

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 may be reasonable 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 a set of seven, memory-intensive benchmarks; we describe *Computation Regrouping*, a general, source-level approach to improving the overall performance of these applications by improving temporal locality to reduce cache and TLB miss ratios (and thus memory stall times); and we demonstrate significant performance improvements from applying Computation Regrouping to our suite of seven benchmarks. With Computation Regrouping, we observe an average speedup of 1.97, 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 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 can account for 58-99% of the execution time of applications in the domains of signal processing, databases, CAD tools, and high performance graphics ⁽¹⁾. Worse, the contribution as a fraction of total execution time increases as the performance gap between processors and memory continues to grow. Reducing this memory stall time improves processor resource utilization, and will allow an application's performance to scale with improvements in processor speed and technology.

Many hardware, software, and hybrid approaches have been investigated for both regular and complex, pointer-based data structures ^(2, 3, 4, 5, 6, 7, 8). Effective optimization methods exist for modest-sized applications with regular and even irregular data structures ^(3, 9, 10, 8), but the memory performance issues of large pointer-based applications have received relatively less attention. Most approaches try to improve spatial locality through controlled data layouts ^(11, 12, 13). However, 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 contrast, our approach strives to improve temporal locality, and is complementary to most spatial locality and local optimizations. Section 6 discusses related work in more detail.

In this paper, we first identify reasons for poor memory performance by qualitatively characterizing benchmark access patterns. We then propose *Computation Regrouping*, which extend application control and data structures to modify the execution order of computations. This restructuring reduces the impact of high memory latency by grouping computations based on

the data objects they access. The “batched” computations can be executed with higher temporal data locality, thereby improving cache performance. Applications that can benefit from this work are those for which processor-memory bottlenecks account for a large portion of the runtime. We find Computation Regrouping to be quite effective on our seven memory-bound benchmarks, which are of varying complexity, and which span a number of application domains. For these initial results, optimizations are implemented by hand. The additional programmer effort appears worthwhile for programs that are difficult to optimize by compiler methods: we observe an average application speedup of 1.97, with individual speedups ranging from 1.26 to 3.03. These speedups result directly from memory stall time reductions of 26-85%.

Section 2 describes in detail our benchmarks and their high-level characteristics. Section 3 presents observations relating the memory performance to the program structure. Section 4 discusses our proposed transformations, and Section 5 presents results obtained on applying these transformations. Section 6 discusses related approaches. Section 7 presents a novel way of looking the regrouping optimization itself. Section 8 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

We select our benchmarks based on the memory performance of publicly available, reference implementations on large inputs and consider only applications spending 25% or more of their execution time servicing secondary data cache and TLB misses. We use carefully optimized versions of each benchmark as baselines and implement function-call and layout-related optimizations where appropriate (for instance, we eliminate MST ⁽¹⁴⁾ from consideration, since it becomes compute-bound after applying such changes). Applications not meeting the 25% threshold are likely to continue to be

compute-bound on next-generation systems, and the small potential performance gains do not justify either the programming effort or run-time overheads of our approach. Many common architecture and compiler benchmarks are therefore excluded. For instance, floating-point costs are sufficiently high to hide the poor memory performance of Barnes-Hut ⁽¹⁴⁾; as a result, it could potentially be memory-bound on a different processor configuration, we exclude this benchmark.

Section 5 describes the SGI Power Onyx system on which we performed all our experiments, including those for identifying an appropriate benchmark suite. We experimented with a range of inputs to determine the sensitivity of each application to input parameters, and we examined 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.

[Table 1 about here.]

The seven benchmarks cover a range of application domains (scientific codes, hardware design tools, graphics, and databases), code sizes, and complexity. Table I presents cache hit ratios for sample inputs and shows how the execution times scales with input sizes. The table also presents the estimated memory stall times. The specific location of the misses that cause these stalls varies by application. However, each benchmark studied here is built around a core data structure, and we find that the bottleneck for each benchmark is due to specific operations over this data structure. We discuss the benchmark specifics below. Source code sizes vary from 280 lines of C 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.

[Figure 1 about here.]

R-Tree. R-Trees, originally proposed by Guttman ⁽¹⁵⁾, 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 ⁽¹⁶⁾. The R-Tree stores spatial information of objects such as airports and restaurants. Queries usually include spatial constraints such as “all printers in this building” and “landmarks within five miles of the airport”. To execute these operations efficiently, the R-Tree groups objects based on their spatial attributes. Figure 1 shows the structure of the DIS implementation of the R-Tree, a height-balanced tree with a branching factor usually between two and fifteen. Each subtree is associated with a bounding box in four dimensional space, or *hypercube key*, that encompasses the bounding boxes of all the nodes within the subtree. Each internal node contains a set of entries implemented as a linked list, where each list member is a (*child, hypercube key*) pair corresponding to one child subtree of the node. Entries at leaf nodes point to the data objects. Simple tree balancing mechanisms are invoked during node merging and splitting.

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. Query and delete tree operations require searching, whereas insert operations access nodes along a single path from root to leaf. 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 performing 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. R-Tree optimizations and performance are discussed in greater detail elsewhere ⁽¹⁾.

[Figure 2 about here.]

IRREG. The IRREG kernel is an irregular-mesh PDE solver taken from computational fluid dynamics (CFD) codes ⁽¹⁷⁾. 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. When the loop in Figure 2 is run on it for 40 iterations, the L1D and L2 hit ratios are 81.81% and 53.86%. As in EM3D, described below, indirect accesses to remote nodes (array x) are expensive. For large data arrays with adequately random index arrays, execution time scales with the number of edges and iterations.

Ray Trace. In Ray Trace, part of the Data Intensive Systems (DIS) benchmark suite ⁽¹⁶⁾, rays are emitted from a viewing window into a 3D scene, and each ray is traced through its reflections. A ray is defined by an origin (position) in the viewing window and a direction. The algorithm checks each ray for intersection with all scene surfaces and determine 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 is the only DIS-specified input that has high miss ratios. It has 9,828 surfaces and 29,484 vertices, and has a memory footprint of about 6.5MBytes, or roughly three times cache size. The L1D and L2 cache hit ratios for a window size of 1024×1024 are 97.6% and 70.15%. Almost the entire object data structure is accessed in processing each ray, and strided accesses and pointer chasing are the dominant access patterns. Execution time scales almost linearly with the viewing window (*i.e.*, the number of emitted rays) and input scene size.

FFTW. The highly tuned DIS FFTW benchmark ⁽¹⁶⁾ outperforms most other known FFT implementations ⁽¹⁸⁾. 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 suffer very high cache and TLB miss ratios. For an input array of size $10240 \times 32 \times 32$, with a memory footprint of 320MBytes, 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 ⁽¹⁹⁾ 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 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 ⁽²⁰⁾. 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, 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 (arbitrar-

ily) at 10. For an input size of 128K nodes and a degree of 10, the total memory footprint is about 30MBytes. The corresponding L1D and L2 hit ratios are 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 half) ⁽²⁰⁾. Execution time scales linearly with the number of nodes, and scales sub-linearly with increasing degree.

Health. From the Olden benchmark suite ⁽¹⁴⁾, Health simulates the Columbian health care system. The core data structure is a quad-tree with nodes that represent 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, and waiting time depends on the numbers 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 the patient lists. For a sample input of depth six, after 500 simulation cycles 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 Logical Operations

Computation Regrouping is targeted at applications whose program structure is expressible at the granularity of *logical operations*, which we define to be short streams of nearly-independent computations. Table II presents the list of logical operations in our benchmarks. Applications typically contain many instances of these logical operations, but are not intentionally programmed that way. Many data objects are accessed within each logical operation, but with little or no reuse. In contrast, there is significant reuse across logical operations. The key insight behind Computation Regrouping is

that application structuring in terms logical operations results in poor memory performance due to the large temporal separation between computations accessing the same data: bringing these computations temporally closer improves memory performance. Computation Regrouping identifies a set of data objects that fit into L2 cache, and then reorganizes the computations from multiple logical operations to maximize utilization of these cached objects. In this context, we define the *critical working set* to be the average size of the data objects accessed by a single logical operation. Table II’s *Estimated Threshold* column gives estimated minimum parameter values for the critical working set to exceed L2 cache size (given a 2MByte cache and a 128Byte line size), the point at which application performance begins to degrade significantly.

[Table 2 about here.]

[Figure 3 about here.]

Consider an R-Tree on which we perform insert, delete and query operations requiring tree traversals whose accesses are data-dependent and unpredictable at compile time. Each traversal potentially contains many hypercube-key comparisons. In particular, query and delete operations traverse multiple tree paths and have large working sets. The average memory cost of the comparisons dominates the processor cycle cost. To better visualize an R-Tree’s memory behavior, consider the graphs in Figure 3. Figure 3(a) shows a small window of queries versus node identifiers in our R-Tree baseline on a test input. Each dot represents an access made to the node identified by the x -coordinate by the query identified by the y -coordinate. Each row of dots 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 overlaps — there are significant gaps along the vertical axis. In general, queries for which there is significant overlap in accessed data are widely separated in time. Second, the set of unique nodes

accessed between successive visits to a given node is the union of the rows of dots whose row (query) number lies between those of the two dots, and the number of unique other nodes accessed is the size of this union. Figure 3(b) shows the distribution of this union’s size with query number. In our observations, the size of this union is typically large — sometimes including half the total tree nodes. With large trees compared to cache sizes, and with the high number of nodes accessed (with repeat accesses to a node spaced widely in time), nodes are evicted from cache before they can be reused.

R-Tree’s access behavior is representative of the applications that we study, which exhibit several conditions that lead to poor temporal locality. These applications consist of many logical operations, where each operation potentially accesses many data objects, but where there are few accesses to any given object and little computation is performed per object. The logical operations execute in a input-dependent order, and each runs to completion before another is started. Finally, the critical working set sizes exceed the cache size. These conditions are strikingly visible in the FFTW benchmark, which consists of a series of column walks that have 100% overlap between cache lines touched, but for which the reuse cannot be exploited. Our other benchmarks behave similarly.

Computing the cost of the memory accesses in any computation requires determining the number of computations within a logical operation, the relationships among computations in different logical operations, and the cache sizes. This information is usually unavailable to the compiler, and thus it is difficult to design generally effective static techniques to take advantage of the data reuse we target here. The approaches in the next section represent preliminary steps towards developing widely applicable techniques, but much work remains in terms of producing tools to automate or semi-automate their application.

4 Computation Regrouping

This section presents four simple optimization techniques that realize Computation Regrouping, summarizes their properties, and discusses the trade-offs involved.

[Table 3 about here.]

4.1 Transformations

We present four related implementation techniques: *early execution*, *deferred execution*, *computation merging*, and *filtered execution*. The ideas behind each transformation are similar, but the implementations, tradeoffs, and effects on application source are sufficiently different that we treat each separately. Applying the techniques requires identifying computations 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 costs and applicability of each technique.

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 column walks access 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 column 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 ⁽²¹⁾, 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 that will 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 dependency 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 Merging. Computation merging 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 that achieves the end 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 merging 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 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 show this for a simple, indirect-access kernel. In each iteration, accesses to locations outside the window are deferred. Like computation merging, filtered execution is a special case of deferred execution in which the deferral is achieved implicitly by the modified loop control structure. The window improves temporal locality in each iteration, but at the cost of added computational and source-code complexity. Figure 6 shows the transformation 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.]

4.2 Tradeoffs

Regrouping can increase complexity. First, control and data structure extensions make it more difficult to code and debug the application. The extent of the modifications depends on the application being optimized. Second,

reordering computations can reorder the output. If the correctness of the application is defined in terms of the program output, then either the definition of the correctness must be modified, or some data structure extensions must be incorporated to ensure correct output order. Third, regrouping is a general approach, and different applications require different implementation strategies. Choosing appropriate values for optimization parameters requires experimentation (*e.g.*, choosing queue length in R-Trees, or filter window sizes).

The increased computational overhead from regrouping is justified in many cases. Improvements in overall execution time can be significant, and most of these come from reducing memory stall times. Few generic techniques exist to optimize for memory performance in complex applications. In our experience, the control and data structure changes required are usually small compared to overall code sizes. Optimizations are architecture-independent, modulo the cache model, and we expect improvements to hold across generations of processor technology. Finally, most existing compiler-based memory optimizations cannot be directly applied to complex data structures, but regrouping achieves some of the desired effects. For example, early execution for FFTW simulates the effect of tiling. Overall, Computation Regrouping often represents a reasonable tradeoff between added overhead and improved memory performance.

5 Results

[Table 4 about here.]

We study the impact of Computation Regrouping on the benchmarks discussed in Section 2. We incorporate the appropriate regrouping optimizations into the reference implementations, and execute them all on the same sample inputs. We perform all experiments on a 14-processor SGI Power Onyx with 194MHz MIPS R10000 processors and 4GBytes of main memory. Each processor has 32KByte instruction and data L1 caches, and a 2MByte

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 caches. We use the SpeedShop performance tool Suite ⁽²²⁾ and the perfex ⁽²³⁾ command line interface to processor performance counters. We compile with the MIPSpro C compiler at the -O3 level of optimization. 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. Data presented represent the mean values obtained for these ten experiments.

Table IV presents a summary of the specific optimization techniques used and their high-level impact on each application. Complex applications such as CUDD can benefit from multiple regrouping techniques. The table shows the extent of modifications performed to the application. 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, the number of instructions issued and graduated is higher in the restructured application, which demonstrates the tradeoff identified in Section 4.2: increased computation can result in improved performance.

In section 5.1 we discuss overall performance results, and in section 5.2 we discuss the regrouping optimization for R-Tree in more detail.

5.1 Performance Summary

Table V summarizes the effectiveness of Computation Regrouping for improving the memory performance of our benchmark applications. The mean speedup is 1.97, although individual speedups vary significantly. Improvements are input-dependent, and the speedups are occasionally negative. Nonetheless, our experiences have been positive in terms of the applicability and effec-

tiveness of Computation Regrouping. We experiment with a range of inputs for each benchmark and observe positive speedups for most cases. Reducing memory stall time yields almost all the measured performance improvements. In some cases computation costs are reduced, too. In cases where there is a *decrease* in the computation costs, especially in applications optimized using computation merging, we could not 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.]

We incorporate the appropriate regrouping optimizations into each reference implementation, and execute all versions on the same sample inputs on a 14-processor SGI Power Onyx with 194MHz MIPS R10000 processors and 4GBytes of main memory. Each processor has 32KByte L1 instruction and data caches, and a 2MByte 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 caches. We use the SpeedShop performance tool suite ⁽²²⁾ and the perfex ⁽²³⁾ command line interface to processor performance counters. We compile with the MIPSpro C compiler at the -O3 level of optimization. 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. Data presented represent the mean values obtained for these ten experiments.

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 incurred to achieve this reduction is 0.3-10.9% of the original stall time.

R-Tree. Section 4 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 16MBytes. 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 by using 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. 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. Cache hit ratios for the L1D and L2 caches are 84.5% and 54.5%, respectively, for the baseline application, and 84.9% and 84.3%, respectively, for the optimized version. The performance improvement comes primarily from the increase in the L2 hit ratio. Table V shows that regrouping eliminates 56% of the memory stall time, and that the corresponding overhead is about 11%. 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, randomness in the input, and small amount of computation in the innermost loop are important to obtaining good performance improvement. 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 thus 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 4MBytes, and a filter window size of approximately half that allows most data to reside in L2 cache in the optimized version. The overheads incurred are due to the filtering control structure and the array that temporarily stores reflected rays. We defer processing reflected rays until all other rays have been processed. Our speedup is 1.98 for an input viewing window of 256×256 (64K rays). The L1D and L2 cache ratios improve from 95.8% and 70.1% to 96.1% and 99.6%, respectively. The optimized application effectively becomes 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 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 column walks on 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. The early execution optimization for FFT requires that when a column walk is executed, future column walks accessing the same data be executed at the same time. In this way, we simulate the effect of a short, synchronized multicolumn walk. The first iteration loads a fixed number of

cache that are reused by the remaining column walks.¹ This 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* multicolumn walk, but there is also significant gain from the *copy* multicolumn walk. The regrouping overhead arises from the necessary control extensions (additional function calls and loops). There is no memory overhead. Table V shows that the application speedup is 1.55. Although the L1D hit ratio decreases slightly, the nearly 50% increase in the L2 hit ratio offsets the impact. Memory stall time decreases from 73.8% of the baseline application execution time to about 45% of execution time after optimization. The same optimization can be applied to the *y*-dimension to further improve performance.

CUDD. We experiment with the *nanotrav* tool from the CUDD package⁽¹⁹⁾. Nanotrav stores input circuits as binary decision diagrams (BDDs) and executes a series of operations to reduce the BDD sizes. The core 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 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 the execution time, and could be further reduced. Our experimentation suggests an alternative implemen-

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

tation 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 merging 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 that 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 merging 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 application experiences a slowdown at the beginning: the waiting lists are short and require frequent modification, which induces high deferral overhead. This is offset by the reduced number of accesses during later execution, when there are long waiting lists and infrequent modifications. The simple structure of Health enables an efficient implementation of the regrouping optimization. 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 the optimization, most of the execution time is spent servicing cache misses. To improve memory performance further, we can implement a more aggressive variation of the

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 merging dramatically reduces the amount of computation. This change in total computation made it impossible to accurately calculate regrouping overhead.

5.2 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, which is 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, and incur 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.

Our results for the synthetic input indicate that regrouping can potentially eliminate most of memory stall time at modest computational cost, even for complex data structures like R-Trees. The memory-stall contribution to execution time decreases from almost 95% to less than 27% in the optimized version, whose total execution time is about 25% of the baseline 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,

respectively. We expect regrouping for temporal data locality to be similarly complementary to other spatial-locality optimization techniques.

Clustering is a layout optimization that attempts to group data structure elements 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 closely together. We call this layout modification *inter-node clustering*. In our implementation, inter-node clustering assumes intra-node clustering.

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 necessary for the general case where the insert, delete and query operations are interleaved, but not needed for our synthetic input case.

[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 op-

erations, and by making sure that deferred operations do not consider nodes of a newer generation, the consistency of the tree structure is maintained, 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 MBytes. 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 beyond 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.

We find that the exact placement of the queues is less important factor compared to 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.

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 ^(24, 25, 26, 27, 28) have been developed for specific applications, such as sorting ⁽²⁷⁾, query processing ⁽²⁸⁾, and matrix multiplication ⁽²⁵⁾. These 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. Such cache-conscious algorithms significantly outperform conventional algorithms, but 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 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, 29, 30, 10, 31). 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 ⁽³²⁾. 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 applica-

tions and data structures, but making it automatic instead of ad hoc is part of future work.

Complex structures or access patterns are not usually considered for compiler-based restructuring because of the difficulty in deriving accurate analytic models. In addition, 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 similarities. 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 pos-

sible, 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, 34), 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: 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 techniques, such as clustering and coloring ⁽¹¹⁾, take this approach and consider more complex linked data structures like trees and graphs for memory optimizations ^(11, 12, 13). Spatial blocking is effective for B-Trees and their 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 ⁽³⁵⁾. 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 how the approach will scale to handle more complex data structures. We extend previous work by considering applications with complex data structures and access patterns. Rinard and Diniz ⁽³⁶⁾ show how pointer-based data structures can be analysed for commutative property, and how commutative operations can guide parallelization of code

in multiprocessor environments. Computation regrouping is very similar to commutativity analysis. We extend their work by showing existence of opportunities for commuting computations across logical operations, and also that such an approach can yield performance improvements on a uniprocessor under certain conditions.

A queue-enhanced R-Tree called a *Buffer Tree* was previously proposed in the context of external memory algorithms^(37, 38). 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 Another View

Computation regrouping may be viewed as simulating a combination of distributed shared machine (DSM) machine and a compiler optimization used in such machines. We discuss the characteristics of this simulation here.

As the speed gap between the memory and the processor increases, the cost structure for on-chip and off-chip accesses on uniprocessors resembles that for local and remote accesses in traditional multiprocessors. We may therefore view main memory as a separate processor, albeit without computing capability. Applications and compilers targeting DSM machines traditionally try to eliminate remote accesses through appropriate computation and data distribution techniques. By simulating—on a uniprocessor—an environment that these DSM techniques target, we can exploit this rich source of existing techniques for optimizing memory-intensive applications. The resemblance of the cost structure in today’s uniprocessors and DSM machines makes this simulation viable. Intuitively, regrouping optimizations perform this simulation. We discuss an example showing how regrouping parameters can be interpreted in terms of the simulation parameters.

We can simulate a virtual two-processor machine, consisting of the CPU and the main memory, on the uniprocessor machine by interleaving their

activities in time, similar to the execution of multiple processes in a standard operating system. The main memory holds the state of the processor that is inactive at that point in time. This state is “switched-in” when the processor corresponding to the state becomes active, and the active processor’s state is “switched-out” into the main memory. We can simulate an arbitrary number of processors, if it suits the application and if there is sufficient main memory. The simulation is performed by embedding computation regrouping code within the application.

Consider the IRREG example discussed in Section 2. We can execute the IRREG code on a arbitrary DSM machine by partitioning the data array and distributing the partitions among processors. We then replicate the code segment in all processors, to reduce instruction cache misses. The size of individual partitions, as well their distribution, is dependent on the specific DSM optimization technique used. Consider the regrouping optimization performed on IRREG. We partition data in a similar way, but only between the CPU and the main memory. Each partition’s size is equal to the sliding window size. The number of windows, or equivalently partitions, is the number of passes that the optimized code makes over the data array. Since we assume that one partition is assigned to each processor, the total number of processors simulated is equal to the number of passes. The simulation of the processors themselves is straightforward: it merely requires moving the window as explained in Section 5. This is effectively a simple round-robin algorithm for temporal interleaving of execution of the simulated processors. A more complex data structure like R-Tree simulates a dynamically varying number of processors with a more complex interleaving algorithm based on the queue occupancy.

Temporal simulation of processors is expensive due to the high costs of switching between those processors. Every simulated processor sees a large number of cold misses each time it is switched in. However, once a processor is switched in, the data and computation distribution techniques derived from the DSM domain help reduce remote accesses in this virtual multipro-

cessor. Remote accesses in the virtual processors correspond to main memory accesses in the real system. This tradeoff is appropriate to only those applications that have large working sets, i.e., they incur many main memory accesses, and have a structure that allows clean distribution of the computation. Our experience with the benchmark applications shows most of them to have both these characteristics.

This virtual multiprocessor differs from the existing multiprocessor targets in several ways. First, true parallelism between the processors does not exist. This eliminates the need for data access synchronization between processors, which reduces the computational overhead. Second, the target itself is flexible. The number of processors, the memory distribution across processors, and the scheduling algorithms can be tuned to the specific application under consideration even at a fine granularity of timeslices. Third, the load distribution among these processors is unimportant, unlike in the DSM domain, where many techniques try to achieve uniform load to exploit parallelism to the maximum extent. This simplifies the DSM algorithms, and allows for exploration of a larger set of potential configurations.

In retrospect, we find that for this problem domain, DSM simulation is a more convenient mental model than logical operations. Further refinement and experimentation is needed to identify the usefulness of the approach beyond the applications considered here. This approach has the advantage that it is likely to work well for complex, pointer-based data structures. However, the approach remains hard to automate and requires substantial understanding of the application data structure. Finally, the coarse-grained simulation has high overhead, and therefore is not useful for many applications (e.g., those without large working sets and lacking an organization that supports a clean distribution of the computation).

8 Conclusions

Application performance is increasingly limited by memory performance. The growing CPU-memory speed gap only exacerbates the problem. We present a software approach to attack the problem, trading extra computation for reduced memory costs. *Computation Regrouping* executes computations accessing the same data closer together in time, which significantly improves temporal locality (and thus performance) for applications with poor locality. The programmer overhead for identifying appropriate logical operations and the execution overhead of the modified application are the costs we trade for increased temporal access locality. We present a few implementation techniques to realize the regrouping approach, and demonstrate that Computation Regrouping successfully eliminates 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 ⁽³⁹⁾ 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 interact in meaningful ways with the cache-based memory model of modern processors and impact application performance. Understanding the nature of these interactions can help us to design techniques to 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 for his support of 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] 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.
- [15] A. Guttman, "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.
- [16] J. W. Manke and J. Wu, *Data-Intensive System Benchmark Suite Analysis and Specification*. Atlantic Aerospace Electronics Corp., June 1999.
- [17] 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.
- [18] 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.
- [19] F. Somenzi, "CUDD: CU Decision Diagram Package Release 2.3.1," 2001.

- [20] 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.
- [21] 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.
- [22] Silicon Graphics Inc., *SpeedShop User's Guide*. 1996.
- [23] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, “Performance Analysis Using the MIPS R10000 Performance Counters,” in *Proceedings of Supercomputing '96*, November 1996.
- [24] 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.
- [25] 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.
- [26] 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.
- [27] A. G. LaMarca, *Caches and Algorithms*. PhD thesis, University of Washington, 1996.
- [28] 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.
- [29] I. Kodukula, N. Ahmed, and K. Pingali, “Data-Centric Multi-level Blocking,” in *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pp. 346–357, June 1997.
- [30] 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.

- [31] 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.
- [32] 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.
- [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] 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.
- [35] 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.
- [36] M. C. Rinard and P. C. Diniz, “Commutativity analysis: A new analysis framework for parallelizing compilers,” in *Proceedings of the SIGPLAN ’96 Conference on Programming Language Design and Implementation*, pp. 54–67, May 1996.
- [37] 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.
- [38] 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.
- [39] E. S. Roberts and M. T. Vandevoorde, “WorkCrews: An Abstraction for Controlling Parallelism,” Tech. Rep. SRC-042, Digital Systems Research Center, April 1989.

List of Tables

I	Benchmark characteristics	37
II	Logical operations for benchmarks	38
III	Characteristics of regrouping techniques	39
IV	Regrouping results	40
V	Perfex-estimated performance statistics	41
VI	R-Tree performance for combinations of Computation Regrouping and clustering	42

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 × 32 × 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 Merging	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 Merging Filtered Execution	700	200	646M	617M	428M	542M
Health	Pointer Chasing	Computation Merging	600	30	1.6B	0.43B	0.83B	0.39B

Table IV: Regrouping results

Benchmark	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 (time)	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

List of Figures

1	R-Tree structure	44
2	IRREG main loop	45
3	R-Tree performance with about 70K nodes	46
4	Original code for indirect access	47
5	Filtered execution code for indirect access	48
6	Filtered execution code for IRREG	49
7	Speedup vs. window size for IRREG	50
8	Modified structures to support regrouping via queuing	51

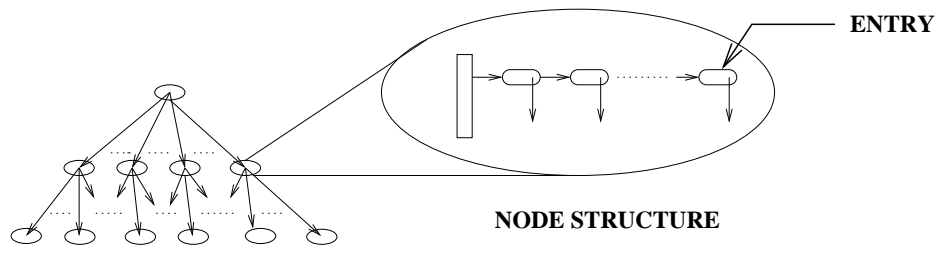
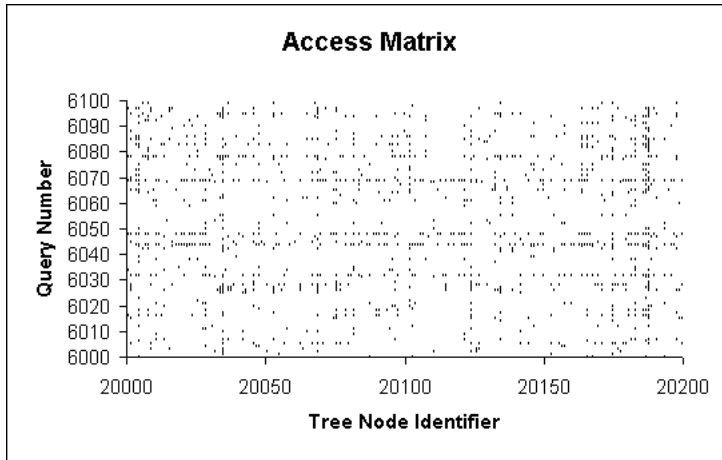


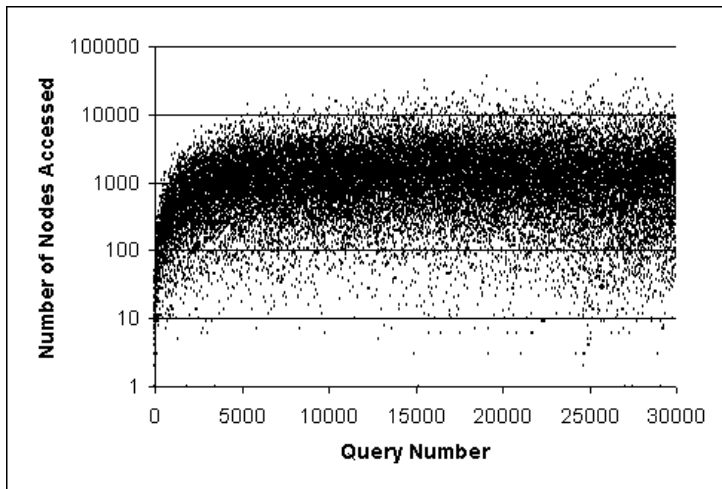
Figure 1: R-Tree structure

```
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

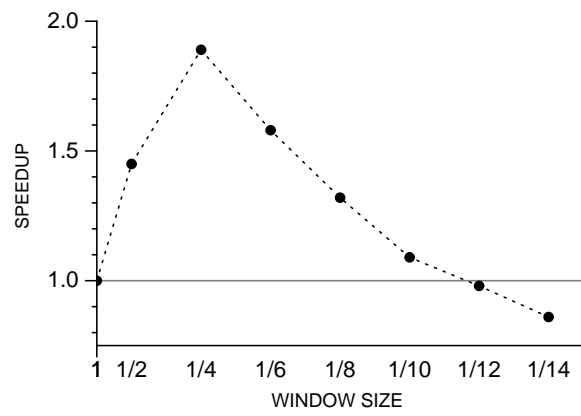


Figure 7: Speedup vs. window size for IRREG

```

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