferred until an update is forced by the addition or deletion of list elements. This optimization is more specialized than deferred execution, and therefore less generally applicable. However, computation summation can be implemented efficiently, and it improves both computation and memory performance. EM3D benefits from similar optimizations.

[Figure 4 about here.]

[Figure 5 about here.]

[Figure 6 about here.]

Filtered Execution: Filtered execution is a special case of deferred execution in which the deferral is achieved implicitly by the modified loop control structure. Filtered execution places a sliding window on a traversed data structure and allows accesses only to the part of the structure visible through this window. Figures 4 and 5 illustrate how filtered execution can

be applied to a simple, indirect-access kernel. In each iteration, accesses to locations outside the window are deferred. The use of a sliding window improves temporal locality in each iteration at the cost of added computational and source-code complexity. This technique is a generalization of the optimization for non-affine array references proposed by Mitchell et al. ⁽¹⁵⁾.

Figure 6 shows the filtered execution applied to the IRREG main kernel from Figure 2. Note that increased computational costs may be higher than the savings from reduced cache misses; Figure 7 illustrates the cost/performance tradeoffs for varying window sizes (with respect to data array sizes) in IRREG. For appropriately large windows, the reduction in miss stalls dominates control overhead, but for windows smaller than a threshold value based on the cache size, control overhead dominates. Ray Trace and EM3D behave similarly.

[Figure 7 about here.]

3.3 Tradeoffs

Regrouping can increase complexity. First, control and data structure extensions make it more difficult to code and debug the application, although libraries or templates could be used to hide much of this complexity. Second, reordering computations can reorder output, so care must be taken to preserve the correct output order (if it matters). Third, some regrouping techniques require selecting appropriate optimization parameters via experimentation: for example, choosing the optimal queue length in R-Tree or an optimal filter window size in IRREG.

4 Results

[Table 4 about here.]

We study the impact of Computation Regrouping on our benchmarks using the experimental setup from Section 2. We incorporate the appropriate

regrouping optimizations into the reference implementations, and execute them all on the same sample inputs. To remove artifacts due to OS scheduling or page allocation and to cover a variety of system load conditions, we execute the baseline and optimized versions of the applications ten times over a period of a few days. The data represents the mean values obtained for these ten experiments.

Table IV presents a summary of the specific optimization techniques and their high-level impact on each application. Complex applications such as CUDD benefit from multiple regrouping techniques. The table shows the extent of the application modifications. We find that they are small compared to the overall application size. In the case of FFTW, the same 40-line modification had to be replicated in 36 files, which is reflected in the large change reported. The table also presents details of the number of instructions issued and graduated. In some cases, such as IRREG and R-Tree, this number is higher in the restructured application, which demonstrates the tradeoff identified in Section 3.3: increased computation can yield improved performance.

In Section 4.1 we discuss overall performance results. In Section 4.2 we discuss the regrouping optimization for R-Tree in more detail.

4.1 Performance Summary

Table V summarizes the effectiveness of Computation Regrouping for improving the memory performance of our benchmark applications. The geometric mean speedup is 1.90, although individual speedups vary significantly. Improvements are input-dependent, and the impact on performance is occasionally negative. Nonetheless, our experiences have been positive in terms of the applicability and effectiveness of Computation Regrouping. We experiment with a range of inputs for each benchmark and observe positive speedups for most inputs. Reducing memory stall time yields almost all of the measured performance improvements. In some cases, computation costs are reduced as well. In cases where computation decreases, especially in

applications optimized using computation summation, we cannot accurately compute overhead.

We present the *typical* perfex estimate values for miss ratios and miss-handling times. Perfex derives these estimates by combining processor performance counter data with execution models. In our observation, perfex performance numbers are generally optimistic, but they provide fairly good estimates of relative costs of performance parameters.

[Table 5 about here.]

Table V presents perfex-estimated statistics for total execution time and for servicing cache misses, along with the computational overhead of regrouping and the execution time savings as a percentage of the original memory stall time. The *Saved* column shows the fraction of original memory stall time recovered. The *Overhead* column shows the fraction of original memory stall time spent on additional computation. Some entries are blank, either because we observe a savings in the computation or because of inconsistencies in the perfex estimates (where the subtotal times do not sum to the total). The decrease in memory stall time is 26-85%, and the computation cost to achieve this reduction is 0.3-10.9% of the original stall time.

R-Tree: Section 3 explains how we extend the R-Tree data structure with queues to support the deferral of tree query and delete operations. Our experimental input, *dm23.in*, consists of a million insert, delete, and query operations. The execution style of these queries is similar to batch processing, so increasing the latency of individual operations to improve throughput is an acceptable tradeoff. Overall speedup for the *dm23.in* input is 1.87. The average tree size is about fifty thousand nodes, which consume approximately 16MB. Cache hit ratios improve significantly in the optimized version of the program: from 95.4% to 97.1% for the L1D cache, and from 49.6% to 77.9% for the L2 cache. TLB performance improves somewhat, but the effect is small. The overhead incurred from the queue structure to defer query and

delete operations is 10.5% of the original memory stall time, but this cost is overwhelmed by the 60% reduction in stall time.

IRREG: IRREG is the simplest code: the main loop is fewer than 10 lines of code. We can easily apply filtered execution, as shown in Figure 6. A filter window that is one-fourth of the array size yields a speedup of 1.74 for the *MOL2* input. The L2 cache hit ratio improves from 54.5% in the baseline application to 84.3% in the optimized version. Table V shows that regrouping eliminates 56% of the memory stall time while executing 11% more instructions. The actual savings in memory stall time is higher than the estimated 56% because the baseline execution time is higher than the perfex estimate by about 50s, and the execution time of the optimized version is lower than the perfex estimate by 15s. The large data memory footprint, the randomness in the input, and the small amount of computation in the innermost loop are important to obtaining good performance improvements. Other standard inputs, such as *MOL1* and *AUTO*, are small enough to fit into the L2 cache, or are compute-bound on our experimental machine.

Ray Trace: We apply filtered execution to Ray Trace. The straight-line code of ray processing is replaced by a two-iteration loop similar to the modified IRREG code shown in Figure 6. The choice of filter window size depends on both the size of the input data structure, and to a lesser extent on the computational overhead of filtered execution. The critical working set of the baseline Ray Trace is about 4MB, so a filter window size of one half of the image allows most data to reside in the L2 cache in the optimized version. We defer processing reflected rays until all other rays have been processed. The overheads incurred are due to the extra filter checks and accesses to the array that stores reflected rays. We achieve a speedup of 1.98 for an input viewing window of 256×256 (64K rays). The L2 cache ratio improves from 70.1% to 99.6%, which makes the optimized version effectively compute-bound. 84.7% of stall time is recovered at a computational cost of 0.3% of execution time. We experiment with only one input scene, but our experience suggests that our results should scale with input size.

FFTW: FFTW translates the multidimensional FFT computation into a series of calls to highly optimized leaf-level functions called *codelets*, each of which contains a predetermined number of floating-point operations and memory accesses. Each codelet’s memory accesses are in the form of short walks along a single dimension of the input and output arrays. The critical working set of any codelet is no more than 64 cache lines, which fits easily into the L1D and L2 caches.

Early execution for FFT modifies its behavior such that when a “column” walk is executed, future “column” walks accessing the same data are executed at the same time. In this way, we simulate the effect of a short, synchronized multi-“column” walk. The first iteration loads a fixed number of cache that are reused by the remaining walks.¹ Doing so eliminates cache misses that would have occurred if the complete walks had been executed serially. The majority of the performance improvement is obtained during the *compute* phase walk along the *z*-dimension, but there is also significant gain from the *copy* phase. Control extensions in the form of function calls and loops cause regrouping overhead, but there is no additional memory overhead.

Table V shows that the application speedup is 2.53. Although the L1D hit ratio decreases slightly, the nearly 50% increase in the L2 hit ratio offsets that impact. Memory stall time decreases from 73.8% of the baseline application execution time to about 45% after optimization. The same optimization can be applied to the *y*-dimension to further improve performance, albeit far less dramatically.

CUDD: We experiment with the *nanotrav* tool from the CUDD package⁽²³⁾. Nanotrav stores input circuits as binary decision diagrams (BDDs) and executes a series of operations to reduce the BDD sizes. The main operation is a variable swap that manipulates large hash tables. Each hash table element is accessed at least three times, and these accesses are distributed among the extraction, insertion, and garbage collection stages of a swap. The

¹We have subsequently learned that the FFTW authors are experimenting with a similar approach.

many hash table elements processed in each stage introduce a temporal separation between successive accesses to a data object, which results in little or no reuse before eviction.

We implement two regrouping optimizations in *nanotrav*. First, we execute the reference counting step of the garbage collection function early — when the nodes become garbage — rather than during a separate garbage collection phase. Second, we defer the sorting of newly inserted objects until the garbage collection phase. We observe an application speedup of 1.26 for the *C3540.blif* input. Improved L1D performance is the primary factor. Memory stall time still accounts for much of execution time, and could be reduced further. Our experimentation suggests an alternative implementation of the BDD structures based on partitioned hash tables, but we have performed only a preliminary study of the feasibility of this solution.

EM3D: The EM3D application has a graph construction phase and a computation phase. We apply computation summation to the graph construction phase, which involves incrementing counters at remote nodes. We use an array of integers to collect the increments to these counters, and access the remote nodes once at the end. We also apply filtered execution to the compute phase, but it has limited impact because of high control overhead. We use a filter window size of one-third of the set of remote nodes. Our optimizations are effective only when the graph construction cost dominates: *i.e.*, when the number of nodes is large and the number of compute-phase iterations is small. For 128K nodes and one compute iteration, we observe a speedup of 1.43, which corresponds to a speedup of about 1.5 in the construction phase and 1.12 in the compute phase. Memory stalls still account for about 71% of the optimized application’s run time, down from 84% in the baseline. A significant source of improvement is the reduced computation in the first phase.

Health: We obtain one of our best results with computation summation in Health. We defer patient waiting-list timestamp updates until the list requires modification, which reduces the number of list traversals and

eliminates many cache misses. The overhead of deferring causes application slowdown at the beginning, when the waiting lists are short and require frequent modification. This is offset by the reduced number of accesses performed during later execution, when the waiting lists are long and incur infrequent modification. The simple structure of Health enables an efficient implementation of regrouping. Overall application speedup is 3.03. The L1D hit ratio increases from 71.9% to 85.2%, whereas the L2 hit ratio decreases from 68.6% to 46.8%. Even after optimization, most of execution time is spent servicing cache misses. To improve memory performance further, we can implement a more aggressive regrouping optimization that defers update operations from other lists and tree traversals.

The perfex estimates are inconsistent for this application, which makes it difficult to compute overheads or reductions in stall times. The difficulty of isolating the overhead is compounded by the fact that computation summation dramatically reduces the amount of computation. This change in total computation made it impossible to accurately calculate regrouping overhead.

4.2 R-Tree Details

In the remainder of this section, we discuss how we optimize the spatial index structure in R-Tree, both as an illustration of the generic use of computation regrouping and as a specific use of the deferred execution technique. We test our queue-extension/deferred-execution version of R-Tree with both real and synthetic inputs. The real *dm23.in* input consists of a million insert, delete, and query operations. The synthetic input, derived from *dm23.in*, consists of 300K inserts followed by 100K queries, a potential best case for this technique. For the synthetic input, we also implement clustering⁽¹¹⁾, an optimization for improving spatial locality. Queries are read-only operations incurring no tree-maintenance overheads; therefore, optimizations may be as aggressive as possible. When implementing regrouping, aggressive deferral

can be used, and when implementing clustering, optimal subtree blocks can be identified.

Clustering is a layout optimization that attempts to group data likely to be accessed with high temporal locality close together in space, as well. Clustering can have significant impact for certain pointer-based data structures, including binary trees ⁽¹¹⁾ and B-Trees ⁽¹³⁾. In an R-Tree, clustering can be achieved at two levels. First, within a single node the linked list of entries can be blocked into an array. We call this *intra-node clustering*. Second, nodes that share a parent-child relationship can be placed together. We call this layout modification *inter-node clustering*. In our implementation, inter-node clustering assumes intra-node clustering.

Our results for the synthetic input indicate that regrouping can potentially eliminate most memory stall time at modest computational cost. The memory-stall contribution to execution time decreases from almost 95% to less than 27% in the optimized version. We observe a query-throughput speedup of 4.38 over the baseline when applying both regrouping and clustering. This speedup is significantly higher than the factors of 1.85 or 3.48 from clustering or regrouping alone. We expect regrouping for temporal data locality to be similarly complementary to other spatial-locality optimization techniques.

Reordering tree operations results in a tree different from that in the original application; thus, we must ensure that tree operations in the modified version consider only those nodes that would have been considered in the unmodified R-Tree. Below we present more details of the mechanisms used to ensure the correctness properties. These mechanisms are not needed for our synthetic input case, but they are necessary for the general case where the insert, delete, and query operations are interleaved.

[Figure 8 about here.]

Recall the R-Tree organization and data structures illustrated in Figure 1. Figure 8 shows the modified data structures that support deferred execution

of tree operations. Query and delete operations are deferred, whereas inserts bypass the queue mechanism. We extend certain nodes within the R-Tree with pointers to fixed-length, deferred-operation queues. Multiple nodes along a path from the root to a leaf may be augmented with queues. We associate a *generation* with the entire tree, every node within the tree, and each of the tree operations. The tree’s generation is incremented on each insert, and the newly inserted node is assigned the tree’s latest generation value. The deferred operations are assigned the tree’s generation value when they are added to the system. By supporting an ordering among deferred operations, and by making sure that deferred operations do not consider nodes of a newer generation, we maintain the consistency of the tree structure, and queries produce the same output as in the baseline implementation.

A query or delete operation starting at the root is allowed to execute normally until it encounters a node with a non-null queue. If the queue has room, the operation is enqueued. Otherwise the queue is *flushed*; all operations contained in the queue are executed. Queues can be flushed for other reasons, such as timeouts, as well. During the flush, deferred operations might encounter more queue-augmented nodes, in which case the operations might again be enqueued. Although this scheme may appear wasteful, the significant improvement in cache performance more than offsets the cost of deferral. The throughput of the tree structure increases on average, but at the cost of increased latency of individual operations. Note that our approach may be inappropriate if queries must be processed in real time.

Table V shows that the overall speedup obtained for the test input, *dm23.in*, is 1.87. The average tree size contains about fifty thousand nodes spanning approximately 16 MB. Cache hit ratios improve significantly, from 95.4% to 97.1% for the L1D cache, and from 49.6% to 77.9% for the L2 cache. We observe some improvement in TLB performance, but the effect is small. The overhead incurred from the queue structure is significant: about 10.5% of the original memory stall time. However, the 60% reduction in stall time more than offsets this overhead.

We use our synthetic input to compare combinations of deferred execution and software clustering techniques, and to investigate how queue size affects R-Tree performance. This input yields an R-Tree structure that is large and relatively static. For regrouping, we initially assign 256-entry queues to nodes at levels two, five, and eight. (For the given input the tree is not expected to grow beyond a height of eight.) Table VI presents the perfex-estimated execution times, cache and TLB miss-handling times, and actual execution times for different optimization techniques. As noted above, the perfex numbers are optimistic, but for consistency we use these numbers to determine the gain and the overheads of regrouping. We provide wall-clock times to validate the perfex estimates. Throughput in Table VI is in terms of the number of queries executed per second. This throughput increases from 21.4 to 59.6, corresponding to an overall application speedup of 2.68. Again, combining regrouping and clustering yields the best results.

[Table 6 about here.]

In the baseline application, cache and TLB misses account for about 95% of the execution time. The theoretically achievable speedup should therefore be about 19. The maximum we observe is 4.38, due to the high overheads incurred in implementing deferral. Note that intra-node clustering is quite effective in improving both cache and TLB performance. We observe a speedup of 1.85, and most of the improvement is due to increased spatial locality from colocating nodes. For inter-node clustering, maintaining node placement invariants requires significant overhead, which we estimate to be about 7% of the overall stall time. These costs limit the speedup to 1.51, which is smaller than for the previous clustering technique, but still significant. In both cases more than 50% of the time is spent servicing cache and TLB misses. In contrast, Computation Regrouping via deferred execution eliminates about 91% of memory stall time and about 40% of TLB miss time, and its computational overhead is comparable to that of clustering. Note that clustering and regrouping can complement each other, reducing the execution time be-

yond that achievable by either in isolation. The main improvement is from reducing TLB misses, but this comes at the cost significant computational overhead — about 12.5% of overall stall time.

The exact placement of the queues is less important than queue length, which has direct impact on the latency of the individual operations. Also, we find that there is a threshold for the parameters beyond which the gain is negligible.

5 Locality Grouping

Computation regrouping optimizations, as presented in the previous sections, are hand-coded. The applicability and usefulness of the approach can be significantly improved if the optimization process can be automated at least to some extent. In this section, we present an abstraction that is simple, and efficient, and allows for flexible implementations.

We expect compiler-based techniques for automatic detecting and optimizing of code using computation regrouping to be difficult and time consuming. Ding and Kennedy ⁽⁹⁾ report that in one instance in which their compiler employs multilevel fusion (an optimization similar to computation regrouping), it takes up to an hour to optimize a particular benchmark when three-level fusion is enabled. Our benchmarks have significantly more complex control and data structures than those presented in their paper, and are likely to be much more difficult to analyze. We therefore expect that developing techniques to automate regrouping will remain an ongoing research topic. We instead consider semiautomatic techniques that assume some support from the application developer. Our experience suggests a run-time library-based approach similar to WorkCrews ⁽²⁶⁾ could work well.

The key insight is that computation regrouping translates into identifying fine-grained thread-like computations and efficiently scheduling these computations for increased cache locality. Extending lightweight threading and scheduling abstractions with locality information may achieve some of

the benefits of computation regrouping. Therefore, we propose allowing programmers to identify regroupable tasks, and having the run-time system perform task scheduling to exploit this information.

Deferrable operations, which we call *tasks*, are classified into *locality groups* (LGs) based on the sets of data objects they access. Thus, tasks within a single locality group should have high spatial locality. The user is presented with a library interface to create locality groups, insert tasks into these groups, and finally force the execution of the deferred tasks in any group. We first describe the interface in more detail, and then discuss the performance implications of semiautomating computation regrouping in this way.

[Figure 9 about here.]

5.1 Description

Figure 9 presents the abstraction we provide to the user. *CreateLG* and *DeleteLG* create and destroy locality groups. The user identifies computations that can be deferred, wraps them into tasks by specifying a call-back function, and inserts them into an appropriate locality group using the *AddToLG* function. The library run-time is free to execute the tasks at anytime. It is difficult for the run-time library to determine automatically how long to defer tasks or how many tasks it should execute at once to best exploit locality.

Independent scheduling decisions by the LG run-time system might result in significant overhead. Therefore, we feel that the decision of when to execute the deferred tasks within a locality group is best left to the application. The application can force the execution using the *FlushLG* function. Our current implementation executes tasks only when they are flushed. This model leaves some flexibility in the implementation of the library run-time. The run-time library can decide the order of execution, the processors on

which the flushed tasks will be executed, and the number of threads to use. This scheme is similar to bin scheduling ⁽²⁷⁾.

5.2 Performance

[Figure 10 about here.]

Figure 10 presents a simple implementation model for locality grouping. Here, LGs are implemented as doubly linked lists, which allows fast insertion and deletion. No LG operation involves a search or lookup. The tasks contained within any LG can be implemented as a linked list or as a fixed length array. In most cases we choose the array-based implementation for efficiency. We now re-code a subset of benchmarks using the locality grouping abstraction. Table VII presents our results. For Health and FFT, we note that the abstraction is a natural fit, but adds nontrivial overhead compared to hand-coding. For applications with indirect accesses such as IRREG, the abstraction is used trivially, and therefore does not add any significant value or overhead. We could not optimize R-Tree or CUDD using locality grouping, since their internal structure does not match the assumptions regarding parallelism. We next describe the modifications we make to individual benchmarks.

[Table 7 about here.]

In FFT, we first unroll the *executor-many* function, which translates the long column walk into a series of calls to *codelets*. We then intercept calls to the codelets, wrap them into task structures, and insert them into appropriate locality groups. We identify the locality group based on the input to the codelet — a fixed offset in the column. We maintain two kinds of locality groups, one each for the *notwiddle* and *inverse notwiddle* codelet calls. After every eight column walks, both *twiddle* and *notwiddle* groups are flushed. This is reasonable since no future column walks share data accessed by the various locality groups, and the FFTW algorithm requires the results for

further processing. As Table VII shows, the computational overhead of using the abstraction is about 6.7%. This can be attributed primarily to the cost of creating and using the task structures. We can reduce this by using a linked-list-based implementation of the locality group, and reusing the task structures. However, optimization of the locality group abstraction is outside the scope of this paper.

In Health, we associate a locality group with each node in the quad-tree. Each update cycle of the waiting list is wrapped into a task structure and added to the locality group associated with the node. When the task array (assuming an array-based implementation of locality groups) fills up, or when a modification to the list is necessary, the locality group is flushed. The computational overhead of LG is high at about 46%. The key reason for this high overhead is the presence of a large number of the nodes with short lists, for which the cost of the cache misses incurred in updating the waiting list is significantly less than the cost of deferring. A simple and effective modification is to dynamically enable deferring of update tasks at any node only when the length of the waiting list at that node exceeds some threshold. Such enhancements are beyond the scope of this paper.

As discussed in Section 4, we optimize the EM3D, IRREG and Ray Tracing applications using the filtered execution technique. Expressing the optimization in terms of the locality groups was simple and computationally inexpensive. The set of computations accessing a specific region of the data structure could be efficiently captured by identifying the beginning and the end of the region they access. Individually listing the computations is too expensive. Since the number of regions is typically small, the number of locality groups is small. Further, since the computations can be captured by only one task, the overhead of using the abstraction is minimal as shown in Table VII.

5.3 Summary

Locality grouping aims to impose a structure on the execution of regrouping. It leaves the task of identifying computations for regrouping to the user. Our prototype is a simple scheduling abstraction extended with locality information, and allows for fine-grained control by the application. The extra computational overhead due to the interface caused a slowdown varying between 0.1% and 46% over the nonlocality grouped versions. Although locality grouping can model only a subset of instances where computation regrouping can be used, the added structure in the processing can potentially help devise novel compiler or run-time-based techniques.

6 Related Work

Application memory performance has been the focus of significant research. We limit our discussion here to existing software approaches for increasing memory performance. These may be classified as cache-conscious algorithms, compiler optimizations, or application-level techniques.

Algorithmic Approaches. Cache-conscious algorithms modify an application’s control flow, and sometimes the data structure layout, based on an understanding of the application’s interactions with memory subsystem. Both theoretically provable ^(28, 29) as well as practical algorithms ^(30, 31, 32) have been developed. The provable algorithms, also called cache oblivious algorithms, have performance characteristics that are independent of the cache configuration whereas rest of the algorithms are usually parameterized over properties of the cache such as the overall size, associativity, and line size. While these algorithms usually do not modify the fundamental I/O complexity of the application, they modify the constant factors involved. This results in significant application performance improvement in practice. While approaches have been developed for specific applications such as sorting ⁽³¹⁾, query processing ⁽³²⁾, and matrix multiplication ⁽³⁰⁾, it has proven hard to design cache conscious algorithms that have wide applicability. The existence

of multiple dominant access patterns within a single program complicates algorithm design significantly. As a result, cache-conscious algorithms are few in number, and are usually domain-specific. Computation regrouping also requires substantial understanding of access patterns, but is a more generic approach. Changes required by regrouping are low-level, but fairly architecture-independent. Regrouping does not require radical application changes, and therefore we believe that regrouping can be applied in more scenarios. We demonstrate this by optimizing a variety of applications.

Compiler Approaches. Compiler-based restructuring techniques for improving spatial and temporal locality are well known ^(3, 4, 5, 33, 34, 10, 35). They are usually applied to regular data structures within loops and to nearly perfect loop nests, and thus are driven by analytic models of loop costs ⁽³⁶⁾. For instance, Kodukula, Ahmed, and Pingali present a comprehensive approach to *data shackling* for regular arrays ⁽³³⁾. They, like we, strive to restructure computation to increase temporal data locality. They focus specifically on the problem domain of dense numerical linear algebra, and have developed rigorous methods for automatically transforming the original source code. In contrast, our approach applies to a broader class of applications and data structures, but making it automatic instead of ad hoc is part of future work.

Bucket-tiling ⁽¹⁵⁾ is an approach in which references are assigned buckets, and the control and data structures are modified to exploit the reference classification. This modification uses a combination of techniques such as array permutation, data remapping and loop generation. The method presented is automatic in its detection of optimization opportunities, and selection of the permutation and remapping functions. The filtered execution is similar in operation to bucket-tiling. The notion of sliding window is similar to the bucket, but is significantly simplified in some ways to enable application of the method to large, complex, dynamic datastructures with little effort. First, filtered execution does not involve any data layout modification or any other kind of preparatory operations. Second, it depends on the user for selection

of the datastructure over which the windowing operation must be performed and correctness issues that may arise. Third, the optimization may result in wasteful, multiple scans of the datastructure as indicated by the increase in the instruction count. The selection of the bucket to which a reference belongs may be evaluated dynamically in the innermost loop. Fourth, the sliding window may not be based on the address range. We depend only on being able to partition the datastructure along some dimension relevant to the application. Bucket-tiling's behavior is similar to filtered execution in that it has significant overhead due to permutation and remapping operations and gives overall performance improvement only for large enough data sizes. Bucket-tiling is an automated technique and therefore immediately useful as opposed to computation regrouping which requires substantial understanding of the application by the user. Automating regrouping (even for the special case of filtered execution) will require sophisticated mode detection techniques such as that developed in conjunction with bucket-tiling ⁽³⁷⁾.

In some approaches ^(38, 39, 40) the compiler takes as input an abstract specification of the operations to be performed and chooses an appropriate datastructure to implement the specification. A specification or model for complex datastructures and high level access patterns could help identify control dependencies, locality issues and regrouping opportunities.

Complex structures or access patterns are not usually considered for compiler-based restructuring due to the difficulty in deriving accurate analytic models. Further, it is not clear how these diverse techniques can be combined efficiently. Computation regrouping can be viewed as a heuristic-based, application-level variation of traditional blocking algorithms that cannot be directly applied to complex data structures. Regrouping requires modest changes to control and data structures. Performance improvements from regrouping should be smaller than from compiler-based restructuring techniques, because of the expected higher control overhead. Profitability analysis based on cost models would be helpful to determine optimization

parameters, but we do not expect them to be as detailed or accurate as in previous work.

Others have taken intuitions similar to those underlying our restructuring approach and have applied them in other arenas. For instance, to reduce accesses to non-local data, Rogers and Pingali ⁽⁸⁾ develop a multiprocessor compiler approach in which the application programmer specifies the data-structure layout among the processors, and then describe how the compiler can generate code consistent with this specification, along with some possible optimizations. Our Computation Regrouping can be viewed as a variation in which spatial decomposition appropriate for the multiprocessors is used, but with the multiple processors simulated serially in time on a uniprocessor. We extend this previous work by showing that the approach is useful for complex data structures and access patterns. In one case, IRREG, the optimized code for both the approaches has structural similarity. Although we started at a different point in the application and machine configuration space, the approaches are similar in that as the memory cost increases, even main memory accesses start looking like remote memory accesses of distributed shared memory machines. Using computation regrouping on multiprocessors is possible, but is likely to require solving more correctness issues and to incur higher costs for integrating partial results.

Prefetching hides high data access latencies for data with poor locality ^(2, 6, 41), and can significantly improve performance for some kinds of programs. The effectiveness of prefetching on some complex data structures is limited because prefetching does not address the fundamental reason why the data accesses are expensive, *i.e.*, large temporal separations between successive accesses to a given object. Computation Regrouping addresses that problem by modifying the application’s control flow, and thus can complement prefetching.

Application Approaches. Application designers can use data restructuring for complex structures. This can be useful when the compiler is unable to identify appropriate legal and profitable transformations. Some tech-

niques, *e.g.*, clustering and coloring ⁽¹¹⁾, take this approach, considering more complex linked data structures like trees and graphs for memory optimizations ^(11, 12, 13). Spatial blocking is effective for B-Trees and variants ^(12, 13). Improving spatial locality is useful, but has limited impact when the access patterns are complicated. Computation regrouping is similar to these approaches in terms of the level of user involvement, but complements them with modifications to enhance temporal locality.

Computation Regrouping. Some types of computation regrouping have been considered in other contexts. A new loop tiling technique for a class of imperfect nests, for example, tries to take advantage of accesses spread across multiple time-steps ⁽⁴²⁾. In compiler-based, reuse-driven loop fusion and multi-level data regrouping work by Ding and Kennedy ⁽⁹⁾, the authors identify opportunities to fuse array references across loop nests. In both cases, it is not clear as how the approach will scale to handle more complex data structures. Here we extend previous work by considering applications with complex data structures and access patterns.

A queue-enhanced R-Tree called a *Buffer Tree* was previously proposed in the context of external memory algorithms ^(43, 44). However, unlike earlier studies that focus on improving I/O performance, in this paper we focus on a memory-resident variation of R-Trees, and we identify the queuing extension as a specific instance of computation regrouping.

7 Conclusions

The growing CPU-memory speed gap causes many applications to be increasingly limited by memory performance. Few generic techniques exist to optimize the memory behavior of complex applications, and most existing compiler-based memory optimizations cannot be directly applied to programs with complex, pointer-rich data structures.

To address these problems, we present a software approach that trades extra computation for reduced memory costs. *Computation Regrouping* moves

computations accessing the same data closer together in time, which can significantly improve temporal locality and thus performance. In our experience, the control and data structure changes required are small compared to overall code sizes. We consider the modest programmer overhead required to identify appropriate logical operations and the execution overhead of the modified application to be a good trade for greatly increased temporal locality. We present four techniques to realize the regrouping approach, and demonstrate that they successfully eliminate a significant fraction of memory stall time. Our hand-coded optimizations improve performance by a factor of 1.26 to 3.03 on a variety of applications with a range of sizes, access patterns, and problem domains. Our initial results are promising, and we believe further research in this direction to be warranted.

We are currently exploring a runtime library-based approach to extend work crews ⁽²⁶⁾ with locality information ⁽¹⁾. Preliminary results are encouraging, but much work remains in identifying an abstraction suitable for compiler-based analysis.

Our conclusion from this investigation is that program-level access patterns, in addition to low-level access patterns, can significantly impact application performance by their interactions with the cache-based memory model of modern processors. Understanding the nature of these interactions can help us to design techniques that improve application memory performance. Our logical operations-based characterization and optimizations designed to use such techniques reflect our current understanding of these program-level patterns. Going forward, the hard problems to be solved include identifying exact or statistical schemes by which non-local effects of memory accesses can be modeled efficiently, and automating the process of using these models.

ACKNOWLEDGMENTS

The authors thank the anonymous referees for their valuable comments. Keshav Pingali's input has been invaluable, as has that of the other members

of the Impulse Adaptable Memory Controller project. We also thank Robert Braden supporting Venkata Pingali during the final stages of publication of this work.

References

- [1] V. Pingali, “Memory performance of complex data structures: Characterization and optimization,” Master’s thesis, University of Utah, August 2001.
- [2] D. Callahan, K. Kennedy, and A. Porterfield, “Software prefetching,” in *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 40–52, April 1991.
- [3] S. Carr, K. McKinley, and C.-W. Tseng, “Compiler Optimizations for Improving Data Locality,” in *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 252–262, October 1994.
- [4] H. Han and C.-W. Tseng, “Improving locality for adaptive irregular scientific codes,” Technical Report CS-TR-4039, University of Maryland, College Park, September 1999.
- [5] M. T. Kandemir, A. N. Choudhary, J. Ramanujam, and P. Banerjee, “Improving locality using loop and data transformations in an integrated framework,” in *International Symposium on Microarchitecture*, pp. 285–297, November–December 1998.
- [6] M. Karlsson, F. Dahlgren, and P. Stenstrom, “A Prefetching Technique for Irregular Accesses to Linked Data Structures,” in *Proceedings of the Sixth Annual Symposium on High Performance Computer Architecture*, pp. 206–217, January 2000.
- [7] I. Kodukula and K. Pingali, “Data-centric transformations for locality enhancement,” *International Journal of Parallel Programming*, vol. 29, pp. 319–364, June 2001.
- [8] A. Rogers and K. Pingali, “Process decomposition through locality of reference,” in *Proceedings of the 1989 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 69–80, June 1989.
- [9] C. Ding and K. Kennedy, “Improving Effective Bandwidth through Compiler Enhancement of Global Cache Reuse,” in *2001 International Parallel and Distributed Processing Symposium*, April 2001.

- [10] S. Leung and J. Zahorjan, "Optimizing Data Locality by Array Restructuring," Tech. Rep. UW-CSE-95-09-01, University of Washington Dept. of Computer Science and Engineering, September 1995.
- [11] T. M. Chilimbi, M. D. Hill, and J. R. Larus, "Cache-Conscious Structure Layout," in *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1–12, May 1999.
- [12] J. Rao and K. A. Ross, "Cache Conscious Indexing for Decision-Support in Main Memory," in *Proceedings of the 25th VLDB Conference*, pp. 78–89, 1999.
- [13] J. Rao and K. A. Ross, "Making B+-Trees Cache Conscious in Main Memory," in *Proceedings of the 26th VLDB Conference*, pp. 475–486, 2000.
- [14] W. Abu-Sufah, D. Kuck, and D. Lawrie, "Automatic program transformations for virtual memory computers," in *Proceedings of the 1979 National Computer Conference*, pp. 969–974, June 1979.
- [15] N. Mitchell, L. Carter, and J. Ferrante, "Localizing non-affine array references," in *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pp. 192–202, October 1999.
- [16] M. Carlisle, A. Rogers, J. Reppy, and L. Hendren, "Early experiences with olden," in *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, pp. 1–20, August 1993.
- [17] Silicon Graphics Inc., *SpeedShop User's Guide*. 1996.
- [18] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, "Performance Analysis Using the MIPS R10000 Performance Counters," in *Proceedings of Supercomputing '96*, November 1996.
- [19] A. Guttmann, "R-Trees : A Dynamic Index Structure for Spatial Searching," in *Proceedings of the 1984 International Conference on Management of Data*, pp. 47–57, August 1984.
- [20] J. W. Manke and J. Wu, *Data-Intensive System Benchmark Suite Analysis and Specification*. Atlantic Aerospace Electronics Corp., June 1999.
- [21] H. Han and C. Tseng, "Improving Compiler and Run-Time Support for Irregular Reductions," in *Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing*, (Chapel Hill, NC), August 1998.

- [22] M. Frigo and S. Johnson, “FFTW: An adaptive software architecture for the FFT,” in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. 1381–1384, May 1998.
- [23] F. Somenzi, “CUDD: CU Decision Diagram Package Release 2.3.1,” 2001.
- [24] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick, “Parallel Programming in Split-C,” in *Proceedings of Supercomputing '93*, pp. 262–273, November 1993.
- [25] A. Appel, J. Ellis, and K. Li, “Real-time concurrent collection on stock multiprocessors,” in *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 11–20, June 1988.
- [26] E. S. Roberts and M. T. Vandevoorde, “WorkCrews: An Abstraction for Controlling Parallelism,” Tech. Rep. SRC-042, Digital Systems Research Center, April 1989.
- [27] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li, “Thread Scheduling for Cache Locality,” in *Proceedings of the 7th Conference on Architectural Support for Programming Languages and Systems*, (Cambridge, MA), pp. 60–73, October 1996.
- [28] M. A. Bender, E. D. Demaine, and M. Farach-Colton, “Cache-Oblivious B-Trees,” in *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pp. 399–409, November 2000.
- [29] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran, “Cache-Oblivious Algorithms,” in *40th Annual Symposium on Foundations of Computer Science*, pp. 285–297, October 1999.
- [30] S. Chatterjee, A. Lebeck, P. Patnala, and M. Thottethodi, “Recursive Array Layouts and Fast Matrix Multiplication,” in *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 222–231, June 1999.
- [31] A. G. LaMarca, *Caches and Algorithms*. PhD thesis, University of Washington, 1996.
- [32] A. Shatdal, C. Kant, and J. Naughton, “Cache Conscious Algorithms for Relational Query Processing,” in *Proceedings of the 20th VLDB Conference*, pp. 510–521, September 1994.

- [33] I. Kodukula, N. Ahmed, and K. Pingali, “Data-Centric Multi-level Blocking,” in *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 346–357, June 1997.
- [34] M. S. Lam, E. E. Rothberg, and M. E. Wolf, “The cache performance and optimizations of blocked algorithms,” in *Proceedings of the 4th ASPLOS*, pp. 63–74, April 1991.
- [35] D. N. Truong, F. Bodin, and A. Sez nec, “Improving Cache Behavior of Dynamically Allocated Data Structures,” in *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pp. 322–329, October 1998.
- [36] S. Ghosh, M. Martonosi, and S. Malik, “Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity,” in *Architectural Support for Programming Languages and Operating Systems*, pp. 228–239, October 1998.
- [37] N. Mitchell, *Guiding Program Transformations with Modal Performance Model*. PhD thesis, University of California, San Diego, August 2000.
- [38] A. J. C. Bik and H. A. G. Wijshoff, “On automatic data structure selection and code generation for sparse computations,” in *1993 Workshop on Languages and Compilers for Parallel Computing*, no. 768, (Portland, Ore.), pp. 57–75, Berlin: Springer Verlag, 1993.
- [39] N. Mateev, K. Pingali, P. Stodghill, and V. Kotlyar, “Next-generation generic programming and its application to sparse matrix computations,” in *International Conference on Supercomputing*, pp. 88–99, 2000.
- [40] V. Menon and K. Pingali, “High-level semantic optimization of numerical codes,” in *Proceedings of the 1999 International Conference on Supercomputing*, pp. 434–443, 1999.
- [41] C.-K. Luk and T. C. Mowry, “Compiler-Based Prefetching for Recursive Data Structure,” in *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 222–233, October 1996.
- [42] Y. Song and Z. Li, “New Tiling Techniques to Improve Cache Temporal Locality,” in *Proceedings of the SIGPLAN ’99 Conference on Programming Language Design and Implementation*, pp. 215–228, May 1999.
- [43] L. Arge, “The Buffer Tree : A New Technique for Optimal I/O-Algorithms,” in *Fourth Workshop on Algorithms and Data Structures*, pp. 334–345, August 1995.

- [44] L. Arge, K. Hinrichs, J. Vahrenhold, and J. S. Vitter, “Efficient Bulk Operations on Dynamic R-Trees,” in *1st Workshop on Algorithm Engineering and Experimentation*, pp. 328–348, January 1999.

List of Tables

I	Benchmark characteristics	40
II	Logical operations for benchmarks	41
III	Characteristics of regrouping techniques	42
IV	Regrouping results	43
V	Perfex-estimated performance statistics	44
VI	R-Tree performance for combinations of Computation Regrouping and clustering	45
VII	Summary of the performance improvements with locality grouping and hand-coding.	46

Benchmark	Lines of Code	Sample Input	Hit Ratio (%)		Stall Time (%)	Execution Time Scaling
			L1D	L2		
R-Tree	7K	dm23.in	95.4	49.6	71	linearly with tree size
IRREG	300	MOL2, 40 iterations	84.5	54.5	74	linearly with number of edges
Ray Trace	5K	balls 256×256	95.8	70.1	58	linearly with viewing window size and input scene
FFTW	35K	$10K \times 32 \times 32$	92.4	60.0	65	linearly with dimensions
CUDD	60K	C3540.blif	84.1	45.9	69	scaling properties unclear
EM3D	700	128K, 10, 1	85.4	47.9	66	linearly with nodes, sub-linearly with degree
Health	600	6, 500	71.9	68.6	70	exponentially with depth

Table I: Benchmark characteristics

Benchmark	Critical Working Set	Estimated Threshold	Logical Operation
R-Tree	tree size	15K nodes	one tree operation (insert, delete, or query)
Ray Trace	3D scene	—	scan of the input scene by one ray
FFTW	cache line-size \times dimension	Y or Z dim $>$ 16K	one column walk of a 3D array
IRREG	array size	256K	group of accesses to a set of remote nodes
CUDD	$2 \times$ hash-table size	—	one variable swap
EM3D	$(4 \times \text{degree} + 12) \times \text{nodes}$	40K nodes (degree=10)	group of accesses to a set of remote nodes
Health	quad-tree size + lists size	3K (listsize=5)	simulation of one time step

Table II: Logical operations for benchmarks

Technique	Data Structure Modifications	Control Structure Modifications	Generality
Early Execution	mid	high	mid
Deferred Execution	high	high	high
Computation Summation	low	low	low
Filtered Execution	low	mid	low

Table III: Characteristics of regrouping techniques

Bench- mark	Access Pattern	Technique	Code		Issued Instructions		Graduated Instructions	
			Total	Change	Base	Restructured	Base	Restructured
R-Tree	Pointer Chasing	Deferred Execution with Queues	7K	600	102.5B	103.9B	89.4B	92.7B
IRREG	Indirect Accesses	Filtered Execution	300	40	15.7B	15.9B	8.2B	13.3B
Ray Trace	Strided Accesses Pointer Chasing	Filtered Execution	5K	300	108.5B	108.5B	90.8B	96.7B
FFTW	Strided Accesses	Early Execution	35K	1.5K	6.3B	5.7B	5.11B	5.07B
CUDD	Pointer Chasing	Combination: Early Deferred Execution	60K	120	13.6B	13.9B	6.9B	6.9B
EM3D	Indirect Accesses Pointer Chasing	Computation Summation Filtered Execution	700	200	646M	617M	428M	542M
Health	Pointer Chasing	Computation Summation	600	30	1.6B	0.43B	0.83B	0.39B

Table IV: Regrouping results

Bench- mark	Input	Hit Ratio (%)				Time (sec)				Saved Time (% stall)	Over- head time)	Speedup
		Base		Restructured		Base		Restructured				
		L1D	L2	L1D	L2	Total	Stall	Total	Stall			
R-Tree	dm23.in	95.4	49.6	97.1	77.9	1312	1194	718	474	60	10.5	1.87
IRREG	MOL2	84.5	54.5	84.9	84.3	253	238	145	104	56.3	10.9	1.74
Ray Trace	balls 256 × 256	95.8	70.1	96.1	99.6	905	531	457	81	84.7	0.3	1.98
FFTW	10K × 32 × 32	92.4	60.0	92.3	93.3	111	82	44	20	—	—	2.53
CUDD	C3540. blif	84.1	45.9	88.8	46.0	307	294	241	217	26	3.7	1.26
EM3D	128K, 10, 1	85.4	47.9	86.6	71.8	13.4	11.24	9.4	6.7	40	4.6	1.43
Health	6, 500	71.9	68.6	85.2	46.8	46	46	15	16	—	—	3.03

Table V: Perfex-estimated performance statistics

Optimization	Clock Time (sec)	Estimated Time (sec)					Saved Time (% stall time)	Over-head	Thruput (queries/sec)	Speedup
		Run	L1D	L2	TLB	Total				
Baseline	4687	4214	261	3254	474	3989	0	0	21.3	1.0
Intra-node Clustering	2523	2355	178	1492	140	1810	55	16	39.6	1.85
Inter-node Clustering	3090	2950	175	1472	94	1741	56	25	32.4	1.51
Deferred Execution	1347	1119	98	213	283	594	85	7.5	74.2	3.48
Deferred + Intra-node	1103	1023	54	171	84	309	92.2	12.2	90.7	4.25
Deferred + Inter-node	1068	1004	51	169	54	274	93	12.6	93.6	4.38

Table VI: R-Tree performance for combinations of Computation Regrouping and clustering

Benchmark time	Original	Locality Grouped		Hand-coded		slowdown due to grouping
		time	speedup	time	speedup	
EM3D	13.4	9.4	1.43	9.9	1.35	5.6%
RayTrace	905	457s	1.98	460s		0.1%
IRREG	205s	133s	1.54	139s	1.47	4.6%
Health	42s	14s	3.0	26s	1.61	46%
FFTW	94s	42s	2.23	45s	2.08	6.7%

Table VII: Summary of the performance improvements with locality grouping and hand-coding.

List of Figures

1	R-Tree structure	48
2	IRREG main loop	49
3	R-Tree performance with about 70K nodes	50
4	Original code for indirect access	51
5	Filtered execution code for indirect access	52
6	Filtered execution code for IRREG	53
7	Speedup vs. window size for IRREG	54
8	Modified structures to support regrouping via queuing	55
9	Locality grouping abstraction.	56
10	Implementation model for computation regrouping.	57

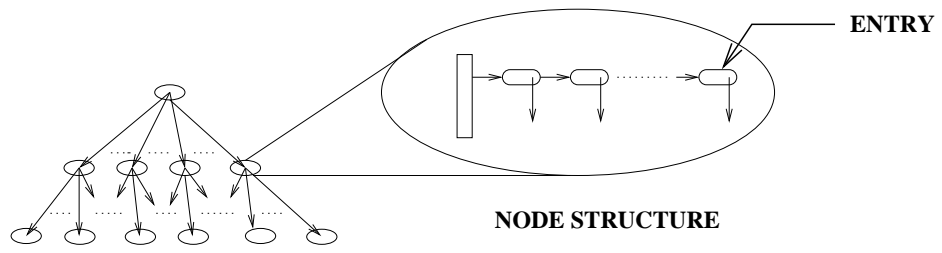
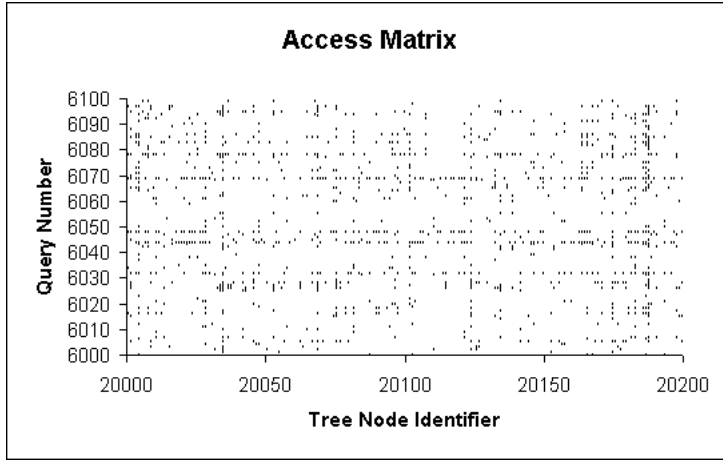


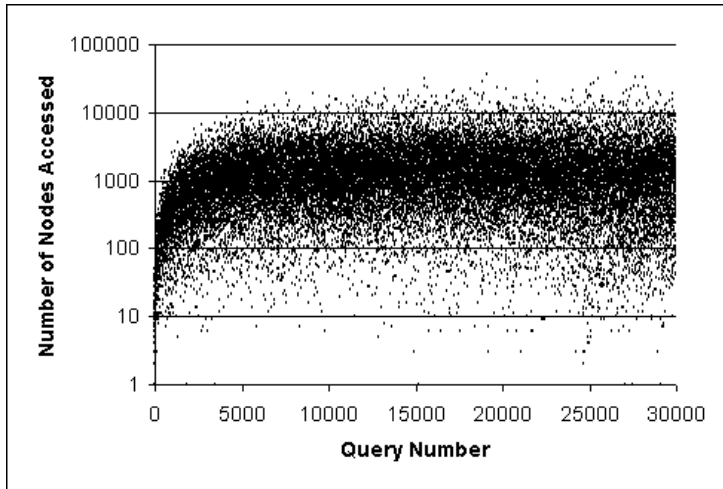
Figure 1: R-Tree structure

```
for (i = 0; i < n_edges ; i++) {  
    n1 = left[i] - 1;  
    n2 = right[i] - 1;  
    rr = (x[n1] - x[n2]) * 0.25;  
    y[n1] = y[n1] + rr;  
    y[n2] = y[n2] - rr;  
}
```

Figure 2: IRREG main loop



(a) Access matrix for queries versus nodes



(b) Distribution of the number of nodes accessed by queries

Figure 3: R-Tree performance with about 70K nodes

```
for (i = 0; i < max; i++) {  
    sum += A[ix[i]];  
}
```

Figure 4: Original code for indirect access

```
#define SHIFT(x) (x + window_size)
...
iters = ...
window_size = N/iters;
for (win = start; iters > 0 ; win = SHIFT(win), iters--) {
    for (i = 0; i < imax; i++) {
        if (ix[i] >= win && ix[i] < (win + window_size))
            sum += A[ix[i]];
    }
}
```

Figure 5: Filtered execution code for indirect access

```
for ( k = 0 ; k < n_nodes; k += blocksize ) {
    int n1, n2;
    for ( i = 0; i < n_edges; i++) {
        n1 = left(i) - 1;
        n2 = right(i) - 1;
        if ( n2 < k || n2 >= (k + blocksize ))|
            continue;
        rr = (x[n1] - x[n2]) * 0.25;
        y[n1] = y[n1] + rr;
        y[n2] = y[n2] - rr;
    }
}
```

Figure 6: Filtered execution code for IRREG

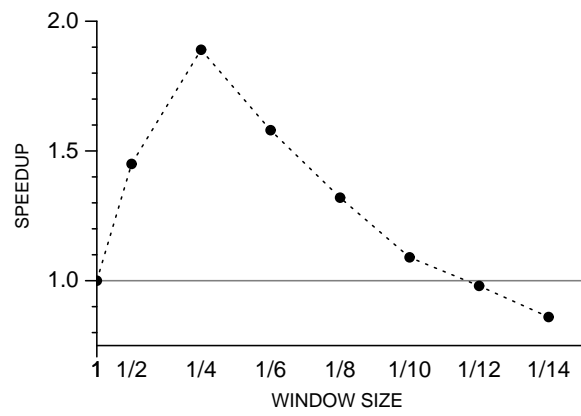


Figure 7: Speedup vs. window size for IRREG

```

typedef struct {
    Int node_id;
    Int node_level;
    Entry *entries;
    Int propagate;
    Queue *q;
} Node;

typedef struct Entry {
    union {
        Node *node;
        DataObject *obj;
    } child;
    Key key;
    struct Entry *next;
    Int generation;
} Entry;

typedef struct {
    struct {
        Float t;
        Float x;
        Float y;
        Float z;
    } lower, upper;
} Key;

```

Figure 8: Modified structures to support regrouping via queuing

1. CreateLG() : create a locality group
2. DeleteLG(*lg*) : delete a locality group
3. AddToLG(*lg*, *proc*, *data*) : add a new task to the locality group *lg* which calls function *proc* with *data* as it argument.
4. FlushLG(*lg*) : wait till the execution of the tasks contained in *lg* are completed.

Figure 9: Locality grouping abstraction.

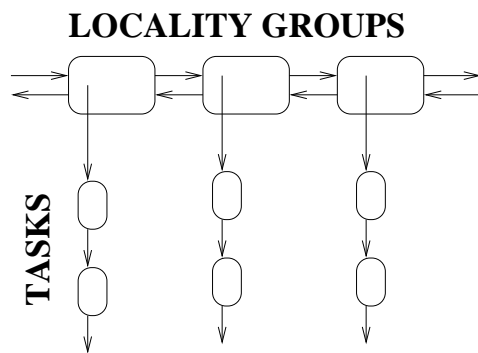


Figure 10: Implementation model for computation regrouping.