Dynamic Trace-Based Analysis of Vectorization Potential of Applications

Justin Holewinski Ragavendar Ramamurthi Mahesh Ravishankar Naznin Fauzia
Louis-Noël Pouchet Atanas Rountev P. Sadayappan

Department of Computer Science and Engineering
The Ohio State University
{holewins,ramamurr,ravishan,fauzia,pouchet,rountev,saday}@cse.ohio-state.edu

Abstract

Recent hardware trends with GPUs and the increasing vector lengths of SSE-like ISA extensions for multicore CPUs imply that effective exploitation of SIMD parallelism is critical for achieving high performance on emerging and future architectures. A vast majority of existing applications were developed without any attention by their developers towards effective vectorizability of the codes. While developers of production compilers such as GNU gcc, Intel icc, PGI pgcc, and IBM xlc have invested considerable effort and made significant advances in enhancing automatic vectorization capabilities, these compilers still cannot effectively vectorize many existing scientific and engineering codes. It is therefore of considerable interest to analyze existing applications to assess the inherent latent potential for SIMD parallelism, exploitable through further compiler advances and/or via manual code changes.

In this paper we develop an approach to infer a program's SIMD parallelization potential by analyzing the dynamic data-dependence graph derived from a sequential execution trace. By considering only the observed run-time data dependences for the trace, and by relaxing the execution order of operations to allow any dependence-preserving reordering, we can detect potential SIMD parallelism that may otherwise be missed by more conservative compile-time analyses. We show that for several benchmarks our tool discovers regions of code within computationally-intensive loops that exhibit high potential for SIMD parallelism but are not vectorized by state-of-the-art compilers. We present several case studies of the use of the tool, both in identifying opportunities to enhance the transformation capabilities of vectorizing compilers, as well as in pointing to code regions to manually modify in order to enable auto-vectorization and performance improvement by existing compilers.

Categories and Subject Descriptors C.4 [*Performance of systems*]: Measurement techniques, Performance attributes; D.1.3 [*Programming techniques*]: Concurrent programming—Parallel programming; D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization

General Terms Performance, Measurement, Algorithms

Keywords Performance analysis, dynamic analysis, vectorization

1. Introduction

The SIMD vector units in modern multi-processors achieve very high performance by applying the same instruction to multiple data elements at once. As newer generations of multi-core processors and GPUs continue to extend the width of vector processors, the exploitation of vector instructions is of increasing importance. Unfortunately, many programs are written using structures, pointers, and other non-array constructs that prevent modern compilers from performing the analyses and transformations that are required to fully exploit these vector-processing resources. Even for programs that use arrays, vectorization potential that exists in the computation is often missed by compilers. Some typical reasons for this are (1) conservative dependence analysis, (2) conditional behavior for handling of boundary cases, which precludes the vectorization of the common case, and (3) data layouts that do not allow the contiguous memory accesses needed for efficient vector processing.

Given the sustained trend of increasingly-wide vector units, one key question is whether existing programs can take advantage of these hardware capabilities. The main contribution of our work is *an automatic approach to characterize the inherent vectorizability potential of existing applications* by analyzing information about run-time dependences and memory access patterns. The approach instruments the program to monitor and record instructions and their data accesses, and then analyzes the resulting trace to construct the dynamic dependence graph for the observed execution. Next, the graph is used to partition the dynamic instances of instructions into sets that are both independent and access the memory with a fixed stride. These sets represent instruction instances that can potentially utilize vector resources effectively.

Technical Challenges The identification of potentially vectorizable operations requires the discovery of fine-grained concurrency among operations that access contiguously located data elements. Although there has been considerable prior work (e.g., [2, 3, 8, 11, 12, 14, 16, 17, 19, 21, 23, 25, 28, 29, 33, 35, 39]) on using dynamic analysis for characterizing parallelism in applications, previously developed approaches have fundamental limitations for discovering potentially vectorizable operations. Existing work on using dynamic analysis to characterize potential parallelism in sequential programs falls broadly under two general categories: (1) generation of a parallelism profile and critical-path analysis of the directed acyclic graph (explicitly constructed or implicitly modeled) representing the run-time dependences of the computation, and (2) loop-level or region-level characterization of parallelism, where computations within the loop/region are constrained to execute in the original sequential order. An advantage of the former approach is that the generated parallelism profile implicitly models all possible dependence-preserving reordering of the operations since it performs critical-path analysis of the computation DAG. However, as discussed in the next section, a disadvantage is that independence and potential concurrency at the level of specific statements or expressions in a loop cannot be deduced. The significant advantage of the second approach is that such specific loop/region level concurrency information is extracted. But unlike the former type of analysis, this characterization may be constrained by the order of operations within the modeled region/loop that is imposed by the original program. Thus, the potential for increased parallelism via dependence-preserving reordering of operations is left unexplored. Finally, none of the previous approaches to dynamic analysis for characterizing parallelism consider the patterns of runtime memory accesses, which are critical in the characterization of vectorization potential.

Approach We develop *a new approach to analysis of the dynamic data dependence graph to characterize maximal concurrency per statement/operator under all possible dependence-preserving reorderings of the computation, with further analysis of concurrent operations accessing contiguous or uniformly strided data.* The analysis is useful in a number of ways:

1. *Characterization of code bases:* An automated tool that can be run through large existing code bases to characterize the inherent vectorization potential of those programs can be valuable to multiple groups. First, ISVs (Independent Software Vendors) with large legacy software systems can assess which portions of the code may need complete algorithmic rewrite (if the tool shows no vectorizability) versus code changes without algorithm change (if the tool shows high vectorizability potential). The quantitative information on average vector lengths can be useful in assessing the potential benefit of converting the code to use GPUs (where much higher degree of SIMD parallelism is needed than with short-vector SIMD ISAs). Second, CPU/GPU designers can assess the potential future benefits of widened SIMD structures for important market segments by characterizing the inherent unexploited fine-grained parallelism available in widely-used software in different domains. In order to illustrate this use of the tool, we provide a characterization of the floating-point benchmarks of the SPEC 2006 benchmark suite.
2. *Aid in performance optimization:* Many existing applications contain hot loops with significant vectorization potential. While sometimes simple scanning of time-consuming loops by an application developer may reveal the potential for vectorization, this is non-trivial in many real codes due to multiple levels of function calls that must be analyzed. The automatic identification of code portions that exhibit inherent vectorizability potential can aid an applications expert who can then manually transform the code to enhance its vectorizability by compilers. We illustrate this use of the approach through several case studies.
3. *Aid to compiler writers:* Identification of potential vectorization (for existing programs) that is not exploited by current vectorizing compilers can lead to new insights for compiler writers, and eventually to new static analyses and transformations for state-of-the-art compiler technology. Further development (beyond the scope of this paper) to characterize patterns of statically-analyzable vectorization opportunities (i.e., no data-dependent conditions) missed by a vectorizing compiler can be helpful to compiler writers in enhancing auto-vectorization capabilities. We provide an illustration of this use case through a case study.

2. Background and Overview

The proposed approach is based on the following key observation: to identify and quantify the vectorization potential of a given program, the dynamic analysis needs to uncover *independent operations* that could be executed concurrently. Furthermore, these independent operations should exhibit a pattern of *contiguous access* to memory locations. The rest of this section provides high-level overview and examples to illustrate these two key issues. The specific details of the approach are elaborated later in the paper.

```
for(i = 1; i < N; ++i) {
  A[i] = 2.0 * A[i-1];            // S1
}
for(i = 0; i < N; ++i) {
  for(j = 1; j < N; ++j) {
    B[j][i] = B[j-1][i] * A[i];  // S2
  }
}
```

Listing 1: Example 1 for dynamic parallelism analysis.

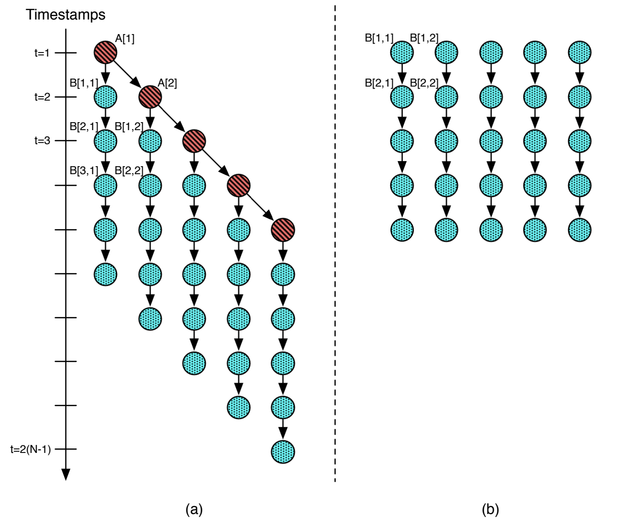

Figure 1: Dependences for Example 1.

2.1 Finding Independent Operations

Prior work on using dynamic analysis to characterize potential parallelism in sequential programs falls broadly under one of two general categories. One approach, exemplified by the early work of Kumar [11], performs timestamp-based analysis of instrumented statement-level execution of the sequential program, using shadow variables to maintain last-modify times for each variable. Each runtime instance of a statement is associated with a timestamp that is one greater than the largest of the last-modify times of all input operands of the statement. A histogram of the number of operations at each time value provides a fine-grained parallelism profile of the computation, and the maximal timestamp represents the critical path length for the entire computation.

In contrast to the above fine-grained approach, an alternate technique by Larus [14] performs analysis of loop-level parallelism at different levels of nested loops. Loop-level parallelism is measured by forcing a sequential order of execution of statements within each iteration of a loop being characterized, so that the only available concurrency is across different iterations of that loop.

To illustrate Kumar's approach, consider the code example in Listing 1. For explanation purposes, this example is extremely simple and is used only to highlight the parallelism characterization

```
for(i = 1; i < N; ++i) {
  A[i] = 2.0 * B[i-1];      // S1
  B[i] = 0.5 * C[i];        // S2
}
```

Listing 2: Example 2 for dynamic parallelism analysis.

from prior work. The run-time instances of the first statement form a chain of dependences of length $N - 1$. The second statement has $N(N - 1)$ run-time instances, with dependences as shown in the figure. The run-time statement instances and their dependences define a *dynamic data-dependence graph* (DDG), as shown in Fig. 1(a). The top row represents instances of statement S1 and the other nodes represent instances of statement S2. For ease of comprehension, a node may be labeled with the array element that is written by that node.

The analysis of potential parallelism computes a timestamp for each DDG node, representing the earliest time this node could be executed; these timestamps are also shown in Fig. 1(a). The largest timestamp, compared to the number of nodes, provides a characterization of the inherent fine-grain parallelism in the program; this largest timestamp gives the length of the *critical path* in the DDG. In essence, the timestamps implicitly model the best parallel execution of all possible dependence-preserving reorderings of the operations performed by the program. In the example from above, the critical path has length $2(N - 1)$, and the overall parallelism is characterized by $(N + 1)/2$, which is the ratio between the number $(N + 1)(N - 1)$ of DDG nodes and the length of the critical path.

All nodes with the same timestamp are independent and can be executed in parallel. However, this method for partitioning of DDG nodes cannot be used to uncover the groups of independent operations needed to characterize the vectorizability of the computation. Consider the example in Listing 1. Statement S2 has large vectorization potential: for a particular fixed value of `j`, all N run-time statement instances for various values of `i` are independent (and, as discussed later, they exhibit a pattern of contiguous access). However, if one were to consider the instances of S2 that are partitioned based on the same timestamp in Fig. 1(a), those partitions uncover *less parallelism* for S2 than what is truly available in the DDG: rather than having $N - 1$ partitions of size N, we have a total of $2(N - 1)$ partitions. Further, the operations in each partition do not access contiguous memory locations, i.e., are not potentially vectorizable.

As described later, we propose a new form of timestamp computation and critical path analysis that focuses on all instances *of a specific statement* (e.g., S2 in this example). This analysis considers whether two instances of the statement of interest are connected by a path in the DDG (with any instances of other statements along the path). If such a path exists, our algorithm guarantees that the timestamp of the first node is smaller than the timestamp of the second node (i.e., the two nodes will be placed in different partitions). Furthermore, each node is guaranteed to have the earliest possible timestamp. For the DDG from Fig. 1(a), our timestamps are shown in Fig. 1(b). Note that all instances of S2 for `j=1` are now given timestamp 1, because they do not depend on any other instances of S2. In general, all instances of S2 for a particular value of `j` have the same timestamp and form a partition. As described later, these partitions (containing independent operations) are then subjected to an analysis for contiguous memory access patterns.

The key problem in this example is that the parallelism analysis interleaves the instances of S1 and S2. An alternative approach could be to separately consider the loops in Listing 1, and perform loop-level parallelism analysis using an approach based on work by Larus [14]. This technique tracks inter-iteration dependences and

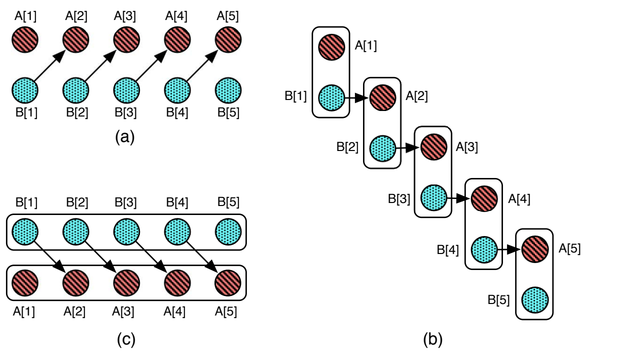

Figure 2: Dependences for Example 2.

computes timestamps for all statement instances in all loop iterations. Inside a loop iteration, the statement instances are executed sequentially. When a statement instance s in loop iteration i depends on a statement instance s' in another iteration, the execution of i stops upon reaching s, until s' is executed. With this approach, the second loop in Listing 1 will be considered independently, and the analysis will uncover that any two iterations of the `i` loop at line 4 are independent. In essence, this will create the parallel partitions shown in Fig. 1(b).

However, this approach for uncovering loop-level parallelism is also inappropriate for our target goal to discover independent operations that can be vectorized. The code in Listing 2 illustrates this point. There is a loop-carried dependence from S2 to S1, as illustrated in Fig. 2(a). As a result, the loop-level parallelism identified by the analysis would be of the form shown in Fig. 2(b). The resulting partitions do not expose the high vectorization potential of S1 (or S2). However, it is easy to see that the computation can be split into two separate loops: first, a loop that iterates over all instances of S2, followed by another loop that iterates over all instances of S1. The loop-level parallelism analysis can easily uncover that each loop (in this hypothetical transformed version) is fully parallel; in fact, each loop is fully vectorizable. However, since the unit of analysis is the original loop code, the potential for parallelism/vectorization is not discovered. Our approach, when analyzing the DDG in Fig. 2(a), will first consider all instances of S1, will discover that they are all independent, and will form a partition containing all of them. Similarly, all instances of S2 will be put in a single partition. The result of our technique is illustrated in Fig. 2(c). Comparing with Fig. 2(b), it is clear that we uncover more parallelism, which in turn leads to finding more potential vectorization.

To summarize, this technique characterizes the parallelism of an entire loop. However, this characterization is constrained by the order of operations within the loop body that is imposed by the sequential execution of the original program. Thus the potential for increased parallelism via dependence-preserving reordering of operations may be missed. Our approach considers all possible dependence-preserving reorderings of all run-time instances of a specific statement of interest, which exposes the necessary parallelism for the purposes of vectorization.

2.2 Finding Operations that Access Contiguously Located Data Elements

Consider again Example 1 from above, and specifically the timestamps shown in Fig. 1(b). Each timestamp defines a partition containing N instances of S2 that are independent of each other. Furthermore, all these instances access contiguous regions of memory.

For a fixed j and varying i, the triples of memory addresses corresponding to the triple of expressions (B[j][i], B[j − 1][i], A[i]) exhibit a pattern of contiguous memory access with the row-major data layout used for arrays in C. This makes the statement instances within each partition viable candidates for vectorization.

We propose *the first analysis* to analyze contiguous memory accesses of independent operations in order to characterize vectorization potential. As described later, the analysis considers all statement instances within a single parallel partition, and defines subpartitions such that within a subpartition, the tuples of memory accesses follow the same pattern of contiguous memory accesses. For example, tuple (B[j][i], B[j − 1][i], A[i]) for a fixed j and varying i will produce tuples of run-time addresses of the form $(c_1 + i \times d, c_2 + i \times d, c_3 + i \times d)$. Here d is the size an array element and $c_{1,2,3}$ are base addresses. These tuples represent accesses to contiguous memory, and together form one single subpartition (which covers the entire parallel partition defined by this particular value of j). One can imagine all statement instances in the subpartition being combined into a single vector operation $[c_1 : c_1+(N-1)\times d] = [c_2 : c_2+(N-1)\times d] \oplus [c_3 : c_3+(N-1)\times d]$ where $\oplus$ represents a vector operation on vectors of size N. Our analysis computes such subpartitions and uses them to characterize the vectorization potential of the analyzed statement.

3. Analyzing Dynamic Data-Dependence Graphs for Vectorization Potential

DDG Generation Generating a dynamic data-dependence graph (DDG) requires an execution trace of the program (or a contiguous subtrace), containing run-time instances of static instructions, including any relevant run-time data such as memory addresses for loads/stores, procedure calls, etc. Our implementation uses LLVM [15] to instrument arbitrary C/C++/Fortran code. The Clang [4] front-end is used to compile C/C++ code into LLVM IR, and the DragonEgg [5] GCC plugin is used to compile Fortran 77/90 code into LLVM IR. The LLVM IR is instrumented to generate a run-time trace to disk, and the instrumented code is compiled to native code.

Once an execution trace is available, the construction of the DDG creates a graph node for each dynamic instruction instance. Edges are created between pairs of dependent nodes (i.e., one instruction instance consumes a value produced by the other). In our implementation each graph node represents a dynamic instance of an LLVM IR instruction, and dependences are tracked through memory and LLVM virtual registers. To construct the graph edges, bookkeeping information for each memory/register remembers the graph node that performed the last write to this location. Note that the graph represents only flow dependences. Anti-dependences and output dependences are not considered, since they do not represent essential features of the computation, and could potentially be eliminated via transformations such as scalar/array expansion. Control dependences are also not considered, since our goal is to focus on the data flow and the optimization potential implied by it. It is straightforward to augment the DDG with additional categories of dependences, without having to modify in any way the subsequent graph analyses (described below).

One interesting case that arises in practice is due to reductions, for example, because of a statement s+=a[i] in an i-loop. The instances of such a statement would form a chain of dependences in the DDG. However, sometimes it is possible to vectorize reductions (e.g., by updating a vector sv instead of a scalar s). Our analysis currently does not consider the potential for such vectorization. A possible enhancement is to identify and remove dependence edges that are due to updates of reduction variables; detection of such

Algorithm 1: Timestamp computation.

```
   Input: id: static instruction ID, graph: DDG
1  foreach Node node in TopologicalOrder(graph) do
2      TS ← 0;
3      foreach pred in Predecessors(graph, node) do
4          TS ← Max(TS, GetTimeStamp(pred));
5      end
6      if GetInstructionID(node) == id then
7          TS ← TS + 1;
8      end
9      AssignTimeStamp(node, TS);
10 end
```

dependences has already been used by prior work [22] in a different context.

Candidate Instructions The execution trace may contain many instructions that should not be analyzed for SIMD parallelism, such as integer operations for loop book-keeping. Hence, the analysis is restricted to instructions that involve floating-point addition, subtraction, multiplication, and division. These instructions correspond to the set of floating-point instructions that have vector counterparts in SIMD architectures. They are also of particular importance for optimization of certain computationally-intensive applications (as exemplified by the SPEC floating-point benchmarks). Of course, all other instructions that participate in dependences are taken into account by the analysis, but their potential SIMD parallelism is not characterized.

3.1 Generation of Parallel Partitions

To benefit from SIMD parallelism, an instruction must exhibit fine-grained parallelism. For a static instruction s that is being characterized, the potential parallelism of $\{s_1, s_2, \ldots\}$ (where s_k is the k-th run-time instance of s) can be uncovered by observing the data dependences from any s_i to any s_j for $j > i$. This is equivalent to identifying whether the DDG contains a path from the node for s_i to the node for s_j. Such a path exists if and only if the data produced by s_i is directly or indirectly used by s_j (i.e., s_i and s_j cannot be executed concurrently).

Each candidate static instruction s (as described earlier) is analyzed independently, using Algorithm 1. A unique ID for s is assigned at instrumentation time. A topological sort traversal of the DDG is performed and a timestamp is assigned to each node. For each visited node, the largest predecessor timestamp is determined. If the node is an instance of the static instruction s being analyzed, the timestamp is incremented by one; otherwise, it is not altered.

The generated timestamps are then used to construct partitions within the graph, by putting all instances of s with the same timestamp into the same partition. An example illustrating this approach was shown earlier in Section 2. Consider the DDG shown in Fig. 1(a). The nodes at the top of the DDG represent the run-time instances of S1, while the rest of the DDG nodes are instances of S2. Suppose we wanted to evaluate the potential vectorizability of S2. For this static instruction, the analysis will compute timestamps for S2 instances as shown in Fig. 1(b). Each timestamp value defines one partition of S2 instances.

PROPERTY 3.1. *Consider any DDG node s_i and a DDG path p ending at s_i. Let $s(p)$ be the number of nodes on p (excluding s_i) that are instances of the static instruction s being analyzed. The timestamp computed for s_i by Algorithm 1 is the largest value of $s(p)$ for all p leading to s_i.*

The proof is by induction on the length of p. This property has two implications. First, consider two instances s_i and s_j of s. If there exists a run-time dependence from s_i to s_j (either directly, or indirectly through instances of instructions other than s, or through other instances of s itself), then the timestamp of s_i is strictly smaller than the timestamp of s_j. Thus, all instances of s with the same timestamp (i.e., in the same partition) are independent of each other. Second, each s_i is assigned the smallest possible timestamp—that is, it is scheduled at the earliest possible time. Considering the average partition size as a metric of available parallelism for the instances of s, the following property can be proven easily.

PROPERTY 3.2. *Algorithm 1 finds the maximum available parallelism for each static instruction s.*

3.2 Partitioning for Contiguous Access

Algorithm 1 ensures that the instances of s within a partition are independent, but efficient SIMD parallelism also requires contiguous memory access. On most SIMD architectures, the cost of loading vector elements individually and shuffling them into a vector register offsets the benefit of exploiting the vector hardware.

Thus, the partitions must be further subdivided into units that exhibit parallelism *and* contiguous memory access. In general, we want to ensure unit-stride accesses—the distance in memory between consecutive memory accesses is equal to the size of the data type. We also allow zero-stride (i.e., the distance in memory between consecutive accesses is zero), since vector splats (copying a scalar value into all elements of a vector) are cheap for most SIMD architectures. Note that the zero-stride case also covers operations with constant operands.

To ensure unit-stride, the instruction instances within a parallel partition are sorted according to the memory addresses of their operands. For constants or values produced by other instructions but not saved to memory, an artificial address of zero is used. The sorted list is then scanned, ending the current subpartition (and starting a new one) when the stride is (1) non-zero and non-unit, or (2) different from the previously observed stride. The result is a (now potentially larger) set of subpartitions such that the dynamic instruction instances within a subpartition are independent, and exhibit uniform zero-stride or unit-stride accesses to memory. The average size of these subpartitions is a metric of the vectorization potential of the static instruction being characterized.

3.3 Non-Unit Constant Stride Access

The contiguous access check performed in the previous stage is important for discovering efficient vectorization potential, but a variation of the check can be used to explore the potential benefit of data layout transformations on the original code. It is not uncommon to find computations where fine-grained concurrency exists but the data accessed has a non-unit but constant stride. In the first loop in Listing 3 there is a loop-carried dependence along the inner j loop but the i loop is parallel. If the loops were permuted, we would have fine-grained parallelism in the inner loop for the instances of S1, but the access stride would be N. A data layout transformation to transpose the array (i.e., swap the first and second array dimensions) would enable unit-stride access and efficient vectorization. Listing 4 shows how the code could be transformed.

The second loop in Listing 3 illustrates a scenario with arrays of structures that results in fine-grained concurrent operations but non-unit access stride. The instances of S2 are all independent but they exhibit stride-2 access (e.g., 8 bytes if `x` and `y` are single-precision floating-point). The same is true for the instances of S3. In this case, changing the data structure from an array of structures to a structure of arrays would enable parallel, stride-1 access which could be automatically vectorized.

By relaxing the unit/zero-stride condition to instead check for any non-unit constant stride, we can detect cases such as the ones

```
for( i = 0 ; i < N ; i++ )
  for( j = 2 ; j < N ; j++ )
    A[i][j] = 2*A[i][j-1] - A[i][j-2]; // S1

for ( i = 0 ; i < N ; i++ ) {
  C[i].x = B[i].x + B[i].y;   // S2
  C[i].y = B[i].x - B[i].y;   // S3
}
```

Listing 3: Vectorization benefits of data layout transformations: stride-N column access and array-of-structures access.

```
// transposed declarations for A, B, and C
for( j = 2 ; j < N ; j++ )
  for( i = 0 ; i < N ; i++ )
    A[j][i] = 2*A[j-1][i] - A[j-2][i]; // S1

for ( i = 0 ; i < N ; i++ ) {
  C.x[i] = B.x[i] + B.y[i];   // S2
  C.y[i] = B.x[i] - B.y[i];   // S3
}
```

Listing 4: Loop transformations and data layout transformations applied to Listing 3.

illustrated in Listing 3. Given the partitions produced by Algorithm 1, we apply the unit-stride analysis from the previous subsection. At the end, any instruction instance that belongs to a subpartition of size one is identified. All such instances (of the same static instruction, and with the same timestamp) are then sorted and scanned. When the currently observed stride does not match the previously observed one, the instruction is put on a waitlist for future processing, and the scanning based on the current stride continues until the end of the list is reached; this results in one subpartition. Any waitlisted instructions are then traversed again, in sorted order, so that the next subpartition can be formed.

4. Evaluation

In this section we present a number of studies to illustrate the use of the dynamic analysis tool. Although the studies are restricted to analysis of sequential programs, the tool can also be used with parallel programs using Pthreads, OpenMP, MPI, etc.—the instrumentation and trace generation would be applied to one or more sequential processes or threads of the parallel program to assess the potential for SIMD vector parallelism within a process/thread. Further, although we only concentrate on characterizing floating-point operations (because they tend to be the focus of most SIMD optimization efforts), such analysis can be carried out for any type of operations, e.g., integer arithmetic.

The experiments were performed on a machine with an Intel Xeon E5630 processor and 12GB memory, running Linux 2.6.32. To obtain performance measurements, the Intel icc compiler (12.1.3) was used to compile the program code, at the `O3` optimization level. Profiling data was obtained with HPCToolkit [10] version 5.2.1, at sampling period 500 thousand cycles. The instrumentation infrastructure was implemented in LLVM 3.0.

4.1 Characterization of Applications

We illustrate the use of the tool for characterizing software collections by applying it to the SPEC CFP2006 floating-point benchmarks, and kernels from the UTDSP benchmark suite [32]. We also include two stand-alone compute kernels: a 2-D Gauss-Seidel stencil code and a kernel from a 2-D PDE grid-based solver; they are elaborated upon later as case studies illustrating potential use of the analysis for performance optimization and compiler enhancement.

					Unit Stride		Non-unit Stride	
Benchmark	**Loop**	**Percent Cycles**	**Percent Packed**	**Average Concur.**	**Percent Vec. Ops**	**Average Vec. Size**	**Percent Vec. Ops**	**Average Vec. Size**
410.bwaves	`block_solver.f : 55`	79.2%	53.0%	39.9	97.5%	11.1	0.0%	–
	`block_solver.f : 176`	65.8%	66.4%	8.3	100.0%	5.0	0.0%	–
433.milc	`gauge_stuff.c : 258`	22.0%	0.0%	10453.4	36.2%	10427.4	49.7%	3.3
	`path_product.c : 49`	17.9%	0.0%	73316.6	36.4%	69441.5	63.6%	3.2
	`quark_stuff.c : 566`	15.2%	0.0%	23687.7	88.3%	11.4	7.5%	4.2
	`quark_stuff.c : 960`	44.9%	0.0%	11447.3	65.1%	15.5	18.7%	2.3
	`quark_stuff.c : 973`	35.0%	0.0%	61566.7	57.4%	13.8	32.9%	2.4
	`quark_stuff.c : 1452`	14.2%	0.0%	20736.0	36.4%	20736.0	63.6%	502.3
	`quark_stuff.c : 1460`	13.6%	0.0%	20736.0	36.4%	20736.0	63.6%	20736.0
	`quark_stuff.c : 1523`	15.4%	0.0%	2921.1	55.0%	2000.0	45.0%	4.2
434.zeusmp	`advx3.f : 637`	11.3%	35.0%	66613.9	74.3%	442.1	16.6%	16.0
435.gromacs	`innerf.f : 3960`	60.4%	4.4%	4.0	60.3%	12.0	21.5%	2.0
	`ns.c : 1264`	16.2%	3.8%	4.9	60.0%	42.0	20.9%	2.1
	`ns.c : 1461`	35.3%	3.3%	40.3	64.1%	31.0	35.1%	2.0
	`ns.c : 1503`	13.3%	0.0%	5.0	62.5%	5.0	30.0%	2.0
436.cactusADM	`StaggeredLeapfrog2.F : 342`	18.4%	100.0%	80.0	100.0%	80.0	0.0%	–
	`StaggeredLeapfrog2.F : 366`	81.1%	96.9%	78.0	100.0%	78.0	0.0%	–
437.leslie3d	`tml.f : 522`	15.6%	98.5%	8805.5	100.0%	158.3	0.0%	–
	`tml.f : 889`	13.4%	99.2%	7434.2	99.9%	178.4	0.0%	–
	`tml.f : 1269`	12.4%	99.2%	438.3	100.0%	22.0	0.0%	–
	`tml.f : 3569`	21.6%	98.6%	8100.0	100.0%	90.0	0.0%	–
444.namd	`ComputeList.C : 71`	33.2%	0.0%	130.2	86.0%	101.1	13.7%	11.4
	`ComputeList.C : 75`	66.4%	0.0%	313.3	93.3%	295.4	6.6%	7.8
	`ComputeNonbondedBase.h : 321`	12.9%	0.0%	15.6	85.9%	262.8	7.8%	5.4
447.dealII	`mapping_q1.cc : 514`	10.4%	0.0%	1.0	0.0%	–	0.0%	–
	`step-14.cc : 715`	16.2%	0.0%	130.9	75.6%	58.2	24.3%	24.9
	`step-14.cc : 780`	10.9%	0.0%	27.0	66.7%	27.0	33.3%	27.0
	`step-14.cc : 3198`	19.1%	3.1%	335.6	87.5%	12.5	12.5%	18.8
450.soplex	`ssvector.cc : 983`	10.6%	0.0%	373.0	32.2%	18.5	56.2%	18.2
	`slufactor.cc : 839`	12.5%	0.0%	59.7	43.2%	6.5	39.4%	2.0
	`spxsolve.cc : 126`	37.7%	0.0%	384.3	92.3%	25.6	3.5%	2.1
	`spxsolve.cc : 200`	58.5%	0.0%	361.6	88.2%	15.9	5.0%	2.1
	`svector.h : 293`	12.8%	0.0%	1.7	0.0%	–	40.0%	2.0
453.povray	`bbox.cpp : 894`	53.3%	0.2%	11.2	62.6%	14.8	27.3%	2.7
	`csg.cpp : 248`	58.6%	0.0%	4.3	35.5%	4.8	41.1%	2.0
	`csg.cpp : 254`	16.2%	0.2%	1.0	0.0%	–	0.0%	–
	`lbuffer.cpp : 1373`	23.0%	0.1%	14.8	63.8%	16.4	29.8%	2.9
	`lighting.cpp : 600`	66.9%	1.0%	13.1	65.4%	13.9	28.1%	2.0
	`lighting.cpp : 938`	27.1%	0.1%	13.7	63.3%	15.7	29.7%	2.8
	`lighting.cpp : 2298`	31.5%	1.0%	7.7	59.3%	16.2	26.5%	2.9
	`lighting.cpp : 4120`	41.7%	0.8%	11.4	64.8%	13.1	25.0%	2.1
454.calculix	`Chv_update.c : 736`	13.6%	91.5%	27.4	48.4%	15.0	51.6%	11.4
	`e_c3d.f : 675`	69.7%	0.1%	35.6	100.0%	12.3	0.0%	–
	`FrontMtx_update.c : 207`	16.4%	91.3%	774.0	96.4%	28.6	3.1%	9.4
	`FrontMtx_update.c : 38`	14.0%	91.2%	1116.3	96.7%	12.9	2.6%	4.7
	`mafillsm.f : 144`	74.7%	0.4%	6064.8	99.2%	136.9	0.8%	3.1
	`Utilities_DV.c : 1241`	11.4%	96.6%	2.0	50.0%	49.0	0.0%	–
459.GemsFDTD	`NFT.F90 : 1068`	17.4%	0.0%	24.2	69.9%	9.9	19.3%	2.1
	`update.F90 : 108`	17.3%	97.4%	201.0	100.0%	201.0	0.0%	–
	`update.F90 : 242`	17.1%	97.3%	200.0	100.0%	200.0	0.0%	–
465.tonto	`mol.F90 : 5565`	15.7%	80.4%	50779.4	99.2%	150.7	0.3%	2.4
	`mol.F90 : 11659`	59.0%	19.5%	266.6	97.2%	31.6	1.0%	4.4
470.lbm	`lbm.c : 186`	99.6%	100.0%	137487.0	61.6%	137487.0	38.4%	72.1
481.wrf	`solve_em.F90 : 179`	87.9%	79.1%	1198.6	97.4%	39.7	1.2%	15.1
	`solve_em.F90 : 884`	14.4%	89.3%	54721.8	99.8%	117.0	0.2%	29.1
	`solve_em.F90 : 1258`	14.8%	89.6%	9887.1	93.6%	89.1	6.4%	28.5
	`solve_em.F90 : 1538`	12.8%	87.4%	95531.4	95.4%	27.6	4.6%	7.6
482.sphinx3	`approx_cont_mgau.c : 279`	39.8%	68.1%	8949.0	75.2%	6886.1	24.8%	2.3
	`cont_mgau.c : 652`	17.0%	72.8%	3.7	75.0%	39.0	0.0%	–
	`subvq.c : 456`	30.8%	75.0%	19154.8	75.5%	15360.0	24.5%	2048.0
	`vector.c : 521`	25.9%	86.1%	3.3	75.0%	13.0	0.0%	–

Table 1: Analysis results for analyzed benchmark loops.

The stand-alone kernels and UTDSP benchmarks were directly analyzed by the tool and the results are shown in Tables 2 and 3. For full application codes, such as the SPEC CFP2006 floating-point benchmarks, the output produced by the dynamic analysis will be very extensive. Therefore, we only analyze and report characteristics for time-consuming loops identified via profiling by HPC-

Benchmark	Percent Packed	Average Concur.	Unit Stride Percent Vec. Ops	Unit Stride Average Vec. Size	Non-unit Stride Percent Vec. Ops	Non-unit Stride Average Vec. Size
2-D Gauss-Seidel Stencil	0.0%	226	22.2%	46.1	77.4%	9.3
2-D PDE Grid Solver	0.0%	231426	100.0%	820.8	0.0%	—

Table 2: Analysis results for computation kernels.

Benchmark	Type	Percent Packed	Average Concur.	Unit Stride Percent Vec. Ops	Unit Stride Average Vec. Size	Non-unit Stride Percent Vec. Ops	Non-unit Stride Average Vec. Size
FFT	Array	49.9%	568.9	79.3%	24.1	12.2%	2.0
	Pointer	0.0%	568.9	79.3%	24.1	12.2%	2.0
FIR	Array	99.8%	99.9	100.0%	57.4	0.0%	–
	Pointer	0.0%	99.9	100.0%	57.4	0.0%	–
IIR	Array	0.00%	43.6	64.8%	14.3	15.6%	8.9
	Pointer	0.00%	43.6	64.8%	14.3	15.6%	8.9
LATNRM	Array	7.8%	7.4	74.6%	23.9	0.0%	–
	Pointer	8.2%	7.4	74.6%	23.9	0.0%	–
LMSFIR	Array	0.0%	2.7	48.3%	22.1	16.5%	21.8
	Pointer	0.0%	2.8	49.4%	28.0	16.2%	21.9
MULT	Array	50.4%	181.9	100.0%	18.2	0.0%	–
	Pointer	0.00%	181.9	100.0%	18.2	0.0%	–

Table 3: Analysis results for UTDSP benchmark suite.

Toolkit, analyzing only loops that account for at least 10% of total execution cycles during a run of the benchmark using the SPEC reference data set (the 10% threshold was selected to reduce the amount of data presented; we also collected data at a threshold of 5%). To perform the DDG analysis for a particular loop, we collected several subtraces corresponding to separate instances of the loop (using the SPEC train data set, and for a few large loops the test data set). A subtrace was started upon loop entry and terminated upon loop exit and its DDG was constructed and analyzed. We randomly chose several instances of the loop, analyzed each corresponding subtrace to obtain the various metrics described later, and chose one representative subtrace to be included in the measurements presented in the paper.

The results of the analysis for SPEC CFP2006 are shown in Table 1. For each benchmark, we only show loops that account for at least 10% of total execution cycles. We start with all innermost loops, and only include a parent loop if the total percentage of execution cycles spent in it is at least 10 percentage points greater than the sum of the percentages for its inner loops. The `gamess` benchmark could not be compiled with LLVM and was not used in the experiments.

The *Percent Packed* metric shows the percentage of floating-point run-time operations that were executed using packed (i.e., vector) SSE instructions, as reported by HPCToolkit. This column provides information on the effectiveness of current compilers (we used Intel icc since we have found it to be superior to other production compilers in vectorization capability) in vectorizing each of the identified hot loops. A high value indicates that a significant fraction of the floating-point operations in that loop were executed via packed SIMD instructions. A zero value indicates that the compiler was unable to achieve any vectorization at all for the loop.

The *Average Concurrency* metric was computed by determining the average partition size across the collection of partitions for all floating-point instructions in the graph, where partitions were formed by considering only instruction independence (Sect. 3.1). For this metric, both singleton and non-singleton partitions were considered. From the non-singleton parallel partitions, subpartitions containing unit-stride operations were formed (as described in Sect. 3.2). The *Percent Vec. Ops* metric (i.e., potentially vectorizable run-time instructions) shows the number of operations that belong to non-singleton unit-stride subpartitions, as percent of the total number of operations in the graph. The *Average Vec. Size* metric represents the average size of these non-singleton vectorizable subpartitions. The general trend is that for most of the loops, many run-time instructions belong to partitions that exhibit both independence and contiguous memory access. Furthermore, the sizes of these partitions are large—in many cases, much larger than the vector sizes in existing and emerging architectures. Thus, the analysis indicates that the majority of the analyzed loops may have high vectorization potential.

The run-time instructions within a non-singleton parallel partition that did not belong in any unit-stride subpartition were further analyzed with the non-unit stride analysis described in Section 3.3. The analysis reported the number of such instructions that could be placed in subpartitions accessing data at some fixed non-unit stride. This number, as percent of the total number of all run-time instructions in the graph, is shown in column *Percent Vec. Ops*. The average size of such subpartitions is given in the last column. There are several examples where a significant number of independent instructions can be combined together using non-unit stride, and the sizes of the partitions are large as well. This indicates that data layout transformations may be beneficial in these cases.

There may be cases where the percentage of packed instructions observed via profiling exceeds the sum of the values in columns *Percent Vec. Ops* (e.g., `Utilities_DV.c:1241` in 454.calculix and `vector.c:521` in 482.sphinx3). This happens in the presence of a reduction (e.g., `s+=expr`): our analysis considers the chain of dependences and treats the computation as non-vectorizable. However, there exist approaches to vectorize reductions, and icc employs some of them. In future work, our approach could be extended to ignore dependences due to reductions, which would uncover these additional vectorization opportunities.

As is typical of other work on dynamic analysis of fine-grained dependences (e.g., [14, 36]), the instrumentation incurs an overhead of two to three orders of magnitude, relative to the execution time of the original unmodified code. The cost of DDG analysis depends on graph size and memory access patterns, and is typically of the order of tens to hundreds of microseconds per DDG node. Although we have not focused on tool optimization, the utility of the current unoptimized implementation of our prototype is not hampered for two reasons. First, the analysis is intended to be performed offline, e.g., during performance tuning. Many profiling analyses have been successfully used in this setting, and various existing techniques can be readily applied to reduce their cost (e.g., [36, 38]). Second, the instrumented code can be run with much smaller problem sizes than the "production" size: in our experience, although metrics such as average vector size can vary with problem size, the qualitative insights about potential vectorizability do not change.

4.2 Assisting Vectorization Experts

Many institutions possess large code bases that were largely developed before the recent emergence of SIMD parallelism in all CPUs/GPUs. When the original developers of the code are not available to adapt it for improved vectorization, an automated tool can be very valuable. For some loops, a quick scan of the code of a hot loop by a vectorization expert will immediately reveal the opportunities for enhancing vectorizability through code changes. But this is certainly not the norm, especially with C++ codes or C codes that make heavy use of pointers. An automated tool allows the vectorization expert to quickly eliminate loops with little to no vectorization potential, and concentrate on the loops with high potential. With these, some of the code structures involve multiple levels of function calls and the output from the tool is valuable input to the expert, indicating that the effort to unravel the code is likely worth it. As an example, the hot loops in `444.namd` are generated using C preprocessor macros and it is very difficult to get an understanding of the code just by scanning it. If we examine the HPCToolkit profile data, we know the loops are hot, but not whether or not we have any hope of vectorizing them. However, our analysis shows that there is a high potential for vectorization in this part of code, so it may be worth the time investment of a vectorization expert to carefully analyze these loops.

Another use case is for identifying missed opportunities in compiler test suites. Vendor compilers are typically tested against large amounts of code to gauge the performance of the compiler's vectorizer. It is easy to automate this testing to see how much of the code is vectorized, but for the remaining code, it is not clear whether the code is just not vectorizable, or if the compiler is missing an opportunity. It would take considerable effort for a vectorization expert to manually analyze all of the non-vectorized code. The analysis tool can help to automate the process and focus the expert's effort on identifying why code that has been identified as being potentially vectorizable is not actually being vectorized by the compiler.

4.3 Array-Based vs. Pointer-Based Code

Auto-vectorizing compilers are becoming increasingly good at vectorizing array-based code, but pointer-based code is often not vectorized due to the added complexities with pointer aliasing and the verification of contiguous access during the compiler's static analysis. A primary benefit of the proposed dynamic analysis technique is the ability to analyze pointer based code just as easily as array based code. Both versions of the same computation will provide the same analysis results, since the dynamic analysis considers IR-level arithmetic operations, and does not make a distinction between data that is read from arrays or pointer dereferencing.

To test this facet of our analysis, we used the UTDSP [32] benchmark suite, which contains both array- and pointer-based versions of several computation kernels for digital signal processors (DSP). The suite was created to evaluate the quality of code generated by a high-level language compiler (e.g., a C compiler) targeting a programmable DSP. Thus, each kernel was written in different styles, including an array-based version and a pointer-based version. Both versions provide identical functionality, except for the use of arrays or pointers to traverse the data structures.

		Original	**Trans.**	**Speedup**
2-D Seidel	Xeon E5630	0.228s	0.074s	3.086×
	Core i7 2600K	0.170s	0.083s	2.055×
	PhenomII 1100T	0.200s	0.068s	2.926×
2-D PDE	Xeon E5630	0.355s	0.185s	1.909×
	Core i7 2600K	0.198s	0.116s	1.699×
	PhenomII 1100T	0.722s	0.401s	1.799×
410.bwaves	Xeon E5630	2.42e-1s	1.88e-2s	1.284×
	Core i7 2600K	1.72e-1s	1.24e-1s	1.382×
	PhenomII 1100T	2.74e-1s	2.50e-1s	1.100×
433.milc	Xeon E5630	2.13e-2s	1.24e-2s	1.725×
	Core i7 2600K	9.10e-3s	7.30e-3s	1.245×
	Phenom II 1100T	3.05e-2s	1.99e-2	1.528×
435.gromacs	Xeon E5630	238.5s	149.4s	1.596×
	Core i7 2600K	163.6s	98.1s	1.667×
	Phenom II 1100T	210.3s	151.8s	1.386×

Table 4: Performance measurements for the case studies.

Table 3 shows the results of this experiment. The measurements include the percentage of vectorizable operations found in the program, the average vector size, and the percentage of operations that are actually vectorized by the Intel icc compiler. We see that our analysis is invariant to the form of the code[1], but icc fails to vectorize some of the pointer-based code. Such knowledge would be very useful in optimizing certain applications, where a conversion from pointer-based code to array-based code may be worthwhile if the potential benefits are high. The dynamic analysis from our tool could be a valuable first step in the process.

4.4 Case Studies

Based on the results from the previous subsections, some benchmarks were manually transformed to enable vectorization by icc. The targeted benchmarks were the Gauss-Seidel stencil, the PDE grid solver, and the 410.bwaves, 433.milc, and 435.gromacs benchmarks from SPEC CFP2006. A comparison of the performance of the original and modified versions is shown in Table 4. In addition to the Intel Xeon E5630 machine used for the measurements presented earlier, two other machines were used in these experiments: an Intel Core i7 2600K and an AMD PhenomII 1100T, both with the same icc configuration. For each benchmark, we show the total execution time for both versions, as well as the achieved speedup. When the target of the optimization is a particular loop (e.g., bwaves and gromacs), the measurements in the table are based on the total time spent in the loop. The reference data sets were used when running the SPEC benchmarks.

Gauss-Seidel In this case study we analyze the vectorization potential of a 9-point Gauss-Seidel stencil code. This code has been identified by our analysis as being not auto-vectorized by the vendor compiler, but possessing non-trivial vectorization potential (see Table 2). Listing 5 shows the original kernel. It has a loop-carried dependence in the innermost `j` loop, since the fourth operand `A[i][j-1]` is produced in the previous iteration of this loop. Similarly, the outer `i` loop also has loop-carried dependences. Due to the dependences, icc was unable to vectorize the code. This was not unexpected. However, what surprised us was that the dynamic analysis revealed vectorization potential for this code.

The analysis classified two out of the eight addition operations (`A[i-1][j-1]+A[i-1][j]+A[i-1][j+1]`) as vectorizable. This is because the operands were all produced in the previous iteration of the `i` loop. The only true dependence in the loop is

[1] The discrepancy in LMSFIR is due to a difference in the way the two versions are written, resulting in slightly different distributions of operations.

```
/* Original */
cnst = 1/9.0;
for(t=0; t<T ; t++)
  for(i=1; i<N-1; i++)
    for(j=1; j<N-1; j++)
      A[i][j] = (A[i-1][j-1] + A[i-1][j] +
                 A[i-1][j+1] + A[i][j-1] +
                 A[i][j]     + A[i][j+1] +
                 A[i+1][j-1] + A[i+1][j] +
                 A[i+1][j+1]) * cnst;
/* Transformed */
cnst = 1/9.0;
for(t=0; t<T; t++)
  for(i=1; i<N-1; i++) {
    for(j=1; j<N-1; j++)
      temp[j] = A[i-1][j-1] + A[i-1][j]   +
                A[i-1][j+1] + A[i][j]     +
                A[i][j+1]   + A[i+1][j-1] +
                A[i+1][j]   + A[i+1][j+1];

    for(j=1;j<N-1;j++)
       A[i][j] = cnst * (A[i][j-1] + temp[j]);
  }
```

Listing 5: Original and transformed Gauss-Seidel code.

due to `A[i][j-1]`. The operations involving elements from row `i+1`, and even the addition of `A[i][j]` and `A[i][j+1]`, could be performed in vectorized mode by splitting the loop into a sequence of two loops, as shown in Listing 5. The first `j` loop in the transformed code is now completely vectorized by icc, resulting in significant performance improvement (see the results in Table 4, obtained for $N = 1000$ and $T = 20$).

The extent of vectorizability of the Gauss-Seidel code was a surprise to us since our initial expectation was that the code would not exhibit vectorization potential due to loop-carried dependences. Closer examination of the dependences shows that all the information needed to transform the code is actually derivable from purely static analysis. However, to our knowledge, no research or production compiler can perform the transformation we performed manually. This example illustrates how the developed dynamic analysis can be valuable for compiler writers to identify scenarios where enhancements to static analysis and transformation capabilities can enable improved code transformations for vectorization.

2-D PDE Solver In this case study we analyze the core computation from a 2-D PDE grid-based solver. The code is from the examples included with PETSc [20] 3.1-p7 and solves the solid fuel ignition problem, modeled as a partial differential equation. The original source can be found under */src/snes/examples/tutorials/ex5.c* in the PETSc distribution. We see that this code is not auto-vectorized by icc, but our analysis shows very high vectorization potential. In this kernel, the 2-D computation grid is distributed onto a 2-D grid of blocks, where the computation is performed by iterating over every cell within every block. For our purposes, we consider only sequential execution of the program.

The per-block kernel code is shown in Listing 6. The `if` condition in the innermost loop is a boundary condition check that forces grid points on the boundary to follow a different path of execution as compared to the other interior points. The loop bounds for this loop nest, along with two of the four conditions that can trigger the `if` statement, are data dependent. As a result, compilers are forced to be conservative and assume that for each iteration of the loop, it is unknown whether the `then` or `else` clause will be executed. Due to this constraint, the vectorizability of this particular loop nest, as written, is very low. Further, without more constraints on the values within the `info` structure, static analysis cannot determine transformations that would enable vectorization.

```
 1 /* Original */
 2 for (j=info->ys; j<info->ys+info->ym; j++) {
 3   for (i=info->xs; i<info->xs+info->xm; i++) {
 4     if (i == 0 || j == 0 ||
 5         i == info->mx-1 || j == info->my-1) {
 6       f[j][i] = x[j][i];
 7     } else {
 8       u = x[j][i];
 9       uxx = (2.0*u-x[j][i-1]-x[j][i+1])*hydhx;
10       uyy = (2.0*u-x[j-1][i]-x[j+1][i])*hxdhy;
11       f[j][i] = uxx+uyy-sc*PetscExpScalar(u);
12     }
13   }
14 }
15 /* Transformed */
16 if(info->ys==0 || info->xs==0 ||
17    (info->ys+info->ym)==my ||
18    (info->xs+info->xm)==mx) {
19   /* Same as lines 2-14 */
20 }
21 else {
22   for(j=info->ys; j<info->ys+info->ym; j++) {
23     for(i=info->xs; i<info->xs+info->xm; i++) {
24       u = x[j][i];
25       uxx = (2.0*u-x[j][i-1]-x[j][i+1])*hydhx;
26       uyy = (2.0*u-x[j-1][i]-x[j+1][i])*hxdhy;
27       f[j][i] = uxx+uyy-sc*PetscExpScalar(u);
28     }
29   }
30 }
```

Listing 6: Original and transformed 2-D PDE Solver.

However, the results of the dynamic analysis show a great potential for vectorizability within this code (see Table 2). Specifically, the `else` clause exhibits perfect vectorizability. To allow a compiler to vectorize this loop, we can rewrite the code to extract the `if/then/else` construct and then hoist an `if` to provide a vectorizable loop. The modified code is shown in Listing 6. The key to the vectorization-enabling transformation is the observation that cells which correspond to the boundary condition can only occur on boundary blocks. We observe that `i` and `j` cannot be zero except within blocks along the top or left edge of the grid, and cannot be equal to the maximum index value (`mx-1` and `my-1`) except within blocks along the right or bottom edge of the grid. Therefore, the kernel code can be split into two separate versions; one for boundary blocks and one for interior blocks, as shown in Listing 6. While this code should provide no speed-up for boundary blocks, it enables vectorization for interior blocks and provides an advantage when the blocks are in at least a 3×3 grid.

Table 4 shows the total execution time for the original and modified code for a case where the block size is 512×512 and blocks are in a grid of size 16×16. As shown in the table, the performance improvement is substantial.

410.bwaves In the 410.bwaves benchmark, one of the loops we analyzed in our extended study (at 5% threshold for hot loops) is at `jacobi_lam.f`:30. This loop exhibits a very low percentage of packed instructions, while the dynamic analysis shows that there are significant numbers of vectorizable operations at unit and non-unit stride. The result from the non-unit stride analysis suggests possible improvements through data layout transformation.

Listing 7 is representative of the computation within this loop. Arrays `je` and `jv` are of size `(5,5,nx,ny,nz)`, and array `q` is of size `(5,nx,ny,nz)` where `nx`, `ny` and `nz` are program constants. While there is no dependence between the iterations of the innermost loop `i`, there are two factors that hinder vectorization. First, there is no unit-stride data access pattern since `i` is used to access the third dimension of the arrays. Second, the use of `mod`

```
 1   ish = 1
 2   ksh = 1
 3   jsh = 1
 4   !! Original
 5   do k = 1,nz
 6      kp1 = mod(k,nz+1-ksh)+ksh
 7      do j=1,ny
 8         jp1=mod(j,ny+1-jsh)+jsh
 9         do i=1,nx
10            ip1=mod(i,nx+1-ish)+ish
11            !! Some computation
12            je(1,1,i,j,k) = ...
13            je(1,2,i,j,k) = ...
14            ...
15            je(4,5,i,j,k) = ...
16            je(5,5,i,j,k) = ...
17            !! Some computation
18            ros = q(1,ip1,jp1,kp1)
19            !! Similar computation for jv as for je
20         enddo
21      enddo
22   enddo
23   !! Transformed
24   do k = 1,nz
25      kp1 = mod(k,nz+1-ksh)+ksh
26      do j=1,ny
27         jp1=mod(j,ny+1-jsh)+jsh
28         do i=1,nx-1
29            ip1=i+1
30            !! Some computation
31            je(i,1,1,j,k) = ...
32            je(i,1,2,j,k) = ...
33            ...
34            !! Some computation
35            ros = q(ip1,1,jp1,kp1)
36            !! Similar computation for jv as for je
37         enddo
38         i = nx
39         ip1 = 1
40         !! Inner loop computation (lines 11-19)
41      enddo
42   enddo
```

Listing 7: Original and transformed bwaves code.

operations to calculate the neighbor with wrap-around boundary conditions hampers the vectorization of accesses to array `q`.

To address these problems, a data layout transformation was performed on arrays `je`, `jv`, and `q`: the dimension which was originally accessed by `i` (and `ip1`) was moved to become the fastest varying dimension of the arrays. This transformation is shown in Listing 7. The modified code has a stride-1 data access pattern. The `mod` operations were removed by peeling the last iteration of the `i` loop, and introducing `ip1=i+1` within the loop. The value of `ip1` for the peeled iteration was set to 1. Table 4 shows the performance of the original and modified versions.

433.milc In this case study we explore the benefits of data layout transformations on the 433.milc benchmark. Specifically, we focus on one of the loops from Table 1. The loop starting at `quark_stuff.c`:1452 shows no automatic vectorization by the compiler. There is also limited potential for vectorization at unit stride. However, the non-unit stride analysis shows significant potential for vectorization, implying that a data layout transformation may speed up the computation.

This loop iterates over every point in a lattice, applying a matrix-vector multiplication operation at each point. The matrices are of size 3×3 and the vectors are of size 3, both containing complex numbers. The lattice itself holds a matrix at each point, and external arrays of vectors are used for the vector inputs and outputs. The matrix-vector multiplication is not vectorized by the compiler due to non-unit stride access (distribution of real/imaginary components) and a small inner loop trip count (3).

```
/* Original data layout */
typedef struct { double r, i; } complex;
typedef struct { complex c[3]; } su3_vector;
typedef struct { complex e[3][3]; } su3_matrix;
su3_matrix lattice[NUM_SITES];
su3_vector vec[NUM_SITES], out_vec[NUM_SITES];
/* Original computation */
for(s = 0; s < NUM_SITES; ++s) {
  for(i = 0; i < 3; ++i) {
    complex x = { 0.0, 0.0 };
    for(j = 0; j < 3; ++j) {
      complex y;
      y.r = lattice[s].e[i][j].r * vec[s].c[j].r -
            lattice[s].e[i][j].i * vec[s].c[j].i;
      y.i = lattice[s].e[i][j].r * vec[s].c[j].i +
            lattice[s].e[i][j].i * vec[s].c[j].r;
      x.r += y.r; x.i += y.i;
    }
    out_vec[s].c[i] = x;
  }
}
/* Transformed data layout */
typedef struct {
  double r[3][3][NUM_SITES];
  double i[3][3][NUM_SITES];
} lattice_dlt;
typedef struct {
  double r[3][NUM_SITES];
  double i[3][NUM_SITES];
} vec_dlt;
lattice_dlt lattice;
vec_dlt vec, out_vec;
/* Transformed computation */
/* Initialize the elements of out_vec to 0.0 */
for(i = 0; i < 3; ++i) {
  for(j = 0; j < 3; ++j) {
    for(s = 0; s < NUM_SITES; ++s) {
      double x_r, x_i;
      x_r = lattice.r[i][j][s] * vec.r[j][s] -
            lattice.i[i][j][s] * vec.i[j][s];
      x_i = lattice.r[i][j][s] * vec.i[j][s] +
            lattice.i[i][j][s] * vec.r[j][s];
      out_vec.r[i][s] += x_r;
      out_vec.i[i][s] += x_i;
    }
  }
}
```

Listing 8: Original and transformed milc code.

To help us isolate the required changes, we created a version of the benchmark that only contains the computation we identified. The full benchmark was too large to optimize manually without in-depth understanding of the entire application, and the smaller, kernel-ized version allowed us to show proof-of-concept benefits of a data layout transformation. To optimize this operation, the transformation was applied to the lattice data structure, where the lattice of matrices was converted to a matrix of lattices. This modification exposes unit-stride operations within the inner loop. The original and transformed code are shown in Listing 8. Table 4 demonstrates a significant speedup for the modified kernel.

435.gromacs For this case study we focus on the loop at line 3960 in `innerf.f` from 435.gromacs. While icc is not able to vectorize this loop in any significant manner, the dynamic analysis results indicate the existence of unit-stride operations. Lines 1–14 of Listing 9 represent the computation within the loop.

The value of `j3` is data dependent (based on the run-time values in indirection array `jjnr`) and is used to index into arrays `pos` and `faction`. The compiler has to assume that the loop is not parallel, in case two or more elements of `jjnr` have the same value and thus create a dependence through `faction(j3)`. Furthermore, the access patterns for `pos` and `faction` are not regular due to the arbitrary values of `j3`. Thus, the loop is not vectorized by icc.

```
 1 !! Original
 2 do k = nj0,nj1
 3    jnr = jjnr(k)+1
 4    j3  = 3*jnr-2
 5    jx1 = pos(j3)
 6    ...
 7    tx11 = ...
 8    fjx1 = faction(j3) - tx11
 9    tx21 = ...
10    fjx1 = fjx1 - tx21
11    tx31 = ...
12    faction(j3) = fjx1 - tx31
13    ...
14 enddo
15 !! Transformed
16 do k = nj0,nj1,4
17    do k_vect = 1,4
18       jnr = jjnr(k+k_vect-1)+1
19       vect_j3(k_vect) = 3*jnr-2
20       vect_jx1(k_vect) = pos(vect_j3(k_vect))
21       ...
22       vect_fjx1(k_vect) = faction(vect_j3(k_vect))
23       ...
24    enddo
25    do k_vect = 1,4
26       tx11 = ...
27       vect_fjx1(k_vect) = vect_fjx1(k_vect) - tx11
28       tx21 = ...
29       vect_fjx1(k_vect) = vect_fjx1(k_vect) - tx21
30       tx31 = ...
31       vect_fjx1(k_vect) = vect_fjx1(k_vect) - tx31
32       ...
33    enddo
34    do k_vect = 1,4
35       faction(vect_j3(k_vect)) = vect_fjx1(k_vect)
36       ...
37    enddo
38 enddo
```

Listing 9: Original and transformed gromacs code.

When the dynamic analysis results were examined at the level of individual statements, it became clear that the loop is in fact parallel: the values in `jjnr` ensure that distinct elements of `faction` are accessed by each iteration. (The relatively low average concurrency in Table 1 is due to the small number of loop iterations and to a few chains of reductions.) Furthermore, although the accesses to `pos` and `faction` do not exhibit any patterns, the rest of the computation in the loop body is done through scalars and can be easily vectorized.

For better vectorization, the loop was strip-mined as shown in Lines 15–38 of Listing 9. (The cleanup loop is not shown.) Loop distribution of the inner `k_vect` loop was then applied to move all reads from `pos` and `faction` above the computation, and to move all writes to `faction` after the computation. Array expansion of temporary `j3` was also performed to hold the necessary indices of `faction`. The middle `k_vect` loop is now vectorized by icc. Table 4 shows the reduction in the total time spent in the loop due to the transformation.

Unlike in the earlier case studies, here the analysis results do not necessarily generalize to arbitrary input data. Specifically, the fact that all iterations of the loop were independent affects both the vectorization potential and the code modifications. As written, the transformed code in Listing 9 is not correct for all possible inputs. It becomes necessary to assert this correctness with the help of additional information—e.g., an expert's knowledge of certain properties of the problem domain, or some compiler analysis of the intra- and inter-procedural code context surrounding the loop.

Limitations Although these case studies demonstrate the usefulness of the analysis reports, the proposed technique has a number of limitations. For example, for the loop from 435.gromacs, the conclusions about vectorizability are dependent on properties of the input data. As another example, we investigated loop `bbox.cpp`:894 from 453.povray in greater depth. This benchmark is a ray-tracer, and the loop in question implements a work-list algorithm that intersects a ray with a tree of bounding boxes. The computation is driven by a priority queue. Each iteration of the loop removes a bounding box from the queue and, if necessary, adds other bounding boxes to the queue. Inside an iteration, what processing is performed on the current bounding box depends on whether it is an inner node or a leaf of the tree, and whether the ray intersects the boxes of its children. The overall structure of the computation is very irregular and heavily depends on the actual run-time data. As part of the intersection tests for boxes and the scene objects contained in them, some low-level operations (e.g., computing the angle between two vectors) occur repeatedly, with high concurrency and with some potential vectorizability for certain subcomputations. However, the highly-irregular structure of the control flow makes it extremely challenging to exploit the vectorization potential without significant changes to the code by a domain expert with a deep understanding of the algorithm.

An interesting direction for future work is to refine the dynamic analysis to distinguish computations with irregular data-dependent control flow from ones where the control flow is more structured and vectorization potential is more likely to be actually realizable through code transformations.

5. Related Work

A number of techniques have been proposed to measure the available parallelism at the statement or instruction level (e.g., [2, 8, 9, 11, 12, 16, 17, 21, 23–25, 28, 33]). Several approaches aim to characterize instruction-level parallelism (ILP) and how it is affected by hardware features and compiler optimizations (e.g., [12, 33]). Austin and Sohi [2] construct a dynamic dependence graph and use it to measure ILP for several configurations. Typically, in these and similar studies, a program execution trace is first created; next, possible parallel schedules are defined by taking into account the dependences between binary instructions in the trace, under various assumptions (e.g., anti and output dependences may be ignored). A notable exception is the work by Kumar [11], which does not create a trace and instead instruments the program to compute the parallel schedule online. Some generalizations of this approach have been proposed in recent work [8].

Researchers have also considered dynamic analysis of loop-level parallelism, where all iterations of a loop may run concurrently with each other. The approach by Larus [14] generates an execution trace and analyzes it to model the effects of loop-level parallelism. A related technique is applied in the context of speculative parallelization of loops, where shadow locations are used to track dynamic dependences across loop iterations [22]. Several other approaches of similar nature have been investigated in more recent work (e.g., [3, 19, 30, 31, 35, 39]).

The efficient collection of dynamic data-dependence information has also been explored by previous work. Tallam et al. use run-time control flow information to reconstruct significant portions of the dynamic data-dependence graph of the program run [26], and have proposed a technique for producing lightweight tracing of multi-threaded code [27]. Zhang et al. use an approach to collect compressed profiles from program executions, including data dependence information [37]; they also propose techniques to decrease the cost of maintaining a dynamic dependence graph in the context of dynamic slicing [36, 38].

Automatic parallelization is also closely related to the characterization of vectorization in programs. Techniques to automatically exploit fine-grained parallelism must first determine the dependences between instructions. There is a body of work on the characterization of dynamic data dependences in the context of speculative execution (e.g., [22, 30, 40]).

Automatic vectorization has been the subject of extensive study (e.g., [1, 6, 7, 13, 13, 18, 34]). These approaches use static dependence analysis and then convert scalar instructions to vector instructions, in cases when the conditions for efficient vectorization are met. To the best of our knowledge, no prior work has addressed the topic of this paper—the development of an approach for dynamic analysis of vectorizability of computations.

6. Conclusions

This paper presents a new dynamic analysis for the characterization of SIMD parallelism potential in programs. Existing methods for characterizing concurrency have fundamental limitations in discovering potentially vectorizable operations, but these problems are overcome by the newly developed approach. The use of the analysis is illustrated by characterizing several computationally-intensive loops in benchmarks. The results demonstrate that the approach can detect potential opportunities for vectorization that are missed by a state-of-the-art vectorizing compiler. In addition to its use in characterizing large software suites for vectorization potential, the proposed technique can assist vectorization experts in identifying potentially profitable code regions on which attention should be focused, as well as aid compiler experts by identifying potentially vectorizable code that the compiler's vectorizer misses.

Acknowledgments

We thank the PLDI reviewers for their valuable comments. This material is based upon work supported by the National Science Foundation under grants CCF-0811781, CCF-0926127, OCI-0904549, CCF-1017204, and by the Department of Energy's Office of Advanced Scientific Computing under grant DE-SC0005033.

References

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

[2] T. Austin and G. Sohi. Dynamic dependency analysis of ordinary programs. In *ISCA*, pages 342–351, 1992.

[3] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *MICRO*, pages 69–84, 2007.

[4] Clang. clang.llvm.org.

[5] DragonEgg. dragonegg.llvm.org.

[6] A. Eichenberger, P. Wu, and K. O'Brien. Vectorization for SIMD architectures with alignment constraints. In *PLDI*, pages 82–93, 2004.

[7] L. Fireman, E. Petrank, and A. Zaks. New algorithms for SIMD alignment. In *CC*, pages 1–15, 2007.

[8] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor. Kremlin: Rethinking and rebooting gprof for the multicore age. In *PLDI*, pages 458–469, 2011.

[9] C. Hammacher, K. Streit, S. Hack, and A. Zeller. Profiling Java programs for parallelism. In *IWMSE*, pages 49–55, 2009.

[10] HPCToolkit. www.hpctoolkit.org.

[11] M. Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE TC*, 37(9):1088–1098, 1988.

[12] M. Lam and R. Wilson. Limits of control flow on parallelism. In *ISCA*, pages 46–57, 1992.

[13] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI*, pages 145–156, 2000.

[14] J. Larus. Loop-level parallelism in numeric and symbolic programs. *IEEE TPDS*, 4(1):812–826, 1993.

[15] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, page 75, 2004.

[16] J. Mak and A. Mycroft. Limits of parallelism using dynamic dependency graphs. In *WODA*, pages 42–48, 2009.

[17] A. Nicolau and J. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE TC*, 33(11):968–976, 1984.

[18] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. In *PLDI*, pages 132–143, 2006.

[19] C. Oancea and A. Mycroft. Set-congruence dynamic analysis for thread-level speculation (TLS). In *LCPC*, pages 156–171, 2008.

[20] PETSc. www.mcs.anl.gov/petsc.

[21] M. Postiff, D. Greene, G. Tyson, and T. Mudge. The limits of instruction level parallelism in SPEC95 applications. *SIGARCH Computer Architecture News*, 27(1):31–34, 1999.

[22] L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *PLDI*, pages 218–232, 1995.

[23] L. Rauchwerger, P. Dubey, and R. Nair. Measuring limits of parallelism and characterizing its vulnerability to resource constraints. In *MICRO*, pages 105–117, 1993.

[24] A. Rountev, K. Van Valkenburgh, D. Yan, and P. Sadayappan. Understanding parallelism-inhibiting dependences in sequential Java programs. In *ICSM*, page 9, 2010.

[25] D. Stefanović and M. Martonosi. Limits and graph structure of available instruction-level parallelism. In *Euro-Par*, pages 1018–1022, 2000.

[26] S. Tallam and R. Gupta. Unified control flow and data dependence traces. *ACM TACO*, 4(3):19, 2007.

[27] S. Tallam, C. Tian, R. Gupta, and X. Zhang. Enabling tracing of long-running multithreaded programs via dynamic execution reduction. In *ISSTA*, pages 207–218, 2007.

[28] K. Theobald, G. Gao, and L. Hendren. On the limits of program parallelism and its smoothability. In *MICRO*, pages 10–19, 1992.

[29] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *MICRO*, pages 330–341, 2008.

[30] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Speculative parallelization of sequential loops on multicores. *JPP*, 37(5):508–535, 2009.

[31] G. Tournavitis, Z. Wang, Zheng, B. Franke, and M. O'Boyle. Towards a holistic approach to auto-parallelization. In *PLDI*, pages 177–187, 2009.

[32] UTDSP Benchmarks. www.eecg.toronto.edu/~corinna.

[33] D. Wall. Limits of instruction-level parallelism. In *ASPLOS*, pages 176–188, 1991.

[34] M. Wolfe. *High Performance Compilers For Parallel Computing*. Addison-Wesley, 1996.

[35] P. Wu, A. Kejariwal, and C. Caşcaval. Compiler-driven dependence profiling to guide program parallelization. In *LCPC*, pages 232–248, 2008.

[36] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *PLDI*, pages 94–106, 2004.

[37] X. Zhang and R. Gupta. Whole execution traces and their applications. *ACM TACO*, 2(3):301–334, 2005.

[38] X. Zhang, R. Gupta, and Y. Zhang. Cost and precision tradeoffs of dynamic data slicing algorithms. *ACM TOPLAS*, 27(4):631–661, 2005.

[39] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *HPCA*, pages 290–301, 2008.

[40] X. Zhuang, A. E. Eichenberger, Y. Luo, K. O'Brien, and K. O'Brien. Exploiting parallelism with dependence-aware scheduling. In *PACT*, pages 193–202, 2009.