Preprint
UCRL-JC-154597

Identifying and Exploiting Spatial Regularity in Data Memory References

T. Mohan, B. R. de Supinski, S. A. McKee, F. Mueller, A. Yoo, M. Schulz,

This article was submitted to *Supercomputing 2003, Phoenix, Arizona*
11/15/2003 – 11/21/2003

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

July 24, 2003

Identifying and Exploiting Spatial Regularity in Data Memory References

Tushar Mohan[1], Bronis R. de Supinski[2], Sally A. McKee[3], Frank Mueller[4], Andy Yoo[2], Martin Schulz[3]

[1] Lawrence Berkeley National Laboratory
Future Technologies Group
Berkeley, CA 94720
TMohan@lbl.gov

[2] Lawrence Livermore National Laboratory
Center for Applied Scientific Computing
Livermore, CA 94551
{*bronis, ayoo*}*@llnl.gov*

[3] School of Electrical and Computer Engineering
Cornell University
Ithaca, NY 14853
{*sam,schulz*}*@csl.cornell.edu*

[4] Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7534
mueller@cs.ncsu.edu

Abstract

The growing processor/memory performance gap causes the performance of many codes to be limited by memory accesses. If known to exist in an application, strided memory accesses forming *streams* can be targeted by optimizations such as prefetching, relocation, remapping, and vector loads. Undetected, they can be a significant source of memory stalls in loops. Existing stream-detection mechanisms either require special hardware, which may not gather statistics for subsequent analysis, or are limited to compile-time detection of array accesses in loops. Formally, little treatment has been accorded to the subject; the concept of locality fails to capture the existence of streams in a program's memory accesses.

The contributions of this paper are as follows. First, we define *spatial regularity* as a means to discuss the presence and effects of streams. Second, we develop measures to quantify spatial regularity, and we design and implement an on-line, parallel algorithm to detect streams — and hence regularity — in running applications. Third, we use examples from real codes and common benchmarks to illustrate how derived stream statistics can be used to guide the application of profile-driven optimizations. Overall, we demonstrate the benefits of our novel regularity metric as a low-cost instrument to detect potential for code optimizations affecting memory performance.

This work was performed under the auspices of the U.S. Department of Energy by the University of California Lawrence Livermore National Laboratory under Contract W-7405-Eng-48, UCRL-JC-XXXXXX and National Science Foundation awards 0073532 as well as NSF CAREER CCR-0237570. The authors would like to thank David Bailey, the Performance Evaulation and Research Center (PERC), and the National Energy Research Scientific Computing Center (NERSC), for providing computing facility use for experiments. This work was performed while the first author was a graduate student in the School of Computing at the University of Utah.

1 Introduction

Processor speeds have increased much faster than memory speeds, and the disparity prevents many applications from making effective use of the immense computing power of modern microprocessors. A variety of approaches have been used to improve the memory performance of scientific codes [12]; we survey but a few of these, since they are too numerous to list here. Which optimizations apply to a given code depend highly on the memory access characteristics of that code. For instance, stencil computations such as iterative 3D PDE solvers often benefit from tiling optimizations [16, 17, 20]. Likewise, adaptive mesh generation codes from computational fluid dynamics and N-body solvers from astrophysics or molecular dynamics often benefit from scatter/gather operations and address remapping [6] or from code and data restructuring optimizations [9, 13, 5, 8].

Most high-performance code optimizations attempt to exploit *locality of reference*. Indeed, the concepts of spatial and temporal locality have been well understood for decades. Unfortunately, current notions of locality are limited to references with proximate or repeating accesses, and cannot capture the existence of other patterns. On the other hand, modern processor architectures and memory subsystems can exploit many of these other patterns, with *streams* (successive memory references with a constant difference in address) being the most common. For instance, the Power3 processor can detect and prefetch strided accesses at a cache-line granularity [15]. The Impulse memory system [6] can prefetch and gather streams within the controller, shipping dense cache lines to the processor's cache hierarchy. To better understand and improve the performance of codes on such architectures, it is useful to extend the concept of locality to include strided patterns (as a first step — certainly, there exist other kinds of patterns that could be described and exploited, but they are beyond the scope of this paper). We define *spatial regularity* as *the likelihood that a memory access will form or continue a strided sequence*. In Section 2.1 we develop a metric to quantify spatial regularity.

The scope for applying many optimizations is intimately related to the presence (or absence) of streams in an application. To a first-order approximation, spatially regular or streamed computations are amenable to a one set of optimizations, and "irregular" applications to another. Poor cache performance for applications in the former class results primarily from one or more of: self/mutual interference causing conflict misses among long streams, large-stride streams causing compulsory and capacity misses, and long streams causing capacity misses by overrunning the cache or TLB.

Irregular applications (or irregular portions of applications) exhibit no obvious patterns in their use of memory. For instance, this behavior could arise from using an indirection vector (IV) to access memory. Such applications generally have large memory footprints and make poor use of cache. Many applications in fluid and molecular dynamics fall in this class, for instance. We have developed a tool that allows us to broadly classify applications (or code sections) into two categories — regular and irregular — on the basis of their regularity metric. We target the two categories with different optimizations. We consider tiling, sequential and stream prefetching, copying and remapping, layout changes and loop transformations for regular applications. For irregular applications we consider prefetching the indirection vector/pointer, software prefetching and code restructuring. We refine our candidate set of optimizations by considering other stream characteristics, such as counts, lengths and strides.

The rest of this paper is organized as follows. In the next section we develop the concept of spatial regularity, and define a metric to quantify it. In the following section, we discuss how the metric, along with other relevant stream statistics, can guide selecting optimizations. In Section 3, we present our online stream detection algorithm. In Section 4, we discuss our stream detection framework, and the use of PAPI and Dyninst therein. Next, in Section 5, we present regularity data for real applications and common benchmarks, and we discuss specific optimizations. Finally, we discuss related and future work.

2 Regularity

The principle of locality is well known and recognized for its importance in determining application memory performance. Traditionally, it is considered to have two primary components. A classic definition of these is found in Hennessey and Patterson's computer architecture text [14]:

- *Temporal locality* (locality in time) – If an item is referenced, it will tend to be referenced again soon.
- *Spatial locality* (locality in space) – If an item is referenced, nearby items will tend to be referenced soon.

Unfortunately, these concepts and their measures alone are insufficient to describe all the key application properties that intelligent memory systems can exploit. Consider the following code fragment for a matrix transpose:

```
double A[N][N], B[N][N];
int i, j;

for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    A[i][j] = B[j][i];
```

With a row-major layout, successive iterations of the inner loop access contiguous elements of A. This pattern exhibits high spatial locality, and will therefore benefit from caching. For such a case, locality measures provide an accurate assessment of performance gains as a result of caching and prefetching on traditional controllers.

In contrast, all accesses to B are separated in memory by $(N-1)$ elements. This sequence exhibits poor spatial locality (with respect to B) and leads to low cache utilization. On traditional prefetching systems, few (if any) performance gains can be expected through data prefetches of B's elements. Despite this lack of locality, however, a very predictable pattern exists for B's accesses. This can be exploited by smarter memory subsystems that detect large-strided streams (in this case the accesses to B), prefetch successive stream elements, and possibly send them to the processor in dense cache blocks [1, 10, 19]. Locality metrics do not give an accurate depiction of application performance in the presence of such memory subsystems. Streams are common in many applications, including compression and archiving, file I/O utilities, image processing, string manipulation routines, and partial differential equation solvers [4].

In order to describe these additional access characteristics, we extend the principle of locality to regularity. As with locality, regularity can be broken into two components:

- *Temporal regularity* (regularity in time) – If a sequence of items is referenced, the entire sequence is likely to be referenced again soon.
- *Spatial regularity* (regularity in space) – If referenced items form a strided sequence (stream), items that continue the sequence will tend to be referenced soon.

Temporal regularity simply extends temporal locality to sequences of items – with no relationship needed between elements forming a sequence. It reduces to temporal locality when the sequence consists of a single item. Spatial regularity requires a linear relationship between the accesses forming a sequence. It reduces to a special case of spatial locality when the stride is small. Most computer architecture features that exploit regularity benefit programs that exhibit spatial regularity. However, compiler-based approaches for exploiting temporal regularity have been identified [7], and we expect that other, hardware-based techniques will emerge as the principle of regularity becomes better understood. In this paper, we focus on detecting and exploiting spatial regularity.

2.1 Spatial Regularity Metric

Regular sequences, or streams, are precisely arithmetic progressions, defined as:

$$x_n = x_{n-1} + c$$

where c is a constant and x_n is the n^{th} reference in the regular sequence. The following metric can be used to quantify the

spatial regularity of an application (or code section):

$$R_{spatial} = \frac{\sum \mid s_i \mid}{N}$$

where $\mid s_i \mid$ is the length of the i^{th} sequence and N is the total number of references. Effectively, the metric is the fraction of all references that belong to *some* stream. Since we do not allow a reference to be included in more than one sequence, the metric is a positive number not greater than unity. Higher metric values imply greater spatial regularity. The example code segment above exhibits high regularity: each memory reference in the code segment belongs to a stream. Consequently, the metric for the code is exactly one.

The regularity metric allows us to classify applications into two broad categories: regular and irregular. This classification is useful in that applicable optimizations are often different for regular and irregular codes: the former benefit from stream/sequential prefetching, loop transformations, layout changes, and stream copying and remapping; while the latter are aided by software prefetching, code restructuring, and scatter/gather optimizations. Our experience indicates that applications widely considered "regular" have a metric value greater than 0.80, while irregular codes have a metric value less than 0.65. We suggest and implement optimizations partly based on this classification. It is very possible that some applications will have a metric in between these values; in that case other stream statistics may highlight the nature of the application (regular or irregular).

2.2 Stream Statistics

Most real programs contain thousands of streams of varying lengths, strides and starting addresses. To further the understanding of the memory behavior of applications in the presence of such streams, we introduce additional important stream statistics.

Mean Stream Length: The mean length of streams qualifies the regularity of applications, since long streams are easier to optimize for effective cache use. Potential optimizations include prefetching, remapping, creating super-pages, and blocking (see `mgrid`, `swim`, and `su2cor` in "Results"). Standard deviation in stream length is a worthwhile companion measure, indicating the significance of the mean in depicting stream length information. Irregular applications often have high variance in stream lengths (see `umt98` and `CG` in "Results"). This variance often arises from long streams existing in the form of indirection vectors and short streams occurring on the indirect accesses through these vectors.

Mean Stream Stride: Streams with low stride often enjoy fewer compulsory misses, since successive stream elements in the same or next prefetched line. Reducing stream stride through copying, remapping, and additional prefetching (apart from the usual one line ahead sequential hardware prefetch) are ways to improve cache use.

Aggregate Stream Counts: Many real codes have stream counts in the thousands, but their smaller functions often have few streams (< 10). Aggregate stream counts can help select possible optimizations: regular codes with few streams and poor cache performance can benefit from array padding and code restructuring optimizations, while having many long streams may suggest an entirely different optimization, such as tiling. Our tool gathers counts of stream lengths, making it possible to answer questions such as "*How many streams greater than a particular length are detected?*". We use this information to perform a code-restructuring optimization in `gzip`.

Interleave Statistics: Measures such as the number of streams with a certain fraction of elements in a temporally interleaved pattern can help determine more precisely *when* and *how* a stream occurs in context of the program. We therefore implement an algorithm to detect temporal interleaving information of streams. Consider:

```
for (i=0; i< N; i++) {
  A[i] = B[i] + C[i]; }
```

Assuming `i` is in a register, every third memory access is to the same stream.

2.3 Exploiting Streams

Optimizing compilers automatically apply a variety of loop and other high-order transformations at high optimization levels. Occasionally, however, a source modification or compiler directive is necessary to enable the application of certain optimizations that might otherwise not be permissible/identified. For instance, tiling optimizations require a loop interchange, which may be illegal until the code is suitably modified. Compiler flags and directives drive certain compiler optimizations and permit more aggressive uses of existing ones. For instance, the MIPS compiler accepts directives for aggressive prefetching, inner loop fission, and unrolling [21]. Indisputably, a programmer's understanding of the need for a certain optimization can aid in its use. Streams and stream characteristics discussed in Section 2.2 receive special merit, for they:

- often contribute to the bulk of memory accesses in loops of regular codes;
- are easier to optimize for, since their strided patterns makes their interaction with the memory hierarchy predictable; and
- can often be treated as a unit for purposes such as data and computation restructuring.

In Table 1 we relate stream statistics and the regularity metric to potential optimizations. We provide here an intuition into the relationship for one such optimization: tiling.

Loop tiling is a combination of strip mining and loop interchange. It reduces capacity and conflict misses by dividing the iteration space into tiles and transforming the loop nest to iterate over them [27]. By definition, a code that benefits from tiling has multiple nested loops and a working set larger than

the cache hierarchy. The memory accesses are often streamed, since they result from array accesses in nested loops. Tiling candidate code sections will therefore have high regularity, with numerous long streams. Table 1 presents simplified numbers for these adjectives – "high" (regularity), "long", and "many" (streams). We generated these guidelines from empirical results for applications on the Power3. These "rules of thumb" attempt to provide a empirical substitute for a complex algebraic analysis that relates architectural parameters, such as cache/TLB sizes, to stream statistics. Section 5 presents stream data for popular codes. In some cases the data leads to new optimizations, and in others it affirms the applicability of optimizations known to improve the code's performance.

Optimization	Characteristics	Code
prefetching	long streams (100+) with short/moderate strides (<10)	`gzip`, `swim`, `mgrid`
tiling	many (10k+) long streams (100+ elements), some with large strides (10+); many scalar streams	`mgrid`, `swim`, `su2cor`, `matmult`, `3D_Jacobi`
loop fission	many short streams; many scalar streams from register pressure; interleaved long streams	`su2cor`
loop fusion	long streams with repetition	
loop interchange	very large strides (32+)	`FT (3D FFT)`
data layout	few, long (100+) streams with short strides (<8); high cache miss-rates	`swim`, `su2cor`
copying, stream remapping	long streams (100+) with large strides (32+)	`FT`, `BT`, `matmult`
superpaging	long streams, spanning many pages (10+); TLB misses and page faults	`gzip`
loop unrolling	many scalar streams due to register pressure	
code restructuring	a few long streams with short stride (<8); high cache/TLB miss rates	`gzip`, `su2cor`
scatter/gather using indirection vector (IV)	irregular(<0.6)	`umt98`, `CG`

Table 1: Optimizing (ir)regular applications

3 Algorithm to detect streams

To measure spatial regularity and to classify codes using the metrics and statistics introduced above, we have developed an efficient algorithm to detect streams by analyzing a program's loads and stores as they are executed. We perform an online analysis, thus avoiding the space and time complexity of archiving traces on auxiliary storage. We allow streams to be interleaved with each other and with non-stream references. This enables us to identify streams in the presence of other accesses. However, constraints on space require us to periodically discard older trace data to make room for new references. This *aging* prevents the detection of streams interspersed with a large number of intervening accesses[1].

The algorithm uses two main data structures:

[1]However, once a stream has been identified, we are no longer forced to age it for space limitations, as the entire stream is compactly represented.

```
WHILE new reference exists DO
  Modulo−increment col; /* move window */
  /* Add reference to pool */
  pool[0][col] := new reference;
  IF reference extends stream IN stream table
  (perform a hash lookup to check), THEN
    Update length of stream in stream table;
    Mark column in pool (shaded in example)
  ELSE
    /* Compute differences between new element and previous ones */
    FOR i := 1 TO (window size − 1) DO
      IF (col >= i), THEN
        c = col − i;
      ELSE
        c = col+w −i ;
      END IF;
      pool[i][col] := pool[0][col] − pool[0][c];
    END FOR;

    /* Search for streams of minimum length 3 */
    found := FALSE;
    FOR i := 1 TO (window size − 1) DO
      IF (col >= i), THEN
        c = col − i;
      ELSE
        c = col+w −i ;
      END IF;
      IF column c is already marked, THEN
        continue;
      END IF;
      FOR k := 1 TO (window size − 1) DO
        IF pool[i][col] == pool[k][c] THEN
          found = TRUE;
        END IF;
      UNTIL found;
    UNTIL found;

    IF found THEN
      Enter stream in stream table;
      Mark corresponding columns in pool (shaded in example);
    END IF;
  END IF;
END WHILE;
```

Figure 1: On-line algorithm to detect streams

Stream Table: The stream table contains a compact description of streams that have already been detected. The table is stored as a chained hash, with the *expected successor to the stream* serving as the hash key. Each node in the table is a triple consisting of the `start address`, `length`, and `stride` of the stream. In addition, we store the age of the last stream element in order to facilitate aging.

Pool: The pool contains *recent* references that have not yet been detected to be part of a stream. As new addresses are referenced, this window of active addresses expands to fill the storage structure of the pool. Once filled, the addition of each new reference causes an earlier reference to be discarded from the pool after a scan for streams, cyclically rotating the active window through the Poole's physical storage structure. In determining the existence of streams, elements of which may be separated by arbitrary numbers of intervening accesses, *differences* between elements of the pool are computed. To reduce the computational complexity in repeatedly determining the differences between existing elements as new elements are added, we store a set of differences with each reference in the pool. Given this pool structure, detecting streams becomes a matter of finding a sequence of elements such that the differences between successive elements match. In our implementation, the pool is organized as a statically allocated, $w \times w$, two-dimensional array, where w is the window size (a compile-time constant).

Multithreaded programs can easily employ our stream detection algorithm. A unique thread running this algorithm – and maintaining its own stream table and pool – can be dedicated to receiving the accesses of a single thread in the instrumented application. By maintaining separate data structures in each

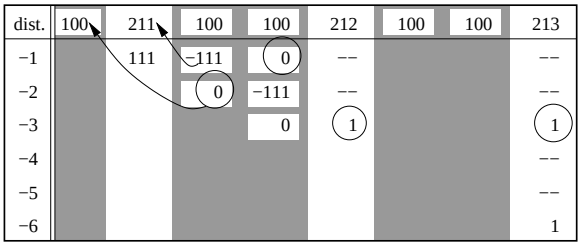

Figure 2: Snapshot of the reservation pool

thread, the detection of streams across application threads is not possible. This is desirable, since most architectural features that exploit streams use processor-specific caches, and cannot take advantage of cross-thread spatial regularity. Another benefit of thread-specific data structures is that it allows our algorithm to efficiently scale for parallel applications. An ancillary benefit of this stream detection process is that it can be leveraged to quantify temporal locality, which corresponds to streams with zero stride. The algorithm, omitting the details of aging and without distinguishing between access types, is presented in Figure 1. We illustrate the application of our algorithm on the following highly regular sequence of accesses, where we do not distinguish between access types for simplicity:

```
100, 211, 100; 100, 212, 100; 100, 213,
        100; 100, 214, 100; ...
```

The pool can be viewed as a table as shown in Figure 2, which depicts a snapshot of the pool after encountering the first eight references. The header row shows the referenced locations. Each column contains the difference between the value in the current column header and the value in a preceding column (see "Compute differences" in Figure 1). The particular element used for calculating the difference depends on the row for which the difference is to be computed. The first row (below the header) consists of the difference between an element and its immediate predecessor (distance -1), indicated by the upper arrow. The second row consists of differences between an element and its penultimate predecessor (distance -2). To capture streams within a window size w, we need only compute the differences above the diagonal of the pool table. In our implementation we *effectively* double the window size by computing differences below the diagonal as well. Elements determined to be part of a stride are removed from the table. In the example above, on seeing the third `100` (assuming a minimum length of three), we identify a stream by observing the two corresponding differences of `0` (circled) in a transitive relationship. Consequently, we insert a stream of `<100,3,0>` in the stream table. The columns containing these stream elements are marked and are not used in future scanning for streams (until the elements age). We keep these marked slots (rather than filling them with new elements immediately) to maintain a consistent window size. The marked columns are shown shaded in the figure. The later `100`s are immediately observed (by way of a hash lookup) to belong to the stream, and the stream fields are modified to `<100,5,0>` on receiving the fifth `100`. On seeing `213`, a new stream is identified by observing an identical difference of `1` (circled) for the transitive relation between 211, 212 and 213. At this point, `<211,3,1>` is inserted in the stream table, and the columns for these elements are marked to indicate their non-participation in further stream detection.

Although not mentioned in the outline of the algorithm, we implicitly assume that sequences must have a minimum length to qualify as streams; the algorithm in Figure 1 and the example discussed assumed a minimum length of three. The value of metrics computed on streams is dependent on this parameter. The worst case complexity of the algorithm is:

$$C = O(N \times (k + (1 - R_{spatial}) \times w^2)))$$

N is the total number of references, w is the window size, k is the number of streams *detected*, and $R_{spatial}$ is the *observed* regularity metric. The quadratic dependence on w – the window size – arises from scanning the pool for streams on adding a new reference. In regular codes a majority of the references belong to some stream, and require no pool scanning, merely a hash lookup. The hash lookup is bounded in the worst case by an $N \times k$ complexity (where k are the number of streams), and in practice has a far lower constant than k, since most references extend a recently modified stream, which lies at the beginning of the hash chain. This analysis explains the paradox, where increasing window size (w) reduces the overhead for certain stencil codes. The explanation lies in the sharp increase in the *observed* regularity metric ($R_{spatial}$) resulting from streams with successive elements widely separated, now fitting in the larger pool, and hence getting identified as such.

4 Dynamic Stream Detection Tool

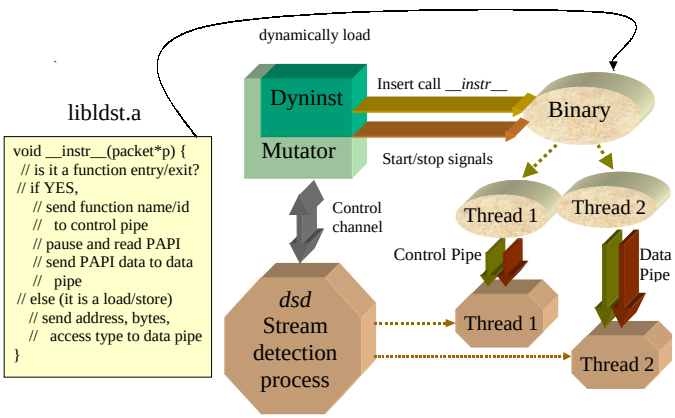

Figure 3: Stream detection framework

Figure 3 shows the setup for our stream detection tool. It consists of three components: `dsd`, `mutator` and the binary to be instrumented. The `mutator` is a generic application that uses the *Dyninst* library to instrument the binary of interest. The instrumentation is used to introduce hardware performance monitoring at function entry/exits and to handle load/store information at memory accesses.

We have written a single function, `__instr__`, to handle both scenarios — the arguments determine the context. The

mutator can be configured to instrument specific functions and modules rather than the whole binary. Once the binary has been successfully instrumented, the mutator starts dsd, the stream detection module. The mutator passes control information — such as its process ID, and the functions instrumented and their *ids* — to dsd using the control channel (shown by the bi-directional arrow between the mutator and dsd). The mutator then starts the instrumented binary. The first time __instr__ is called, it sets up one-way data and control pipes between itself (the instrumented application) and dsd. The control pipes carry information about function entry/exits, while the data pipes are used to pass memory access information and counter values obtained from PAPI calls.[2] Subsequent calls to __instr__ pass hardware performance counter values or memory access information (type, address, and number of bytes) to dsd. The kernel automatically blocks the application when the data pipe fills up, since dsd usually processes data slower than it is generated. If a stop-sampling condition for a particular function is reached (e.g., a limit on number of samples), dsd signals the mutator through the control channel. The mutator pauses the application, deletes the inserted snippets from the function, and resumes the application.

The user may not know which are the performance-critical code sections, and can use our tool to dynamically instrument all (or some) of the functions with PAPI calls to first determine low-level performance statistics, such as memory load/store counts, cache/TLB misses and floating-point operations. The list of functions in various modules can be determined using a probe feature of Dyninst, and requires no source code. In this simple form, our tool provides portable and dynamic function-level performance statistics. Once the functions of interest are known, their memory accesses can be instrumented (again, dynamically), to obtain regularity and stream statistics at a function-level granularity. We provide a number of different *sampling modes* to accommodate differing needs:

Complete: The chosen functions are sampled for an entire application run.

Max samples: An upper limit is provided on the number of samples of the function(s). After the requisite number of samples, the instrumentation is removed. This is useful if many calls are made to the function(s).

Periodic sampling: A sample period and an inter-sample interval is provided for long-running functions. During the inter-sample interval, *all* instrumentation is disabled.

Convergence sampling: A user-prescribed value is specified such that sampling continues until the fractional change between measures from successive samples falls below that threshold. This method is useful for functions with varying behavior between successive calls.

The stream detection program — dsd — can work alongside compiler-implemented static memory access mechanisms, too. The Portland Group's compilers can instrument memory accesses when given a command-line option [24]. We have leveraged our tool to perform stream detection (and regularity measurements) on applications instrumented by these compilers. A significant limitation of any static instrumentation approach is its high run-time overhead, since the instrumentation remains in place for the entire application run. Our instrumentation via Dyninst allows function-level regularity profiling. However, the user can determine stream and regularity information for *arbitrary* code sections by manually inserting sentinel calls to __instr__. Such a technique can help determine regularity and stream information in specific loops.

Overhead is an important issue for any approach that instruments the code/binary. Incorporating Dyninst's ability to disable, insert, and remove instrumentation at run time gives the user more control over instrumentation overhead costs. For most results mentioned in Section 5, the overhead factor is between 50 and 500. Nevertheless, the overhead depends on the extent and duration of instrumentation applied by the user, and may vary widely.

For implementing profile-driven optimizations, our framework is best applied to applications whose memory access patterns do not change significantly with different input datasets. Unlike compile-time analysis techniques, in which regularity measurements can be predicated on symbolic constants [23], our measurements are for *actual* runs. If our tool is used to implement dynamic optimizations, then this need for a typical profile is removed. Our framework can be used efficiently for multithreaded applications: dsd can be configured to use pThreads or OpenMP for parallelization. Note that the current release of the Dyninst library is not multithread-safe, but the forthcoming release is expected to be. In the meantime, we have applied our stream detection process to compiler-instrumented parallel applications, obtaining meaningful results.

5 Results

This section presents regularity metrics obtained with our tool and discusses how these metrics guide application optimization. The primary platform for the regularity and PAPI results is an RS/6000 SP running AIX 4.3 on Power3 processors. We use a single processor for all experimental results presented here. The C and FORTRAN compilers xlc and xlf, versions 5.0.2.2 and 7.1.0.1, respectively, are used with optimization flag -O2, unless otherwise stated. We use a MIPS R10000/IRIX6.5 system to implement an optimization in gzip.

We apply our tool to three real applications (gzip, umt98, and smg2000), numerous benchmarks (SPEC: mgrid, swim, su2cor; NAS/NPB: FT, BT, CG), and a couple kernels (3-D Jacobi and dense matrix multiply). gzip is the GNU file compression utility. umt98 and smg2000 are part of the ASCI Purple benchmark suite. umt98 is an irregular mesh transport code with poor memory performance and substantial use of indirection vectors; smg2000 is a semi-coarsening multigrid solver. mgrid and swim are stencil codes that benefit from tiling and stream prefetching [25]. su2cor applies a Monte-Carlo method to compute the masses of elementary particles using the Quark-Gluon theory; it is known to benefit from inter-variable padding [26]. FT is a 3D Fast Fourier Transform benchmark

[2] Actually, we multiplex the two information flows over a single channel for efficiency and synchronization reasons.

known to benefit from copying array transpositions [3]. BT solves multiple, independent systems of non-diagonally dominant, block tridiagonal equations; it benefits from blocking and copying/remapping. CG is a conjugate gradient sparse matrix inversion code that is aided by scatter/gather optimizations at the memory controller [6].

Our results are promising: for gzip we uncover and implement two optimizations that reduce overall memory stalls by a few percent; in FT (a Fast Fourier Transform benchmark) we implement a loop interchange that reduces overall TLB misses and memory stalls by 58% and 8%, respectively. In many cases, our tool suggests optimizations previously known to benefit the code, while in some it suggests new ones. In most analyzed codes, the tool indicates the specific function(s) to optimize and a small set of potential optimizations. Table 2 lists the regularity statistics for the codes. The sampling mode — convergence, periodic, or fixed — depends on the performance results in the first pass, and differs according to the application. The "Optimization" column lists a subset of the optimizations that are suggested from Table 1, i.e., those that are known to actually improve the code based on previous research or from our experiments. umt98 and smg2000 are exceptions: the optimizations listed for them are not supported by earlier research, and implementing them is an area of future work. Here we detail how our tool guides optimization of gzip and FT. Full results can be found elsewhere [22].

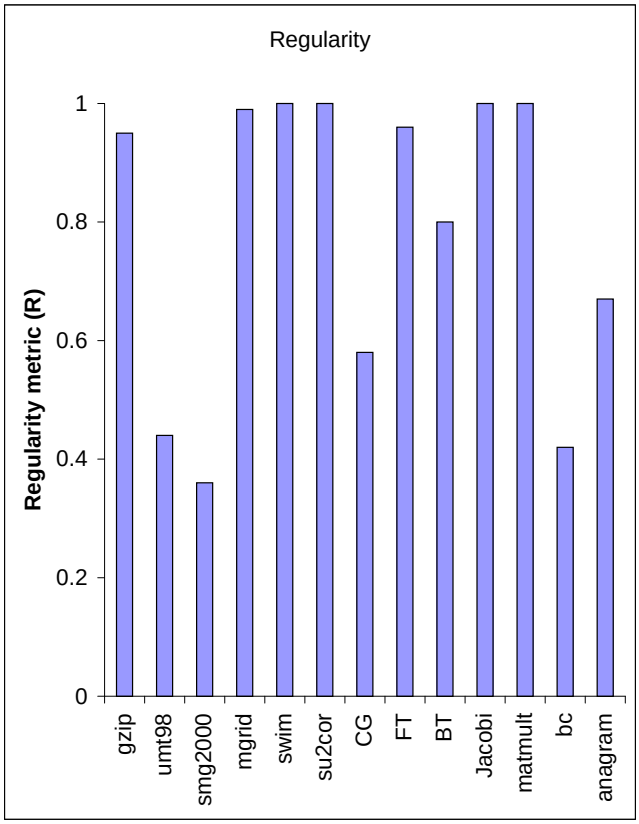

Figure 4: Regularity metric

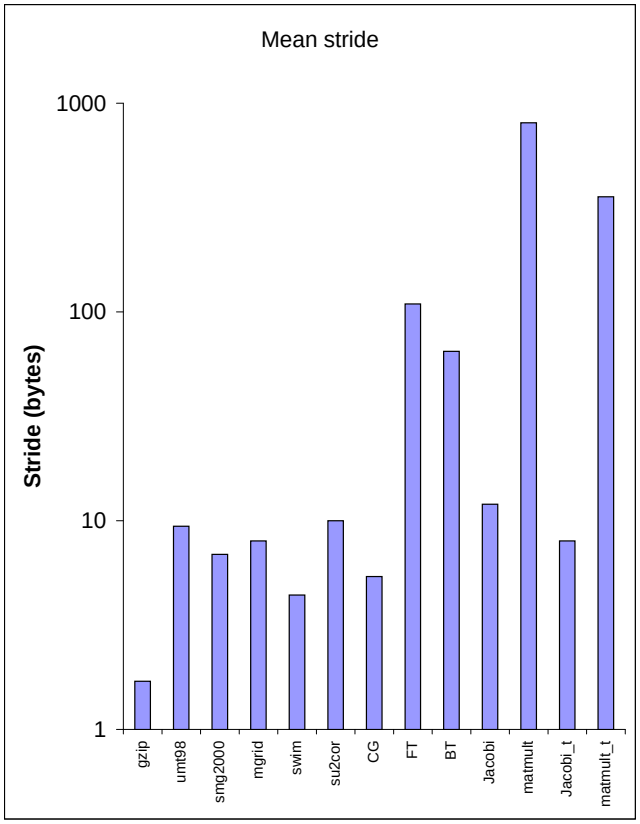

Figure 5: Stream strides

Figure 4 shows the regularity metric for all applications. Notice the high regularity in stencil codes such as mgrid, swim, and Jacobi, and the far lower metric values for indirection-vector codes like umt98 and CG. In Figure 5 we see mean stream strides on a logarithmic scale. In general, high stride values can imply the need for loop interchange and copying/remapping optimizations for regular codes. We implement a loop interchange in FT and suggest copying/remapping for BT. Hand-tiling significantly improves the performance of the matmult and Jacobi kernels and correspondingly reduces their mean stream strides. gzip's low mean stride is the result of a byte-by-byte scan for repeated strings. Mean stream lengths and the distribution of streams with lengths is shown in Figure 6. Applications with many long streams benefit from tiling: mgrid, swim, su2cor, Jacobi, and matmult fall into this category. Tiled codes exhibit far lower mean stream lengths and variances; we see this when we compare untiled matmult and Jacobi with their tiled counterparts. A high variance in stream lengths is present in many irregular codes, such as umt98 and CG. Regular codes with long streams benefit from prefetching (sequential or stream), and this is observed for gzip, swim and mgrid.

5.1 gzip

Gzip is a popular, free GNU compression utility, based on a variation of the Lempel-Ziv 1977 (LZ77) algorithm, and avail-

Program	Reg.	Streams					Stream Length		Stride	Optimization
		Total	]4-32]	]32-128]	]128-16384]	> 16384	Mean	Dev.		
`gzip`	0.95	5402	4625	76	17	584	3552	170	1.71	aggressive prefetching, super-paging and code restructuring
`FT`	0.96	755750	734755	20481	513	1	12	23	109.3	loop interchange, array transposition
`umt98`	0.44	310655	307386	1818	1377	44	31	570	9.4	scatter/gather using IV in `snswp3d`
`smg2000`	0.36	100675	100519	78	74	4	13	108	6.95	code restructuring
`mgrid`	0.99	82359	52	82307	0	0	91	3.2	8.0	prefetching and tiling in `RESID`
`swim`	1.00	38259	1052	2	37188	17	614	223	4.4	aggressive prefetching, blocking, padding
`su2cor`	1.00	50274	33688	289	16317	0	1208	52	10.0	code restructuring, fission, padding, tiling, prefetching
`CG`	0.58	184954	143587	40555	805	7	46	518	5.4	scatter/gather using IV
`BT`	0.80	1470357	1419110	51247	0	0	9	4.8	64.8	copying, base-stride remapping
`Jacobi` (Untiled)	1.00	77127	99	76832	196	0	85	26	12.0	tiling
`Jacobi_t` (Tiled)	1.00	61741	61741	0	0	0	11	1	8	none
`matmult` (Untiled)	1.00	88683	8081	0	80602	0	183	4.2	806.0	tiling
`matmult_t` (Tiled)	0.97	1788850	1788805	0	45	0	10	0.5	356.3	none

Table 2: Regularity statistics

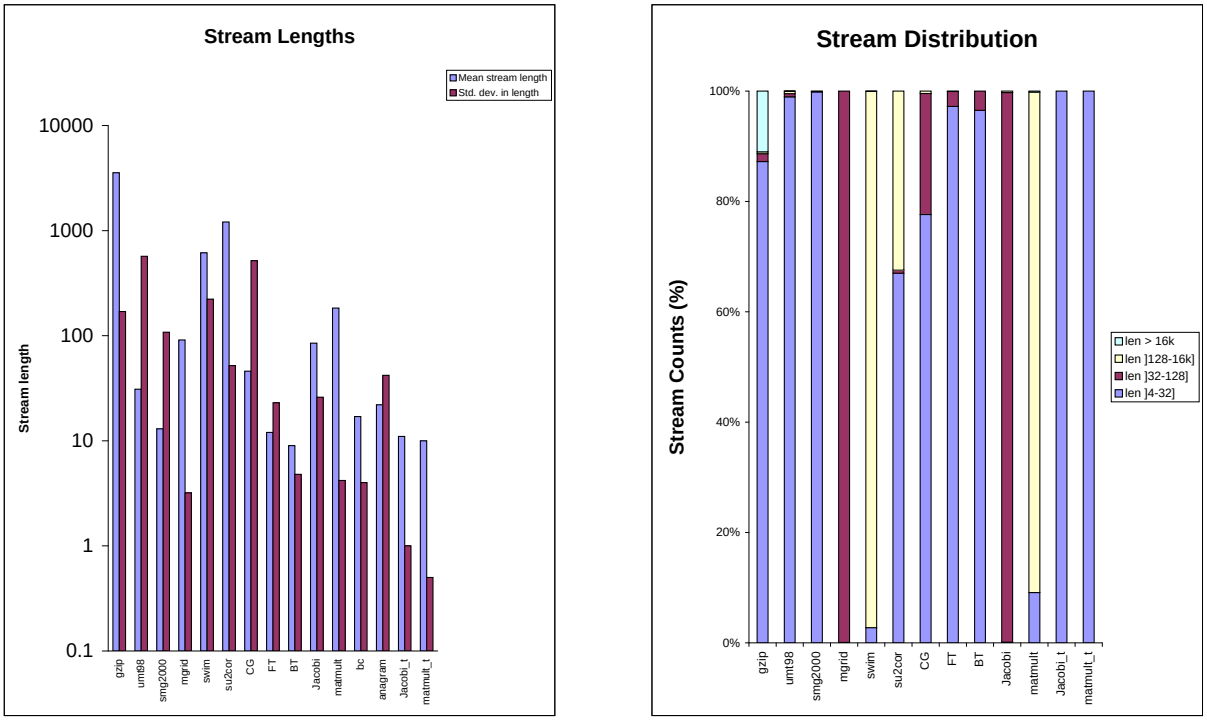

Figure 6: Stream lengths

able for a variety of Unix platforms. We expect `gzip` to be regular on the basis of its compression algorithm. The code has little computation and is memory bound (for our purposes we ignore all file I/O activities). Table 3 shows a summary of the results obtained from PAPI and a single-pass regularity sampling of all the functions, in the compression of a 9MB binary using `gzip` version 1.2.4. The table lists functions with the most memory stalls in decreasing order.

`gzip` has a very high regularity (0.95) and mean stream length (3552) and a significantly low mean stride (1.71). The stream data collected shows that the two functions with the most memory stalls — `fill_window` and `deflate` — also have very high regularity metrics. We focus on these func-

function	STALLS (million)	TLB_MISS (thousand)	L1_MISS (thousand)	Reg.	Streams	Len.	Stride
fill_window	345	7	461	1.00	4	32766	2.0
deflate	169	367	452	0.96	4917	3867	35.5
longest_match	125	283	1505	0.31	2	5	4.0
compress_block	77	283	21	0.16	207	1592	17.1
send_bits	76	275	23	0.53	2	5	4.0
updcrc	76	18	55	0.00	0	-	-
ct_tally	48	198	175	0.00	0	-	-

Table 3: `gzip` – regularity and performance statistics

tions, rather than the irregular `longest_match` and `compress_block`, as regular functions are generally easier to optimize. `fill_window` has four streams with very high mean stream length (32766) and very low stride (two bytes). From Table 1 we find that the applicable optimizations are prefetching, data and code restructuring, and super-paging. `deflate` has a moderate number of streams (4917) with high mean stream length (3867) and high mean stride (35.5 bytes). Candidate optimizations for `deflate` are loop interchange and copying/remapping.

We base our first optimization on the very high mean stream length, high regularity, and low mean strides in `fill_window`. We recompile the code with compiler flags to enable aggressive prefetching with a higher prefetching distance (five cache lines) with the MIPS Pro C compiler on an R10000 system (the IBM compiler does not support these options). The improvements are modest: There is a 14% reduction in the total cycle count of `fill_window`. `deflate`, with its larger stride and more streams, predictably, benefits much less from this optimization (1.6% reduction in the total cycle count). Nonetheless, the overall performance improvement for the program is around four percent, as shown in Table 4.

Our second optimization, implemented on the IBM SP, also targets `fill_window`. Despite this function's very regular nature (100%), few streams (four) and low stride (2.0), it exhibits many memory stalls. A cursory inspection of the function reveals a sequential update of two 64KB arrays. The code structure precludes conflict misses, and Power3's stream prefetching makes substantial compulsory misses for small-strided data unlikely. With the Power3's small page size (4KB) the two arrays span 32 pages, leading us to conclude that TLB misses and page faults cause a majority of the stalls. By reversing the array update order in every alternate call to `fill_window`, we increase reuse in the TLB (and cache). This code restructuring reduces the TLB misses and memory stalls in `fill_window` by 34% and 7%, as shown in Table 5. The overall memory stalls for the application are reduced by 2.4%. A far more significant optimization would create a super-page for the two arrays, with a provision that the page be locked in memory. The optimization, however, requires special hardware and operating system support and can hence not be implemented on the AIX/Power3 platform.

The gains from our optimizations are very modest, but it is worth noting that the code is highly optimized to begin with; the first optimization requires no study of the code, while the second, only a cursory glance at a function. Similarly, the superpage optimization could reduce page faults and TLB misses and, thus, memory stalls, even further. The reader may rightly question the sensitivity of our regularity data to variance in the input to `gzip`. For this purpose, we took three different data inputs: Two binary files of different sizes and a large text file. We found that the regularity of some functions, such as `deflate` varied significantly with changing input, being highest for the text data file, where the number of repeated strings found by the compression algorithm were the most. More interestingly, `fill_window` exhibited *exactly* the same statistics in each case. A glance at the function clearly shows its data independence. Predictably, the second optimization gave similar speedups in all cases, while the first gave variable results. This highlights that regularity statistics, and hence optimizations based on them, are, in their most general form, a function of the code and the data input. Optimizations applied on invariant statistics should hence be the prime target for the person optimizing the code. Implicit in our statements regarding application (ir)regularity, is its specificity to the input used, though in many cases it is evident that the algorithm is the overriding factor.

function	Total Cycles (millions)		% speedup
	No Prefetch	`-pf_ahead=5`	
deflate	176.7	173.9	1.6
updcrc	150.4	150.4	0.0
fill_window	125.2	107.0	14.0
longest_match	109.6	109.1	0.4
ct_tally	9.5	9.1	4.0
compress_block	5.9	5.9	-0.4
send_bits	5.7	5.7	0.3
Total	**598.0**	**575.7**	**3.7**

Table 4: `gzip` – SpeedShop output with and w/o prefetching for MIPS R10000

function	TLB_MISS (thousands)			MEM_STALL (millions)		
	Normal	Cyclic	% change	Normal	Cyclic	% change
fill_window	7.6	5.0	34.2	345.6	321.2	6.9
deflate	367.0	341.1	7.2	169.2	175.1	-3.4
longest_match	283.3	252.6	10.8	125.4	122.3	2.5
compress_block	283.5	274.3	3.2	77.1	77.0	0.1
send_bits	275.8	261.4	5.2	76.8	76.3	0.7
updcrc	1.8	1.7	4.5	76.2	76.1	0.1
ct_tally	198.9	189.7	4.8	48.7	48.2	1.0
Total	**143.4**	**134.2**	**6.4**	**925.7**	**903.1**	**2.4**

Table 5: `gzip` – PAPI output with and w/o cyclic optimization for Power3

5.2 FT

FT, from the NAS NPB and NPB-2 suite [2], is a 3-D Fast Fourier Transform solving a Poisson PDE problem. We use a serial version of the benchmark. On cache-based and NUMA systems, large strides in 3-D FFTs are known to seriously degrade performance. The common solution is to perform an array transposition by copying [3]. The data in Table 6 is for a fully optimized (-O3) binary.

The extraordinarily high stride (512) in `compute_indexmap` highlights the opportunity for a loop

interchange, array transposition or copying/remapping optimization. Copying incurs a cost in implementing the optimization and remapping requires special hardware/OS support. A brief inspection of the code confirms that a loop interchange will improve cache and TLB usage in `compute_indexmap`. Table 7 presents a summary of the change in performance for binaries compiled with the `-O3` flag. There is a reduction in TLB misses, L1 D-cache, memory stalls, and total cycles in `compute_indexmap` by 99%, 95%, 90% and 72%, respectively. The overall benchmark TLB misses, cache misses, memory stalls and total cycles are reduced by 58%, 4%, 8% and 5%, respectively. The high mean strides in `cffts1`, `cffts2` and `cffts3` suggest that an array transposition (which potentially affects the entire application) may offer better results than the loop interchange we hand-implemented in a single function.

function	STALLS (million)	TLB_MISS (thousand)	L1_MISS (%)	Reg.	Streams	Len.	Stride
`fftz2`	172	54	3.0	0.98	513	16	12.0
`compute_int_`	146	6	0.1	0.00	0	-	-
`evolve`	103	78	12.0	-	-	-	-
`cffts1`	98	35	6.7	1.00	147969	28	39.0
`cffts3`	76	576	3.6	0.91	78307	7	24.0
`cffts2`	60	43	8.4	0.92	523776	7	24.7
`cfftz`	52	9	3.0	0.96	1087	8	39.7
`compute_index`	42	1049	83.9	1.00	4097	140	511.9

Table 6: `FT` – regularity and performance statistics

	TLB_MISS (thousands)			MEM_STALL (millions)			TOT_CYC (millions)		
function	Normal	Opt.	% less	Normal	Opt.	% less	Normal	Opt.	% less
`fftz2`	54	24	55.0	173	169	2.3	459	450	1.9
`compute_int_`	6	7	-16.6	146	147	-0.7	329	329	0.0
`evolve`	78	80	-2.5	104	86	17.3	120	105	12.5
`cffts1`	35	35	0.0	98	98	0.0	119	120	-0.7
`cffts3`	576	573	0.5	76	73	3.9	103	98	4.8
`cffts2`	44	43	2.2	60	59	1.6	83	80	3.6
`cfftz`	10	6	40.0	52	51	1.9	126	125	0.7
`compute_index`	1049	2	99.8	42	4	90.4	44	12	71.7
Total	**1855**	**771**	**58**	**753**	**689**	**8.5**	**1389**	**1320**	**5.0**

Table 7: `FT` – PAPI output with and w/o loop-interchange optimization

6 Related Work

Gerlek et al. determine complex sequences, including periodic and geometric sequences, based on loop induction variables [11]. Their algorithm assists their restructuring Fortran 90 compiler – *Nascent* – in performing a data dependence analysis. Parker et al. use the AST (abstract syntax tree) to detect linearity in the subscript expression of array accesses [23]. Approaches that work at the source code level, typically, have low overhead, but limit themselves to specific language/compilers, and have difficulty contending with pointer-based structures.

Hardware stream detection has been exploited by prefetching and dynamic access optimization mechanisms. Baer and Chen [1] employ a hardware reference prediction table and a look-ahead program counter to preload data with regular access patterns in cache to avoid access penalty. While hardware stream detection is likely to have a lower overhead than our approach, we gather more detailed stream statistics to target a range of profile-driven optimizations.

Chilimbi attempts to address the processor-memory performance gap by introducing the abstraction of *exploitable locality* – a combination of locality and regularity [7]. His definitions of regularity, both spatial and temporal, are sufficiently different from ours to merit special attention. His definition of regularity corresponds to our notion of temporal regularity, while his definition of temporal regularity has no counterpart in our work. Furthermore, his usage of "spatial regularity" differs significantly; ours denotes a property of programs, whereas his quantifies a particular property of streams, as he defines them. Chilimbi focuses on non-scientific codes with references to scalar variables while we address non-scalar references with access patterns of varying regularity. While the specific development of regularity, and the patterns dealt with, differ between his work and ours, the underlying theme is the same: the memory performance of applications can be improved by exploiting the presence of patterns in the memory references of an application.

Marathe *at al.* use dynamic binary rewriting to gather partial traces and perform detailed cache simulations to determine per-reference cache metrics as well as evictor information [18]. They apply code transformations to optimize memory behavior by transforming loops and utilizing tiling. Their work differs in that their cache analysis is much heavier weight. In contrast, we develop regularity metrics. From these metrics, we can infer opportunities for optimizations such as prefetching and loop inversion as well as interchange without detailed cache simulation. Hence, the two approaches are complementary in that we determine opportunities for optimizations at a very low cost in terms of analysis while Marathe *et al.* may find additional opportunities at a considerably higher analysis cost.

7 Conclusion

This paper demonstrates a tool that dynamically detects streamed memory accesses in applications. Despite the intuitive understanding of such accesses, there exist few formalisms to reason about them. To this end, we define a notion of spatial reference regularity that retains previous notions of regularity. This analysis aids a parallel algorithm in dynamically detecting streams within a run-time environment, in contrast to previous efforts that leverage static analysis or hardware detection mechanisms. Unlike static approaches, the run-time mechanism may selectively focus on performance bottlenecks. The hardware mechanisms that would otherwise allow such dynamic flexibility are not available on general-purpose platforms and, hence, are inaccessible to most applications.

This work ameliorates the overhead inherent in an online, software-based detection approach through a framework where sampling is selective and instrumentation is transient. This allows a user to select an appropriate trade-off between measurement overhead and its accuracy and extent. The resultant application of optimizations based on regularity data from the ap-

plication show performance improvements in real applications and popular kernels.

References

[1] BAER, J.-L., AND CHEN, T.-F. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing'91* (Nov. 1991), pp. 176–186.

[2] BAILEY, D., BARSZCZ, E., BARTON, J., BROWNING, D., CARTER, R., DAGUM, D., FATOOHI, R., FREDERICKSON, P., LASINSKI, T., SCHREIBER, R., SIMON, H., VENKATAKRISHNAN, V., AND WEERATUNGA, S. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications 5*, 3 (Fall 1991), 63–73.

[3] BAILEY, D., BISWAS, R., AND DER WIJNGAART, R. V. NAS Applications and Advanced Architectures, Nov. 1997.

[4] BENITEZ, M., AND DAVIDSON, J. Code generation for streaming: An Access/Execute mechanism. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems* (Apr. 1991), pp. 132–141.

[5] CALDER, B., CHANDRA, K., JOHN, S., AND AUSTIN, T. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct. 1998), pp. 139–149.

[6] CARTER, J., HSIEH, W., STOLLER, L., SWANSON, M., ZHANG, L., BRUNVAND, E., DAVIS, A., KUO, C.-C., KURAMKOTE, R., PARKER, M., SCHAELICKE, L., AND TATEYAMA, T. Impulse: Building a smarter memory controller. In *Proceedings of the Fifth Annual Symposium on High Performance Computer Architecture* (Jan. 1999), pp. 70–79.

[7] CHILIMBI, T. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2001), pp. 191–202.

[8] CHILIMBI, T., HILL, M. D., AND LARUS, J. R. Cache-conscious structure layout. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation* (May 1999), pp. 1–12.

[9] DING, C., AND KENNEDY, K. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation* (May 1999), pp. 229–241.

[10] FU, J., AND PATEL, J. Data prefetching in multiprocessor vector cache memories. In *Proceedings of the 18th Annual International Symposium on Computer Architecture* (Toronto, Canada, May 1991), pp. 54–65.

[11] GERLEK, M. P., STOLTZ, E., AND WOLFE, M. Beyond induction variables: detecting and classifying sequences using a demand-driven ssa form. *ACM Transactions on Programming Languages and Systems (TOPLAS) 17*, 1 (1995), 85–122.

[12] HAN, H., RIVERA, G., AND TSENG, C. Software support for improving locality in scientific codes, Jan. 2000.

[13] HAN, H., AND TSENG, C.-W. Improving locality for adaptive irregular scientific codes. Technical Report CS-TR-4039, University of Maryland, College Park, Sept. 1999.

[14] HENNESSY, J., AND PATTERSON, D. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[15] INTERNATIONAL BUSINESS MACHINES INC. *RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide*, 1998.

[16] KENNEDY, K., AND MCKINLEY, K. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Workshop on Languages and Compilers for Parallel Computing* (Aug. 1993), Berlin: Springer Verlag, pp. 301–320.

[17] KODUKULA, I., AHMED, N., AND PINGALI, K. Data-centric multi-level blocking. In *SIGPLAN Conference on Programming Language Design and Implementation* (June 1997), pp. 346–357.

[18] MARATHE, J., AND MUELLER, F. Metric: Tracking down inefficiencies in the memory hierarchy via binary rewriting. In *International Symposium on Code Generation and Optimization* (Mar. 2003). accepted.

[19] MCKEE, S., WULF, W., AYLOR, J., KLENKE, R., SALINAS, M., HONG, S., AND WEIKLE, D. Dynamic access ordering for streamed computations. *IEEE Transactions on Computers 49*, 11 (Nov. 2000), 1255–1271.

[20] MCKINLEY, K., CARR, S., AND TSENG, C.-W. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems 18*, 4 (July 1996), 424–453.

[21] MIPS TECHNOLOGIES, INC. MIPS Pro C and C++ Pragmas. `http://techpubs.sgi.com/` .

[22] MOHAN, T. Detecting and exploiting spatial regularity in data memory references. Master's thesis, University of Utah Department of Computer Science, May 2003.

[23] PARKER, E., DE SUPINSKI, B., AND QUINLAN, D. Measuring the regularity of array references, Oct. 2001.

[24] PORTLAND GROUP, INC. `http://www.pgroup.com/`, 2001.

[25] RIVERA, G., AND TSENG, C.-W. Locality optimizations for multi-level caches. In *Proceedings of Supercomputing '99* (Nov. 1999).

[26] VERA, M. X., LLOSA, J., AND GONZALEZ, A. Near-optimal padding for removing conflict misses, July 2002.

[27] WOLF, M., AND LAM, M. A data locality optimizing algorithm. *ACM SIGPLAN Notices 26*, 6 (June 1991), 30–44.