The Potential of Computation Regrouping for Improving Locality

Chen Ding and Maksim Orlovich

Computer Science Department
University of Rochester
{cding,orlovich}@cs.rochester.edu

Abstract

Improving program locality has become increasingly important on modern computer systems. An effective strategy is to group computations on the same data so that once the data are loaded into cache, the program performs all their operations before the data are evicted. However, computation regrouping is difficult to automate for programs with complex data and control structures.

This paper studies the potential of locality improvement through trace-driven computation regrouping. First, it shows that maximizing the locality is different from maximizing the parallelism or maximizing the cache utilization. The problem is NP-hard even without considering data dependences and cache organization. Then the paper describes a tool that performs constrained computation regrouping on program traces. The new tool is unique because it measures the exact control dependences and applies complete memory renaming and re-allocation. Using the tool, the paper measures the potential locality improvement in a set of commonly used benchmark programs written in C.

1 Introduction

The memory performance of a program is largely determined by its temporal data locality, that is, how close the same data are reused during an execution. The primary method for improving the temporal locality is regrouping those computations that use the same data, hence converting long-distance data reuses into short-distance cache reuses. Well-known techniques such as loop interchange, blocking (tiling) and fusion regroup computations in regular loop nests. Recent studies show that more general forms of computation regrouping have significant benefits in other complex programs such as information retrieval, machine verification, and physical, mechanical, and biological simulations [11, 32, 36]. Computation regrouping is the only form of computation reordering we study in this paper, so we use the terms *computation regrouping* and *computation reordering* interchangeably in this paper.

The goal of computation regrouping is to maximize program locality. Program locality is difficult to define, so is the optimality of computation regrouping. We use a precise measure of locality in this paper, which is the capacity cache miss rate of an execution, that is, the miss rate of the fully-associative LRU cache save the first access to each element [20]. We consider an abstract form of cache that consists of data elements rather than cache blocks, so we separate the effect of the spatial locality. Spatial locality has a straightforward upper-bound, where cache blocks and the cache are fully utilized. The lowest cache miss rate is the lowest capacity miss rate divided by the size of the cache block.

Computation regrouping is difficult to apply to large programs because related computations often spread far apart in the program code. Traditional dependence analysis is often not effective because of the complex control flow and indirect data access especially recursive functions and recursive data structures. Until now, the automatic methods are limited to programs written in loop nests, and manual transformation is limited to small fragments of a limited number of applications.

We present a study on the potential of computation regrouping. First, we show that maximizing the locality is different from maximizing the parallelism or maximizing the cache utilization. We show that problem of computation regrouping is NP hard even when not considering program dependences and cache organization. We then present a simulation tool that measures the potential of computation regrouping. The tool allows for aggressive yet correct reordering by respecting the exact data and control dependences and applying complete data renaming and memory re-allocation. It uses a constrained regrouping heuristic to improve the locality for a given cache size. Finally, we evaluate the new tool on a set

0-7695-2153-3/04 $20.00 (c)2004 IEEE

of numerical and integer programs. The tool may have a significant value in high-performance computing, as discussed in Section 5.

Many studies have measured the potential of the parallelism at the instruction, loop, and procedure level (for examples [25, 31, 38]). Most of them ignored the locality effect, assuming a single-cycle memory latency. However, on modern machines, the difference in the memory access time is reaching two orders of magnitude depending on the data locality. On SGI Origin 2000, the matrix multiply executes the same number of instructions 100 times faster after the locality optimization. Hence, studying the potential of locality is important because it may lead to significant performance improvement. For general-purpose programs, where the average number of parallel instructions is mostly below 10, the performance potential from the locality may rival or exceed that from the parallelism.

The simulation tool is trace-based and has several limitations. A trace may not represent the program behavior on other inputs, and a trace may be too large to be analyzed. For many programs, earlier work has shown that the temporal locality follows a predictable pattern and the (cache miss) behavior of all program inputs can be predicted by examining medium-size training runs [14, 15, 26, 34, 44]. In this paper, we use a medium-size input for each program. Determining the improvement across all program inputs is a subject of future study. Furthermore, the simulation uses heuristics and does not find the optimal locality. We use the simulation tool to measure the lower bound of locality improvement in complex programs by trace-level computation reordering.

2 The Optimal Temporal Locality

Given a sequence of memory accesses and the cache of a given size, we want to minimize the number of cache misses. The past work in memory and cache management shows that given a fixed access sequence, the cache has the optimal performance if it keeps in cache the data with the closest reuse, i.e. the *OPT* strategy [3, 4]. However, the optimality no longer holds when we reorder the sequence, that is, when we change the order of data reuses.

To find the best access order, we need to first determine whether and when it is legal to reorder by observing the dependences among program operations and then finding the independent and parallel operations. Many past studies examined the parallelism at the instruction, loop, and program level. The available parallelism does not directly measure the available locality. Even assuming the maximal parallelism, where program operations can be freely reordered, finding the optimal locality is not trivial, as shown next by an example.

Inst. list	data access	OPT caching
T:	**(a h)**	**a h**
U:	**(b c)**	**a b c h**
V:	**(d e f)**	**a b c d e f h**
W:	**(b g h)**	**a c d e f g**
X:	**(c f k)**	**a d e g k**
Y:	**(a d g)**	**e k**
Z:	**(e k)**	

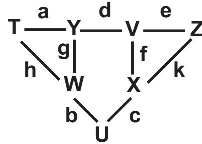
Figure 1: An example sequence of seven instructions, each uses two to three data elements. It needs minimally seven-element cache to avoid capacity misses. **Before preceding further, the reader is urged to reorder the sequence to reduce its cache demand, assuming the sequence is fully parallel and can be freely reordered.*

Figure 1 has three columns: the first shows a sequence of operations, the second shows the set of data elements accessed by each operation, and the third shows the cache content under the OPT strategy. All data reuses are cached if the cache size is at least seven elements. It is possible to reduce the cache demand to fewer than seven elements, if one reorders the sequence. We urge the reader to try the reordering problem, assuming the operations are fully parallel and can be arbitrarily reordered.

The solution to the example problem is given in Figure 1. Part (a) uses a graph to model the data sharing between operations. Part (b) gives a sequence where each pair of neighboring nodes are separated by at most one other node. This property is called *bandwidth-1*. Each pair of data-sharing operations are closest in this sequence. Indeed, Part (c) shows that the reordered sequence needs only a three-element cache and therefore has much better locality than the original sequence.

As shown by the example, the goal of the locality optimization is to reduce the distance between operations that reuse the same data. The definition of the distance depends on the cache management scheme especially its replacement policy. Instead of modeling the exact cache management algorithms, we model a sufficient but not necessary condition. To ensure a cache hit, we require that the *time distance*, which is the number of instructions between the two accesses, is bounded by a constant. The time distance between two instructions is the analogous of the bandwidth in the bandwidth-minimization problem. We now define the time-distance problem and show it is NP-hard.

Definition The time-distance problem: A program execution is a sequence of instructions. Some instructions may access three or more data elements. Given a program and a cache size k, the problem is to reorder the instruction sequence so that for all pairs of instructions that access the same data, the number of the pairs whose time distance is bounded by k is maximized.



(a) The data-sharing relation as a graph. Each edge connects two instructions that use the same data.

T W Y U V X Z

(b) A bandwidth-1 sequence, in which the endpoints of any edge in (a) are separated by at most one other node. The graph has only two such sequences: this one and its reverse.

Inst. list	data access	OPT cache	4-elem. LRU
T:	**(a h)**	**a h**	**a h**
W:	**(b g h)**	**a b g**	**a h b g**
Y:	**(a d g)**	**b d**	**b a d g**
U:	**(b c)**	**c d**	**d g b c**
V:	**(d e f)**	**c e f**	**c d e f**
X:	**(c f k)**	**e k**	**d e c f**
Z:	**(e k)**		**c f e k**

(c) The recordered sequence according to (b). All reuses are fully buffered by 3-element OPT cache, 4-element LRU cache, or 3 registers.

Figure 2: The example problem of optimal locality, solved as an instance of the bandwidth problem

Theorem 2.1 *The time-distance problem is NP-hard.*

Proof We reduce the tree bandwidth-k problem [18] to the time-distance problem. Let T be a free tree with all vertices of degree ≤ 3. The bandwidth-k problem is to find a linear sequence of all tree nodes so that any two connected tree nodes are separated by no more than k nodes in the sequence. Given a tree in the bandwidth-k problem, we construct an execution trace as follows. Let each tree edge represent a unique data element. Put the tree nodes in a sequence. Convert each node to an instruction that accesses the data elements represented by its edges. Since the node degree is no more than three, an instruction accesses at most three data elements. The conversion takes the time linear to the graph size. Assume no data or control dependence in the converted sequence. A bandwidth-k sequence exists if and only if the optimal time-distance solution exists, that is, one can reorder the instruction trace so that no two data-sharing instructions are separated by more than k instructions. Since bandwidth-k is NP-hard for trees whose nodes have up to three edges, the time-distance problem is also NP-hard. ∎

The theorem shows that even without considering the complication from data dependences and cache organization, the locality problem is NP-hard. In addition, maximal parallelism and optimal cache management do not imply optimal locality. Next we study the potential of locality improvement through trace-driven computation regrouping.

3 Trace-Driven Computation Regrouping

The section presents the five steps of trace-driven computation regrouping. The first step constructs instruction traces by instrumenting programs to record relevant information. Of course, not all ordering is permissible. The next two steps identify the exact control and data dependences in the trace. The control dependence analysis uses a novel algorithm to handle recursive programs. The data dependence analysis uses complete renaming to avoid false dependences. The fourth step applies constrained computation regrouping. Finally, the last step reallocates data variables to reduce the memory usage after reordering. We now describe these steps in the following five sub-sections.

3.1 Trace Construction

We construct instruction traces through source instrumentation of original programs. We first transform program statements (except for call statements) to make the assignment and control flow explicit, using the C and Fortran compilers we developed for earlier studies [13, 43]. The

following example shows the source-to-source transformation of a compound statement in C.

```
a[0][i==0?i++:i].a[i] = 0;
```

becomes

```
  RecInst(1, 1, &i);
  if ((i != 0)) goto L4;
  RecInst(2, 2, &tmp2, &i);
  tmp2 = (i + 1);
  RecInst(2, 2, &i, &tmp2);
  i = tmp2;
  RecInst(2, 2, &tmp1, &tmp2);
  tmp1 = tmp2;
  goto L5;
L4:;
  RecInst(2, 2, &tmp1, &i);
  tmp1 = i;
L5:;
  RecInst(1, 1, &(a[0][tmp1].a[i]));
  a[0][tmp1].a[i] = 0;
```

After the transformation, we can simply prefix each source line with a call to a special RecInst function. Its parameters give the number of locations being accessed, the number of reads, and the list of locations where a write, if any, is followed by reads. The locations are given by the virtual memory address. For example, the three parameters of the first RecInst call show that the next instruction has one memory access, it is a read, and the read location is the address of variable i. This of course provides exact information on all scalar dependencies, as to see what a read is dependent on one merely needs to look up the last write to the location. It includes false dependences as local variables reuse the stack space. These and other false dependences will be systematically removed in a later step through variable renaming.

This is not, however, sufficient for per-structure and other non-scalar operations. To address this, we split all per-structure operations into per-field operations, making all memory accesses explicit. Another issue is the accesses to portions of scalars using either pointers of different types of unions. We did not address this issue in our implementation, choosing to assume the program is wholly type-safe. Run-time checks can be applied to this case to map byte or similar accesses to the original scalar types, thus providing at least a conservative estimate of the dependencies.

Another complication is in parameter passing, as there needs to be a dependence from the calculation of parameters to the use. We resolve this by introducing special variables, to which we simulate writes of the passed-in values using synthesized RecInst calls, and from which we simulate copies to the actual parameters. Return values are handled similarly. This, of course, also fails to work if a structure is passed or returned by value. We address that by passing a copy by reference instead, and adjusting the code accordingly.

3.2 Control Dependence

Given a program, a statement x is control dependent on a branch y if the following two conditions hold: in one direction of y, the control flow must reach x; in the other direction, the control flow may not reach x [2]. Like the static data dependence, the static control dependence is a conservative estimate. It is not exact. The statement x and the branch y may have many instances in an execution, but not all instances of x are control dependent on all instances of y. Next we discuss the the exact control dependence through an example and then present an analysis algorithm.

3.2.1 Exact Control Dependence

The program in Listing 1 uses a recursive subroutine to traverse a linked list twice. It increments a counter after visiting a node. At the program level, the cost increment is control dependent on the empty test in the main function (whether to enter a list traversal). Via way of the recursive call, it is transitively dependent on the test in the recursive function (whether to continue the traversal). Ignoring these constraints may lead to fictitious executions where the counting is finished before knowing what to count. However, the static control dependence is not exact. During an execution, each cost increment is directly control dependent on only one of the two tests. The first increment depends on the test in the main function, but each of the later increments depends on the test that happened one invocation earlier in the recursive subroutine.

Listing 1: Examples of control dependence

```
main() {
  if (my_list != null) {
    count(my_list, condition_1);
    count(my_list, condition_2);
  }
}

count(list, condition) {
  if (list.next != null)
     count(list.next, condition);
  ...
  cost += 1;
}
```

Lam and Wilson observed the difference between the static and the run-time control dependence and gave an algorithm for tracking run-time control dependences during program simulation [25]. Presumably for efficiency reasons, they did not measure the exact control dependence

in the presence of recursion, for example the program in Listing 1. Other researchers such as Zhang and Gupta used exact control dependence but did not detail the algorithm [42]. We next present a method that finds exact control dependences by storing and searching relevant information in a stack.

Using the exact control dependence tracking, we find legal computation regrouping across complex control flows. For the example program, a legal regrouping merges the two list traversals and hence greatly improves the temporal locality in this recursive program. We note that current static analysis cannot recognize the two list traversals because of the information loss when analyzing the recursive call. By modeling recursive functions, we can study programs that use (mutual) recursive data structures such as lists, trees, and graphs as well as programs that use divide-and-conquer algorithms such as quick-sort and multi-grid simulation.

3.2.2 Exact Control Dependence Analysis

To make the discussion somewhat clear, we refer to statements in the source code as *source lines*, and the instances of source lines in the execution trace as *instructions*; we refer to an *underlying source line* of an instruction i to denote the source line i is an instance of, and abbreviate is as $\mathrm{usl}(i)$. We also limit the discussion to single procedures at first, and will generalize it to whole program shortly.

We define an instruction i to be run-time control-dependent on an instruction c when a conditional executed in c has chosen to follow a path in the control flow graph from $\mathrm{usl}(c)$ to a successor that is post-dominated by $\mathrm{usl}(i)$, while there exists a successor of $\mathrm{usl}(c)$ that does not have this property. Clearly, in this case $\mathrm{usl}(i)$ must be control dependent on $\mathrm{usl}(c)$. Further, we claim that c is the instance of a source line that $\mathrm{usl}(i)$ is control-dependent on that executed the closest before i. To show this, let i, c_1, c_2 be instructions, such that $\mathrm{usl}(s)$ is control dependent on $\mathrm{usl}(c_1)$ and $\mathrm{usl}(c_2)$, and c_1 occurs before c_2 in the execution trace. Notice that the underlying control-flow path for the portion of trace from c_1 to i could not have taken the path from $\mathrm{usl}(c_1)$ to $\mathrm{usl}(i)$ that has all intermediate notes that are post-dominated by $\mathrm{usl}(i)$, as $\mathrm{usl}(c_2)$ does not have that property, and it is on the path, so i is not run-time control-dependent on c_1, and hence if s is run-time control-dependent on c_j, no other candidate can intervene between the two.

Now, consider the case of multiple invocations of the same procedure. Clearly, we only want to consider statement instances in the current invocation of a procedure when testing for in-procedure control-dependence, since conditionals run in a previous copy are not directly relevant. Further, if an instruction is run-time control-dependent on an another instruction in the same subroutine instance, we clearly do not have to worry about any control dependencies within the same invocation. Since the partial order relationship induced by control dependence is transitive, and taking care of any control dependence of the controlling branch between invocations would ensure their execution before the controlled statement.

Thus, it remains to consider the case where an instruction does not have an intraprocedural control-dependence. But then, the run-time control dependence for the node is clearly the same as for the procedure call instruction used to invoke the current function.

To implement this run-time analysis, we first perform traditional static analysis, constructing the control flow graph, and then the control dependence graph [2]. The run-time state is maintained for each invocation of a function. The state includes the last access of the basic blocks of the function. The states are maintained as a stack just as the call stack. The compiler inserts code that pushes a new state at entry to a function and pops the top state at each exit of a function. The exit call is inserted before the actual return statements, but after the data dependence instrumentation produced for them. Further, each basic block is given a unique ID, and an array is constructed for it listing all of the basic blocks it is statically control-dependent on.

At the entrance to an execution b of basic block B, the analysis finds the source of the control dependence by calling $ControlDependence(b)$, given in Algorithm 1. It searches the state stack in the top down order for the last execution a of all basic block B' which B statically control dependent on. When closest a is the source of the control dependence. Once it is found in one state in the stack, no further search is needed for the rest of the stack.

Algorithm 1 Control Dependence Analysis
procedure *ControlDependence(b)*
  {*b is an execution of basic block B*}
  {*Find the source of control dependence, a*}
  Let a be null
  for each function f in the call stack in the top-down order **do**
    for each basic block B' that B statically control depends on **do**
      Let b' be the last instance of B' executed
      if distance(b', b)<distance(a, b) **then**
        {*b' is closer than a*}
        Let $a = b'$
      end if
    end for
    if a is not null **then**
      {*Stop the search*}
      break
    end if
  end for

return a as the source of the control dependence
end *ControlDependence*

3.3 Data Dependence

Given an instruction trace and the exact data access by each instruction, data dependence analysis is straightforward. In this study, we track only the *flow dependence*, caused by the flow of a value from its definition to one of its uses [2]. Two other types of data dependence are *anti*- and *output-dependences*, which are caused by the reuse of variables. They are removed by renaming, which creates a new name for each value in the same way as value numbering or static single assignment does in a single basic block.

The analysis traverses the instruction trace, records the last assignment of each program variable, defines a new name after each write, and records the current value (name) for each variable. For each variable read in the trace, the analysis converts it to a use of the current value and adds a flow dependence edge from the instruction of the last definition to the instruction of the current read. In the worst case, renaming adds a number of names proportional to the length of the trace. We will remap the names into variables in the last step of the analysis.

3.4 Constrained Computation Regrouping

Computation reordering clusters instructions that access the same data values. We call the data accessed by a group of instructions the *data footprint*. For a group of instructions to reuse data in cache, the size of their footprint should be no greater than the size of cache. Shown by Algorithm 2, constrained computation regrouping is a heuristic-based method for reordering instructions while limiting the size of their footprint.

The function *reorder* traverses the instruction trace. When an instruction is not yet executed, it calls *execute* to start the next reordering process from this instruction with an initial data budget ω_d, which is the size of the data cache. As the reordering picks an instruction that reuses a particular datum, it temporarily skips all other unexecuted instructions. It is a form of speculation. It may incur the execution of many other instructions and over spend the data budget, in which case we need to roll back and cancel the speculative reordering. Three other data structures are used to support the speculation. The instruction log L_i and the data log L_d, initially empty, keep the speculated but not yet committed reordering results. The instruction budget ω_i limits the depth of the speculation for efficiency reasons that we will discuss later.

Algorithm 2 Constrained Computation Regrouping

data structures
P is the instruction trace to be reordered
P' is the reordered trace
S_i is the set of executed instructions
S_d is the set of data current accessed

procedure *Reorder(*P, ω_d, ω_i*)*
$\{P$ *is the instruction trace;* ω_d *and* ω_i *are data and instruction budgets.*$\}$
Let L_d and L_i be the logs of the trial executions
for each instruction i in P in the execution order **do**
  if i is not executed **then**
    $\{$*Start a trial execution*$\}$
    call Execute(i, L_d, ω_d, L_i, ω_i)
  end if
end for
end *Reorder*

procedure *Execute(*i, L_d, ω_d, L_i, ω_i*)*
$\{i$ *is the current instruction;* ω_d *and* ω_i *are data and instruction budgets; and* L_d *and* L_i *be the logs of the trial execution.*$\}$
Store the current state $S =< L_d, L_i, P' >$
if not Check(i, L_d, ω_d, L_i, ω_i) **then**
  $\{$*The current trial failed*$\}$
  for each instruction i' since $S.L_i$ **do**
    Remove i' from $executed$
  end for
  for each data d since $S.L_d$ **do**
    Remove d from S_d
  end for
  Roll back to state S
else
  $\{$*The current trial succeeded*$\}$
  for each operand *op* in i **do**
    $\{$*Divide the budgets*$\}$
    Let n be the next (not yet executed) instruction that uses *op*
    call Execute(n, L_d, $L_d.size + \Delta\omega_d$, L_i, $L_i.size + \Delta\omega_i$)
  end for
end if

$\{$*Clean up*$\}$
if directly invoked by *Reorder* **then**
  for each data d in L_d **do**
    Remove d from S_d
  end for
  Clear L_d and L_i
end if
end *Execute*

procedure *Check(*i, L_d, ω_d, L_i, ω_i*)*
$\{$*Add the effect of* i *and check the budget*$\}$

```
    for each operand op in i do
        if op is not in S_d then
            Add op to S_d and L_d
            Add i to L_i and S_i
        end if
    end for
    if L_d.size > ω_d or L_i.size > ω_i then
        return false
    end if

    {Check all dependences}
    for each (not yet executed) instruction j that i depends
    do
        if not Check(j, L_d, ω_d, L_i, ω_i) then
            return false
        end if
    end for
    return true
end Check
```

The function *Execute* calls *Check* to inquire whether the execution of the current instruction would exceed the budget. Both functions may speculatively reorder instructions, so *Execute* first checkpoints the current state. If the current instruction cannot be executed within the budget, it will roll back to the beginning state and return. Otherwise, it continues the speculation on instructions that reuse the data of the current instruction. When more than one candidate is present, it divides the budget, currently, equally among the candidates. The budget division balances the reordering permitted for candidate instructions. The function *Check* traces back the dependences, executes (and checks) all predecessors, accounts their consumption of the budget, and checks for any over spending. If so, it returns false and causes the *Execute* to discard the trial. Otherwise, it completes the speculative execution and records the reordered execution.

The time cost of the algorithm is linear to the size of the trace, because it limits the size of each trial by the instruction budget. Since a trial takes a constant time, and every trial executes at least an instruction, the total time cost is linear. Without the instruction budget, the time complexity can be quadratic in extreme cases. The trials can be efficiently implemented by a combination of stack-like transaction logs and set data structures.

3.5 Memory Re-allocation

Memory re-allocation is the reverse of memory renaming. It assigns values with non-overlapping lifetimes into a single variable in order to minimize the storage requirement. It uses a single pass to calculate the live range of each value. It keeps free variables in a stack to minimize the distance of the data reuse. At the definition a value, it pops a vacant variable from the top of the stack (or create a new variable if the stack is empty). At the last use of a value, it frees its variable by pushing it into the stack. Note that this scheme does not necessarily minimize the number of needed variables. As shown in the evaluation, memory re-allocation after constrained computation reordering can eliminate the storage expansion caused by variable renaming. Therefore, removing all anit- and output-dependences does not cause an increase in the memory demand of a program.

4 Evaluation

In an earlier study, we used unconstrained computation regrouping on a set of Fortran 77 programs and observed significant locality improvements [12]. C programs are more difficult for computation regrouping because they often have complex data and control structures (including recursion). In this section, we use a set of C programs to evaluate the effect of the exact control dependence tracking, memory renaming and reallocation, and constrained computation regrouping. For each program, Table 1 shows the name, the source, a description, and the size of the input. The last two columns show that the programs have from under two thousand to about two hundred thousand data elements being accessed by a few hundred thousand to over six million memory times.

We first instrument a program, run it to get an execution trace, and measure the capacity miss rate on all cache sizes on the original trace [27]. Then, the trace is reordered, and the capacity miss rates are measured again. The temporal locality of the two versions are then compared by the cache capacity miss rates on all cache sizes. The regrouping does not change the number of memory accesses. However, memory renaming and reallocation changes the size of program data. The new sizes are shown inside parentheses in the last column of Table 1. The combined transformations reduces the size of program data in all cases. Therefore, a reduction in the miss rate means a higher reduction in the number of capacity misses.

We first test the efficacy of analysis in cases where there are well-known static improvement techniques. One test program consists of two simple loops which make two passes over an array. The two loops can be fused together by an aggressive optimizing compiler to reuse data immediately after it is first accessed. As shown by the left-hand graph in Figure 3, the trace-level computation regrouping eliminates capacity misses for a cache size greater than eight data elements, successfully reproducing the effect of loop fusion.

For the matrix multiply, shown by the right-hand graph in Figure 3, computation regrouping removes most of the capacity misses due to the longest data reuse. To be precise, it reduces the miss rate of the 4K cache from 7.6%

Table 1: Test programs

name	source	description	num. accesses (K)	num. data (K)
Fuse	local	two loops reading a single array	352	64.0 (0.0)
MMult	textbook	multiplication of two matrices	6,688	38.4 (25.8)
BSort	textbook	integer bubble sort, 768 elements	4,446	1.58 (1.53)
QSort	textbook	integer quick sort, 15K elements	3,543	30.6 (30.1)
Health	Olden [7]	linear searches of hospital patient records	3,916	59.8 (5.65)
MST	Olden	finding a minimal spanning tree	810	27.0 (3.25)
TSP	Olden	a traveling salesman problem solver	3,394	14.9 (7.03)
TreeAdd	Olden	recursive summing a balanced 15-level binary tree	2,589	196.9 (196.6)

* *The parenthesized numbers in the last column are the data size after memory renaming and reallocation.*

to 3.7%, a reduction of 58%. It demonstrates considerable potential for cache-performance improvement in this test, as is expected, but is not as effective as a hand-coded stripmine-and-interchange version, which resulted in a 87% improvement. Constrained reordering is a heuristic-based solution to an NP-hard problem. It loses to compiler blocking because it does not exploit the high-level structure of the matrix multiply program.

For further evaluation, we tested two common sorting algorithms — bubble sort and quick sort, and four programs from the Olden benchmark set: *Health*, *MST*, *TreeAdd*, and *TSP*. As illustrated in Figures 4, most of the tests showed major potential for locality improvement, while two tests — *QSort* and *TreeAdd* — showed no major improvement. The result for *TreeAdd* is consistent with expectations, since the benchmark performs a single walk over a binary tree and thus has very little reuse. On the other hand, *QSort* has excellent locality because it naturally blocks computation for power-of-two cache sizes. Computation regrouping finds little opportunity for improvement.

The exact miss-rate reduction is given in Table 2 for a cache size equal to $\frac{1}{32}$, $\frac{1}{16}$, $\frac{1}{8}$, $\frac{1}{4}$, and $\frac{1}{2}$ of the power-of-two size needed to have no capacity miss. The computation regrouping has little and no effect on *TreeAdd* and *QSort*. It reduces the average miss rate by 30% for *MMult* and 75% for *TSP*. For the other four programs, the reduction is at least 90% or a factor of 10, showing the great potential of computation regrouping in these programs.

The effect of exact control dependence tracking Many test programs do not have regular loop structures. The control dependence allows computations to be regrouped across different control structures including branches and procedural calls. Over a half of the programs, *QSort*, *Health*, *MST*, *TreeAdd*, and portions of *TSP*, use recursion. This is perhaps the first study that finds the exact parallelism in recursive programs and automatically measures the effect of computation regrouping.

The effect of constrained computation regrouping A greedy, unconstrained regrouping algorithm may degrade rather than improve the program locality because too much regrouping of related computations may overflow the cache of a limited size. As an example, we use unconstrained regrouping in *QSort*. The result is shown by the left-hand side graph in Figure 5. The capacity miss rate is increased by at least a factor of two for cache sizes greater than 128. The size needed to fully cache the execution is increased from 16K to 65K elements.

The effect of memory renaming and reallocation Complete memory renaming removes all false data dependences. Without renaming, there is much less chance of regrouping. As an example, we apply constrained regrouping on *BSort* without memory renaming and reallocation. The right-hand side graph in Figure 5 shows that there is no visible difference between the miss-rate curve of the reordered version and that of the base version. The presence of false dependences prevents computation reordering to have an observable effect. As shown before in Table 2, the capacity miss rate of *BSort* can be reduced by a factor of 9 if we remove the false dependences through memory renaming and reallocation. Renaming increases the data size. However, as shown in Table 1, after reallocation, the data size is actually reduced in *BSort* and in all other programs.

5 Potential Uses of the System

The new system is useful to programmers, compiler writers, and computer architects. A programmer can use it to estimate the potential of locality improvement for a program before trying complex transformations for a specific cache system. A compiler writer can use it to study the potential improvement over the current techniques and to experiment with new regrouping strategies without a full compiler implementation. A computer architect can use it to tailor the memory system design to the available program locality.

The limit of program locality is vitally important to

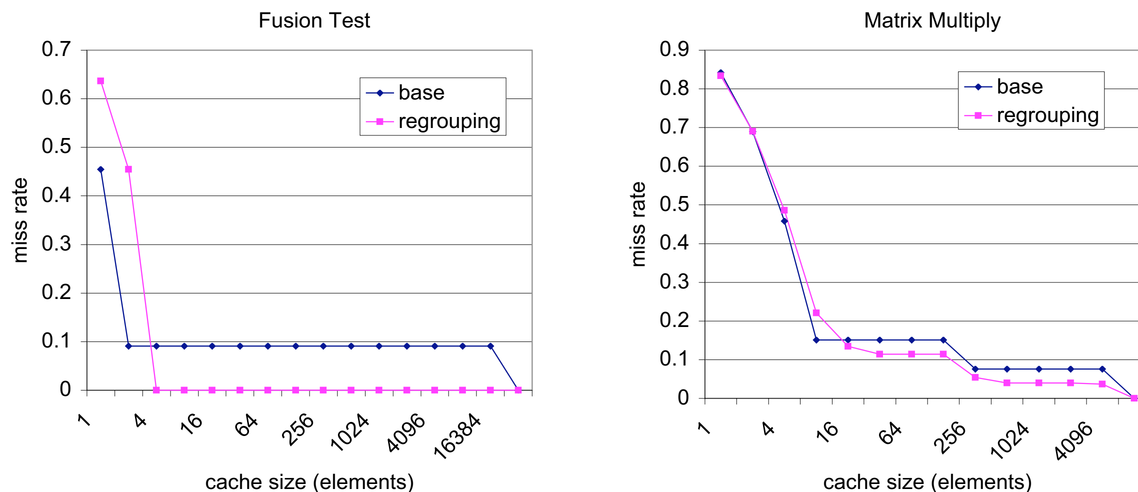


Figure 3: The fusion test and matrix multiply

Table 2: The miss-rate reduction by constrained computation regrouping

cache size	*Fuse*	*MMult*	*BSort*	*QSort*	*Health*	*MST*	*TreeAdd*	*TSP*	**average**
1/32 data size	100%	25%	87%	6%	64%	78%	0%	93%	57%
1/16 data size	100%	25%	89%	3%	94%	87%	0%	61%	57%
1/8 data size	100%	25%	89%	3%	94%	87%	0%	61%	60%
1/4 data size	100%	28%	91%	1%	100%	100%	0%	61%	60%
1/2 data size	100%	47%	92%	1%	100%	100%	0%	100%	68%
average	100%	30%	90%	3%	92%	93%	0%	75%	60%

high-performance computing, where large systems are built to run a few applications. In 1988, Callahan, Cocke, and Kennedy gave a model called *balance* to measure the match between the memory demand of a program and the data bandwidth of a machine [6]. Some programs run well on machines with a memory hierarchy, and some need the high-bandwidth vector memory. The simulation tool can show how the situation changes when program transformations are considered. If the locality of a program cannot be improved by the simulation tool, it is unlikely that the program can utilize cache well even with the best program optimization.

6 Related Work

Numerous studies in the past forty years have characterized the memory behavior of computer programs, modeling them as a trace of accesses to memory pages or cache blocks. Some studied more structured programs and finer grained data—data reuses within and across loop nests [28], per-statement [29], and across program inputs [14]. These studies show that long-distance data reuses cause cache misses, but they do not show how well we can improve the locality of data access. Our earlier work collected instruction traces from a set of Fortran programs and applied a greedy heuristic to shorten the distance of data reuses [12]. They observed that the greedy regrouping sometimes worsens the locality. They did not model the exact control dependence, nor did they use memory renaming.

Loop-nest optimization has achieved great success and become commonplace in industry compilers. Loop fusion improves the locality by combining loops that access the same data. Kennedy and McKinley [23] and Gao *et al.* [17] modeled loops and data reuses as a graph. The complexity of loop fusion was studied using a graph model [23] and a hyper-graph model [12]. Many fusion techniques used greedy heuristics [12, 22]. The graph model does not measure the exact locality. It treats the loop fusion as a clustering process and assumes full cache reuse within a fused loop and no cache reuse between loops. In this paper, we present a new program model based on the bandwidth minimization problem. We treat the computation regrouping as a sequencing process and consider all cases of the cache reuse.

Computation blocking is arguably the most complex form of computation regrouping. Callahan *et al.* described the transformation as unroll-and-jam and gave its legality test [6], which allows a compiler implementation [8]. The later studies, for examples Wolf and

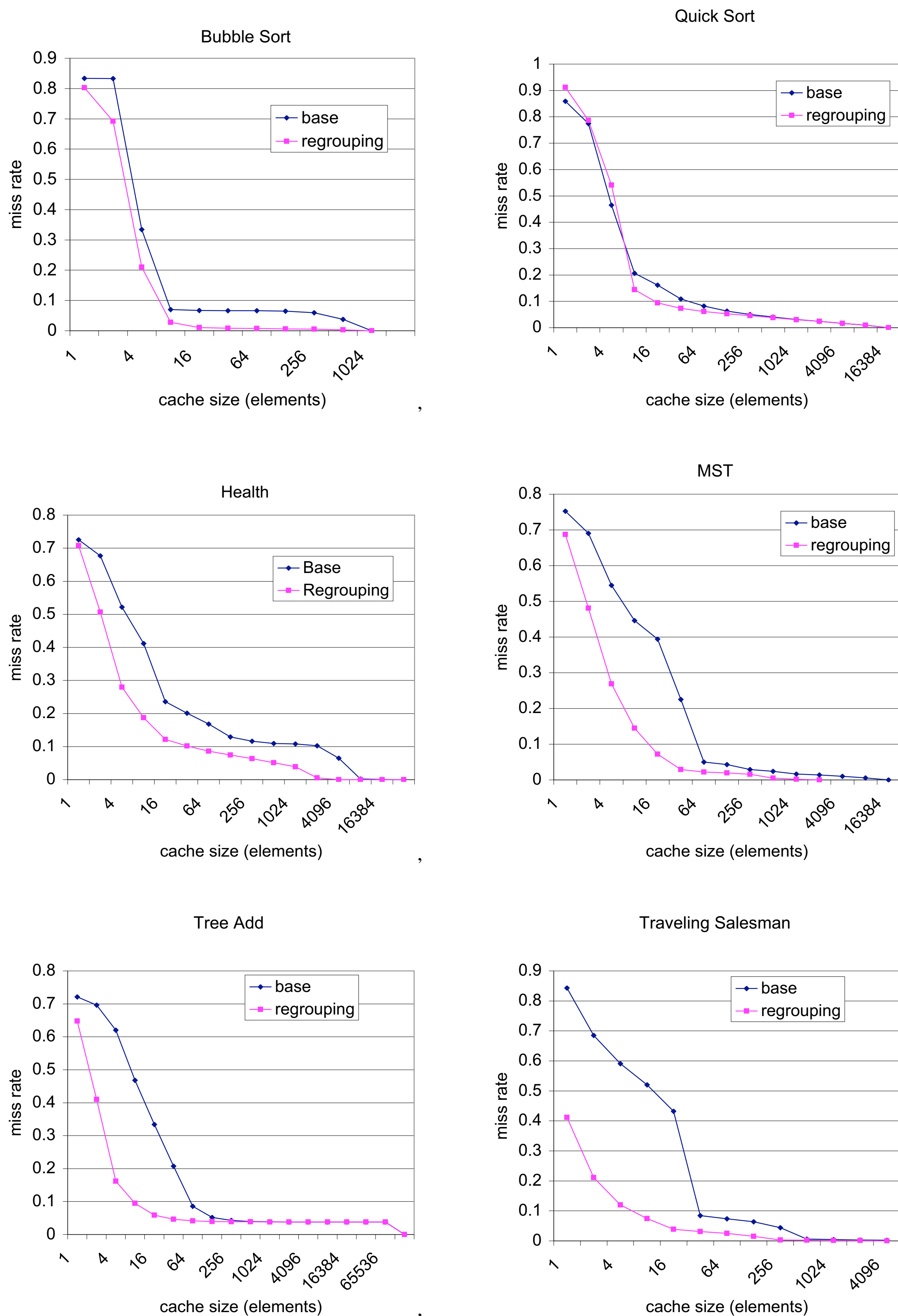


Figure 4: The effect of constrained computation regrouping

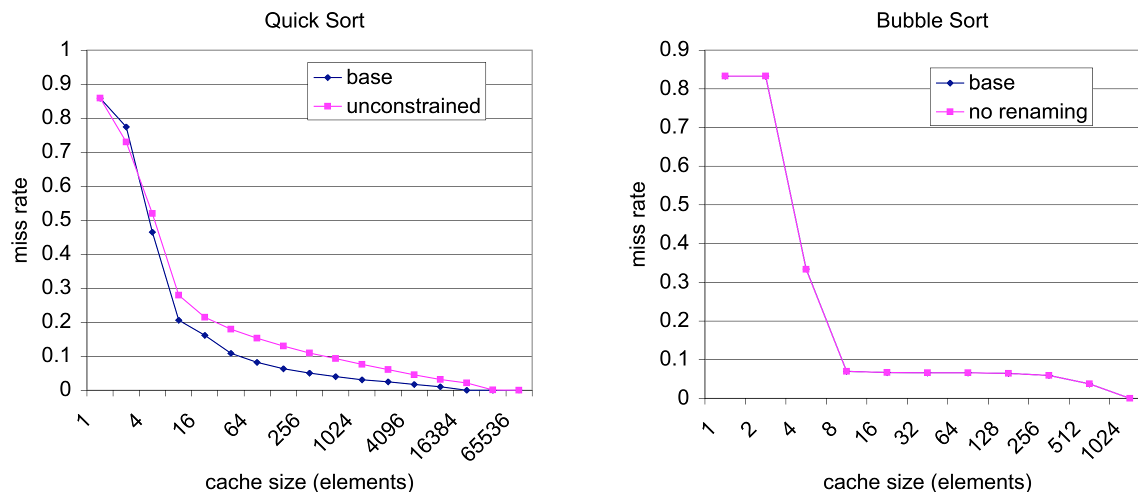


Figure 5: *Quick sort* using unconstrained regrouping and *Bubble sort* not using data renaming

Lam [39], Cierniak and Li [9], and Kodukula *et al.* [24], modeled tiling as an optimization problem in the (high-dimensional) iteration space and approximated the optimal solution in various ways. Loop fusion and tiling may be combined for loops that are all fusible [24, 33, 35, 40, 41]. Computation fusion is also used to transform dynamic programs [11, 19, 30, 36, 37].

Unlike high-level program transformations, the trace-based reordering knows the exact control and data dependence. However, it does not know the high-level control structure needed for tiling. It identifies legal computation sequences, but they may not be coded efficiently by a program. Still, the trace-level simulation can be used to study the potential for a wide range of programs. Pingali *et al.* generalized the idea of loop fusion to computation regrouping and applied it manually to a set of C programs commonly used in information retrieval, computer animation, hardware verification and scientific simulation [32]. The new tool allows these techniques to be studied and their potential measured without labor-intensive manual programming and reprogramming.

Nicolau and Fisher first showed the limit of parallelism at the instruction level when the control and false data dependences are ignored [31]. Later studies added different levels of constraints. Lam and Wilson were the first to model the run-time control dependence [25]. As described in Section 3.2.1, their method was not accurate for recursive procedures. They analyzed binary programs and removed a subset of false dependences involving register-allocated loop index variables. They, as well as most others, used greedy reordering, which is sufficient for the data flow model with an unlimited number of processors. One exception is the work of Theobald *et al.*, who measured the “smoothability” of the parallelism and used it to consider the resource constraint [38]. In comparison, our tool tracks the exact control dependence, applies the general form of renaming and reallocation (and removes *all* false dependences), and uses constrained reordering. Almost all previous limit studies assumed uniform, single-cycle access to memory. On modern machines, the memory speed is orders of magnitude slower than the register and cache speed. Hence the potential gain from locality is as significant as the potential form parallelism, especially for non-numerical programs.

Program control flow is usually analyzed in two ways. If-conversion uses predicates to construct straight-line code and convert the control dependence to the data dependence. Allen *et al.* gave a method that systematically removes the control flow in loops [1]. Ferrante *et al.* modeled control dependence explicitly [16]. Cytron *et al.* gave an efficient measurement algorithm [10]. Later studies followed these two classical approaches. A comprehensive treatment can be found in the textbook by Allen and Kennedy [2].

Data renaming is a special form of memory management. It uses a minimal amount of storage for the active data values (but does not need paging or cache eviction). Huang and Shen studied the reuse of values [21]. Burger *et al.* measured the benefit of the optimal cache management on program traces [5]. As discussed in Section 2, memory and cache management is orthogonal to the problem of computation reordering.

7 Summary

We have shown that optimizing for locality is different from optimizing for parallelism or cache utilization. We have presented a heuristic-based algorithm for constrained computation regrouping. We have designed a trace-driven tool that measures the exact control depen-

dence and applies complete memory renaming and re-allocation. The new algorithm and tool significantly improved the temporal locality of a set of C programs, reducing the number of cache capacity misses by a factor of ten for half of the programs. In addition, complete variable renaming before reordering did not increase the data size when applied before constrained regrouping and memory re-allocation. While the results suggest that optimizing for locality has a great effect, the potential is not uniform across all programs. Therefore, the automatic tool is valuable in allowing a programmer to estimate the benefit of computation regrouping before applying it in complex programs.

Acknowledgement Daniel Williams participated in the initial part of this work. We thank SC'04 reviewers for their comments, especially for pointing out a problem with the use of an example. The latex style file was due to Keith Cooper. Both authors are supported by the Department of Energy (Contract No. DE-FG02-02ER25525). Additional support comes from the National Science Foundation (Contract No. CCR-0238176, CCR-0219848, and EIA-0080124), and an equipment grant from IBM. Special thanks go to Michelle Strout, whose constructive comments and steadfast shepherding gave consistency, order, and polish to the presentation of the paper.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Conference Record of the Tenth Annual ACM Symposium on the Principles of Programming Languages*, Austin, TX, Jan. 1983.

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

[3] J. Backus. The history of Fortran I, II, and III. In Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, 1981.

[4] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[5] D. C. Burger, J. R. Goodman, and A. Kagi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23th International Symposium on Computer Architecture*, Philadelphia, PA, May 1996.

[6] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5(4):334–358, Aug. 1988.

[7] M. C. Carlisle. *Olden: parallelizing programs with dynamic data structures on distributed-memory machines*. PhD thesis, Department of Computer Science, Princeton University, June 1996.

[8] S. Carr and K. Kennedy. Blocking linear algebra codes for memory hierarchies. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, IL, Dec. 1989.

[9] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, California, 1995.

[10] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

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

[12] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *Journal of Parallel and Distributed Computing*, 64(1), 2004.

[13] C. Ding and Y. Zhong. Compiler-directed run-time monitoring of program data access. In *Proceedings of the first ACM SIGPLAN Workshop on Memory System Performance*, Berlin, Germany, June 2002.

[14] C. Ding and Y. Zhong. Predicting whole-program locality with reuse distance analysis. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

[15] C. Fang, S. Carr, S. Onder, and Z. Wang. Reuse-distance-based miss-rate prediction on a per instruction basis. In *Proceedings of the first ACM SIGPLAN Workshop on Memory System Performance*, Washington DC, June 2004.

[16] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[17] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, Aug. 1992.

[18] M. Garey and D. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., New York, NY, 1979.

[19] H. Han and C. W. Tseng. Locality optimizations for adaptive irregular scientific codes. Technical report, Department of Computer Science, University of Maryland, College Park, 2000.

[20] M. D. Hill. *Aspects of cache memory and instruction buffer performance*. PhD thesis, University of California, Berkeley, November 1987.

[21] S. A. Huang and J. P. Shen. The intrinsic bandwidth requirements of ordinary programs. In *Proceedings of the 7th International Conferences on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, October 1996.

[22] K. Kennedy. Fast greedy weighted fusion. In *Proceedings of the 2000 International Conference on Supercomputing*, Santa Fe, NM, May 2000.

[23] K. Kennedy and K. S. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report TR93-208, Dept. of Computer Science, Rice University, Aug. 1993. (also available as CRPC-TR94370).

[24] 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*, Las Vegas, NV, June 1997.

[25] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of International Symposium on Computer Architecture*, Queensland, Australia, 1992.

[26] G. Marin and J. Mellor-Crummey. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of Joint International Conference on Measurement and Modeling of Computer Systems*, New York City, NY, June 2004.

[27] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

[28] K. S. McKinley and O. Temam. Quantifying loop nest locality using SPEC'95 and the perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4):288–336, 1999.

[29] J. Mellor-Crummey, R. Fowler, and D. B. Whalley. Tools for application-oriented performance tuning. In *Proceedings of the 15th ACM International Conference on Supercomputing*, Sorrento, Italy, June 2001.

[30] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. *International Journal of Parallel Programming*, 29(3), June 2001.

[31] A. Nicolau and J. A. Fisher. Measuring the parallelism available for very long instruction word architecture. *IEEE Transactions on Computers*, 33(11), 1984.

[32] V. S. Pingali, S. A. McKee, W. C. Hsieh, and J. B. Carter. Restructuring computations for temporal data cache locality. *International Journal of Parallel Programming*, 31(4), August 2003.

[33] W. Pugh and E. Rosser. Iteration space slicing for locality. In *Proceedings of the Twelfth Workshop on Languages and Compilers for Parallel Computing*, August 1999.

[34] X. Shen, Y. Zhong, and C. Ding. Regression-based multi-model prediction of data reuse signature. In *Proceedings of the 4th Annual Symposium of the Las Alamos Computer Science Institute*, Sante Fe, New Mexico, November 2003.

[35] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation*, Atlanta, Georgia, May 1999.

[36] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

[37] M. M. Strout and P. Hovland. Metrics and models for reordering transformations. In *Proceedings of the first ACM SIGPLAN Workshop on Memory System Performance*, Washington DC, June 2004.

[38] K. B. Theobald, G. R. Gao, and L. J. Hendren. On the limits of program parallelism and its smoothability. Technical Report ACAPS Memo 40, School of Computer Science, McGill University, June 1992.

[39] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[40] D. Wonnacott. Achieving scalable locality with time skewing. *International Journal of Parallel Programming*, 30(3), June 2002.

[41] Q. Yi, V. Adve, and K. Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, Canada, June 2000.

[42] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.

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

[44] Y. Zhong, S. G. Dropsho, and C. Ding. Miss rate preidiction across all program inputs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, New Orleans, Louisiana, September 2003.