SD^3: A Scalable Approach to Dynamic Data-Dependence Profiling

Minjang Kim, Nagesh B. Lakshminarayana, Hyesoon Kim, Chi-Keung Luk‡
College of Computing, Georgia Institute of Technology, Atlanta, GA
‡Intel Corporation, Hudson, MA

Abstract—As multicore processors are deployed in mainstream computing, the need for software tools to help parallelize programs is increasing dramatically. *Data-dependence profiling* is an important technique to exploit parallelism in programs. More specifically, manual or automatic parallelization can use the outcomes of data-dependence profiling to guide *where* to parallelize in a program.
However, state-of-the-art data-dependence profiling techniques are *not scalable* as they suffer from two major issues when profiling large and long-running applications: (1) *runtime overhead* and (2) *memory overhead*. Existing data-dependence profilers are either unable to profile large-scale applications or only report very limited information.
In this paper, we propose a *scalable* approach to data-dependence profiling that addresses both runtime and memory overhead in a single framework. Our technique, called *SD^3*, reduces the runtime overhead by *parallelizing* the dependence profiling step itself. To reduce the memory overhead, we compress memory accesses that exhibit *stride patterns* and compute data dependences directly in a compressed format. We demonstrate that SD^3 reduces the runtime overhead when profiling SPEC 2006 by a factor of 4.1× and 9.7× on eight cores and 32 cores, respectively. For the memory overhead, we successfully profile SPEC 2006 with the reference input, while the previous approaches fail even with the train input. In some cases, we observe more than a 20× improvement in memory consumption and a 16× speedup in profiling time when 32 cores are used.

Index Terms—profiling, data dependence, parallel programming, program analysis, compression, parallelization.

1 Introduction

As multicore processors are now ubiquitous in mainstream computing, parallelization has become the most important approach to improving application performance. However, specialized software support for parallel programming is still immature although a vast amount of work has been done in supporting parallel programming. For example, automatic parallelization has been researched for decades, but it was successful in only limited domains. Hence, unfortunately, parallelization is still a burden for programmers.

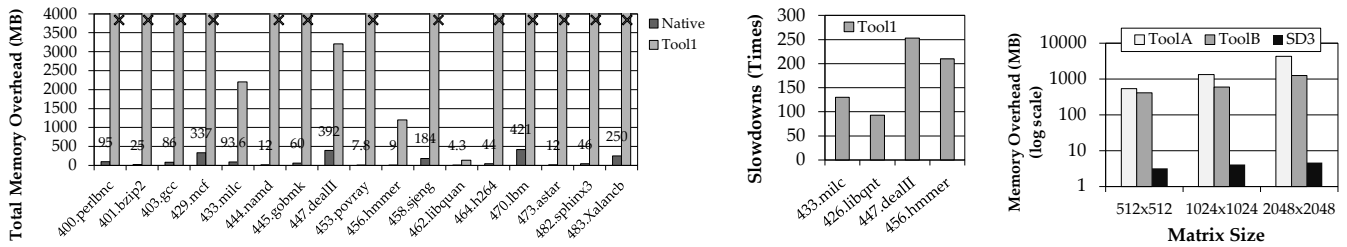
Recently several tools including Intel Parallel Studio [12], CriticalBlue Prism [6], and Vector Fabric vfAnalyst [31] have been introduced to help the parallelization of legacy serial programs. These tools provide useful information on parallelization by analyzing serial code. A key component of such tools is *dynamic data-dependence analysis*, which indicates if two tasks access the same memory location and at least one of them is a write operation. Two data-independent tasks can be safely executed in parallel without the need for synchronization.

Traditionally, data-dependence analysis has been done *statically* by compilers. Techniques such as the GCD test [25] and Banerjee's inequality test [16] were proposed in the past to analyze data dependences in array-based data accesses. However, this static analysis may not be effective in languages that allow pointers and dynamic allocation. For example, we observed that state-of-the-art production automatic parallelizing compilers often failed to parallelize simple embarrassingly parallel loops written in C/C++. The compilers also had limited success in irregular data structures due to pointers, indirect memory accesses, and control flows.

Rather than entirely relying on static analysis, dynamic analysis using *data-dependence profiling* is an alternative or a complementary approach to address the limitations of the static approach since all memory addresses are resolved in runtime. Data-dependence profiling has already been used in parallelization efforts like speculative multithreading [4, 8, 21, 28, 34] and finding potential parallelism [18, 26, 30, 32, 36]. It is also being employed in the commercial tools we mentioned. However, the current algorithm for data-dependence profiling incurs significant costs of time and memory overhead. Surprisingly, although there have been a number of research works on using data-dependence profiling, almost no work exists on addressing the performance and scalability issues in data-dependence profilers.

As a concrete example, Fig. 1 demonstrates the scalability issues of our baseline algorithm. We believe that this baseline algorithm is still state-of-the-art and very similar to the current commercialized tools. Figures 1(a) and 1(b) show the memory and time overhead, respectively, when profiling 17 SPEC 2006 C/C++ applications with the train input on a 12 GB machine. Among the 17 benchmarks, only four benchmarks were successfully analyzed, and the rest of the benchmarks failed because of insufficient physical memory (consumed more than 10 GB memory). The runtime overhead is between 80× and 270× for the four benchmarks that worked. While

• *minjang@gatech.edu, nageshbl,@cc.gatech.edu, hyesoon@cc.gatech.edu, ‡chi-keung.luk@intel.com*



(a) Memory overhead of Tool1 (✕: Dependence profiling takes 10+ GB memory.) (b) Time overhead of Tool1 (c) Memory overhead of matrix addtion

Fig. 1: Overhead of our baseline algorithm and current tools.

both time and memory overhead are severe, the latter will stop further analysis. The culprit is the dynamic data-dependence profiling algorithm used in these tools, which we call the *pairwise method*. The algorithms of previous work [4, 18] are similar to the pairwise method. This pairwise method needs to store all outstanding memory references in order to check dependences, resulting in huge memory bloats.

Another example that clearly shows the memory scalability problem is a simple matrix addition program that allocates three N × N matrices ($A = B + C$). As shown in Fig. 1(c), the current tools require significant additional memory as the matrix size increases while our method, SD^3, needs only very small (less than 10 MB) memory.

In this paper, we address these memory and time scalability problems by proposing a scalable data-dependence profiling algorithm called SD^3. Our algorithm has two components. First, we propose a new data-dependence profiling technique using a compressed data format to reduce the memory overhead. Second, we propose the use of parallelization to accelerate the data-dependence profiling process. More precisely, this work makes the following contributions to the topic of data-dependence profiling:

1) *Reducing memory overhead by stride detection and compression along with new data-dependence calculation algorithm:* We demonstrate that SD^3 significantly reduces the memory consumption of data-dependence profiling. However, SD^3 is not a simple compression technique; we should address several issues to achieve the profiling to be memory efficient. The failed benchmarks in Fig. 1(a) are successfully profiled by SD^3.
2) *Reducing runtime overhead with parallelization:* We show that our memory-efficient data-dependence profiling itself can be effectively parallelized. We observe an average speedup of 4.1× on profiling SPEC 2006 using eight cores. For certain applications, the speedup can be as high as 16× with 32 cores.

2 Background

2.1 Usage Models of Data-Dependence Profiler.

Before describing SD^3, we illustrate usage models of our dynamic data-dependence profiler, as shown in Fig. 2.

A tool using the data-dependence profiler takes a program in either source code or binary and profiles with a representative input. A raw result from our dependence profiler is list of

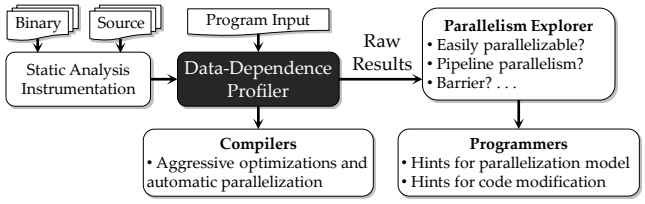


Fig. 2: Examples of data-dependence profiler applications

discovered data-dependence pairs, as shown in Table 2. All or some of the following information is provided by our profiler:

- **Sources** and **sinks** of data dependences (in source code lines if possible, otherwise in program counters)[1],
- **Types** of data dependences: Flow (Read-After-Write, RAW), Anti (WAR), and Output (WAW) dependences,
- **Frequencies** and **distances** of data dependences,
- Whether a dependence is **loop-carried** or **loop-independent**, and data dependences carried by a particular loop in **nested loops**,
- **Data-dependence graphs** in functions and loops.

A raw result can be further analyzed to give programmers advice on parallelization models and transformation of the serial code. We discuss some details in Section 9. The raw results also can be used by aggressive compiler optimizations and opportunistic automatic parallelization [30]. Among the three steps, obviously the data-dependence profiler is the bottleneck of the scalability problem, and we focus on this in the paper.

2.2 Why the Dynamic Approach?

One of the biggest motivations for using the data-dependence profiling is to complement the weaknesses of the static data-dependence analysis. Table 1 shows an example of the weaknesses of the static analysis. We tried automatic parallelization on the OmpSCR benchmarks [1] (OpenMP Source Code Repository), which consist of short mathematical kernel code, by Intel C/C++ compiler (Version 11.0) [11]. The compiler options are calibrated to maximize the opportunities to find automatic parallelism.

Programmer# column of Table 1 shows the number of loops that the programmer manually parallelized. Note that the benchmarks of OmpSCR are parallelized by OpenMP, and most of the parallelized loops are embarrassingly parallel.

1. Data dependences from registers can be analyzed at static time.

TABLE 1: The number of parallelizable loops based on a compiler and a data-dependence profiler (static vs. dynamic) in OmpSCR

Benchmark	Programmer #	Compiler #	Reason	Profiler #
FFT	2	1	Recursion	3
FFT6	3	0	Pointers	16
Jacobi	2	1	Pointers	8
LUreduction	1	0	Pointers	4
Mandelbrot	1	0	Reduction	2
Md	2	1	Reduction	10
Pi	1	1	N/A	1
QuickSort	1	0	Recursion	4

```
1: // A, B are dynamically allocated integer arrays
2: for (int i = 1; i <= 2; ++i) {     // For-i
3:   for (int j = 1; j <= 2; ++j) {   // For-j
4:     A[i][j] = A[ i ][j-1] + 1;
5:     B[i][j] = B[i+1][ j ] + 1;
6:   }
7: }
```

Fig. 3: A simple example of data-dependence profiling.

Compiler# column represents the number of loops that were automatically parallelized by the compiler. *Reason* column summarizes why the compiler misidentified actually parallelizable loops as non-parallelizable based on the diagnostic information of the compiler. The result shows that only four loops (out of 13) were successfully parallelized. Pointer aliasing was the main cause of the limitations. Finally, *Profiler#* column shows the number of loops that are identified as parallelizable through our data-dependence profiler. Our profiler identified as parallelizable *all* loops that were parallelized by the programmer. However, note that the numbers of *Profiler#* are greater than those of *Programmer#* because our profiler reported literally parallelizable loops without considering cost and benefit.

Despite the effectiveness of the data-dependence profiling, we should note that any dynamic approach has a fundamental limitation, *the input sensitivity problem*: a profiling result *only* reflects given inputs. A purely dynamic dependence profiler can only report *potential* parallelism. However, as Section 7.4 will show, data dependences in frequently executed loops are not changed noticeably with respect to different inputs. Furthermore, our primary usage model of the data-dependence profiler is assisting programmers in manual parallelization, where compilers cannot automatically prove the parallelizability of the code. In this case, programmers should empirically verify the correctness of the parallelized code with several tests. We believe a dependence profiler should be very informative to programmers in manual parallelization.

3 The Baseline Pairwise Method

We describe our baseline algorithm, *the pairwise method*, which is still the state-of-the-art algorithm for existing tools. SD^3 is implemented on top of the pairwise method. At the end of this section, we summarize the problems of the pairwise method. We begin our descriptions of the algorithm by focusing on data dependences within *loop nests*, because loops are major parallelization targets. Note that our algorithm can be easily extended to find data dependences in arbitrary program structures, for example, dependences among function calls, which will be discussed in Section 9.

TABLE 2: lMemory traces and discovered dependences of Fig. 3

	j = 1	j = 2
i = 1	**A[1][1]** = A[1][0] + 1; B[1][1] = **B[2][1]** + 1;	A[1][2] = **A[1][1]** + 1; B[1][2] = **B[2][2]** + 1;
i = 2	**A[2][1]** = A[2][0] + 1; **B[2][1]** = B[3][1] + 1;	A[2][2] = **A[2][1]** + 1; **B[2][2]** = B[3][2] + 1;

(a) Memory traces (Boldfaces are conflicting accesses)

Loop	Var	Source/Sink (R/W, Line#, Col#)	Dependence (Type, Freq)
For-i	B[]	(R,5,15)→(W,5, 5)	Loop-carried WAR, 2
For-j	A[]	(W,4, 5)→(R,4,15)	Loop-carried RAW, 2

(b) Discovered dependences (Inductions `i` and `j` are ignored.)

3.1 Checking Data Dependences in a Loop Nest

In the big picture, to calculate data dependences in a loop, we find conflicts between the memory references of the current loop iteration and the previous iterations.

Our pairwise method temporarily buffers all memory references during the *current* iteration of a loop. We call these references *pending references*. When an iteration ends, we compute data dependences by checking pending references against the *history references*, which are the memory references that appeared from the beginning to the previous loop iteration. These two types of references are stored in the *pending table* and *history table*, respectively. Each loop has its own pending and history table, instead of having the tables globally. This is needed to compute data dependences correctly and efficiently while considering (1) nested loops and (2) loop-carried/independent dependences.

We explain the pairwise algorithm with a simple loop nest example in Fig. 3 and Fig. 4. Note that we intentionally use a simple example. This code may be easily analyzed by compilers, but the analysis could be difficult if (1) dynamically allocated arrays are passed through deep and complex procedure call chains, (2) the bounds of loops are unknown at static time, or (3) control flow inside of a loop is complex. We detail how the pairwise algorithm works with Fig. 4:

- ❶: During the first iteration of `For-j` (`i = 1`, `j = 1`), four pending memory references are stored in the pending table of `For-j`. After finishing the current iteration, we check the pending table against the history table. At the first iteration, the history table is empty. Before proceeding to the next iteration, the pending references are *propagated* to the history. This propagation is done by *merging* the pending table with the history table. (This merge operation will be complex in our SD^3.)
- ❷: After the second iteration (`i = 1`, `j = 2`), we now see a loop-carried RAW on `A[1][1]` in `For-j`. Meanwhile, `For-j` terminates its first invocation.
- ❸: At the same time, observe that the first iteration of the outer loop, `For-i`, is also finished. In order to handle data dependences across loop, we treat an inner loop as if it were completely *unrolled* to its upper loop. This is done by propagating the history table of `For-j` to the pending table of `For-i`. Hence, the entire history of `For-j` is now at the pending table of `For-i`.
- ❹ and ❺: `For-j` executes its second invocation.
- ❻: Similar to ❸, the history of the second invocation of `For-j` is again propagated to `For-i`. Data dependences

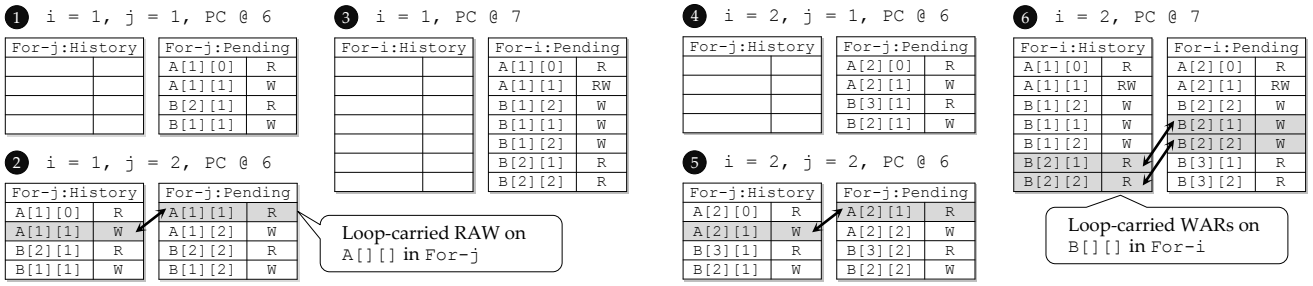


Fig. 4: Snapshots of the pending and history tables of each iteration when the pairwise method profiles Fig. 3. Each snapshot was taken at the end of an iteration. 'PC@6' (PC is at line 6) and 'PC@7' mean the end of an iteration of For-j and For-i, respectively. Note that the tables actually have absolute addresses, not a symbolic format like A[1][1]. The table entry only shows R/W. However, it also remembers the occurrence count and the timestamp (trip count) of the memory address to calculate frequencies and distances of dependences, respectively.

```
1: void scan_recognize(...) {
2:  for (j = starty; j < endy; j += stride) {
3:    for (i = startx; i < endx; i += stride){
4:    ...
5:    pass_flag = 0;            11: void match() {
6:    match();                  12:   ...
7:    if (pass_flag == 1)       13:   if (condition)
8:      do_something();         14:     pass_flag = 1;
9:    ...                       15:   ...
                                16: }
```

Fig. 5: Loop-independent dependence on `pass_flag` in 179.art.

are checked, and we discover two loop-carried WARs on `B[][]` with respect to `For-i`.

In this example programmers can parallelize the outer loop after removing the WARs by duplicating `B[]`. The inner loop is not lucrative for parallelization due to the short-distant loop-carried RAWs on `A[]`.

3.2 Handling Loop-independent Dependences

When reporting data dependences inside a loop, we must distinguish whether a dependence is *loop-independent* (i.e., dependences within the same iteration) or *loop-carried* because its implication is very different on judging parallelizability of a loop: Loop-independent dependences do not prevent parallelizing a loop, but loop-carried flow dependences generally do prevent. Consider the code in SPEC 179.art.

A loop in `scan_recognize` calls `match()`. However, the code communicates a result via a global variable, `pass_flag`. This variable is always initialized on every iteration at line 5 before any uses, and may be updated at line 14 and finally consumed at line 7. Therefore, a loop-*independent* flow dependence exists on `pass_flag`, which means this dependence does not prevent parallelization of the loop.[2] However, if we do not differentiate loop-carried and loop-independent dependences, programmers might think the reported dependences could stop parallelizing the loop.

To handle such loop-independent dependence, we introduce a *killed* address (this is very similar to the *kill* set in a dataflow analysis [2]). We mark an address as killed once the memory address is written in an iteration. Then, all subsequent accesses within the same iteration to the killed address are not stored in the pending table and reported as loop-independent dependences. Killed information is cleared on every iteration.

2. A global variable `pass_flag` may make loop-carried output dependences, but it can be removed easily by privatizing the variable.

In this example, once `pass_flag` is written at line 5, its address is marked as killed. All following accesses on `pass_flag` are not stored in the pending table, and are reported as loop-independent dependences. Since the write operation at line 5 is the first access within an iteration, `pass_flag` does not make loop-carried flow dependences.

3.3 Problems of the Pairwise Method

The pairwise method needs to store all distinct memory references within a loop invocation. Hence, it is obvious that the memory requirement per loop is increased as the memory footprint of a loop is increased. However, the memory requirement could be even worse because we consider nested loops. As explained in Section 3.1, history references of inner loops propagate to their upper loops. Therefore, only when the topmost loop finishes can all the history references within the loop nest be flushed. However, many programs have fairly deep loop nests (for example, the geometric mean of the maximum loop depth in SPEC 2006 FP is 12), and most of the execution time is spent in loops. Hence, whole distinct memory references often need to be stored along with PC addresses throughout the program execution. In Section 4, we solve this problem by using *compression*.

Profiling time overhead is also critical since extremely many memory loads and stores may be traced. We attack this overhead by *parallelizing* the data-dependence profiling itself. We present our solution in Section 5.

4 The SD^3 Algorithm Part I: A Memory-Scalable Algorithm

4.1 Overview of the Algorithm

The basic idea of solving this memory-scalability problem is to store memory references as a *compressed* format. Since many memory references show stride patterns[3], our profiler can also compress memory references with a *stride* format (e.g., `A[a*n + b]`). However, a simple compression technique is not enough to build a scalable data-dependence profiler. We also need to address the following challenges:

- How to detect stride patterns dynamically,
- How to perform data-dependence checking with the compressed format *without* decompression,

3. This is also the motivation of hardware stride prefetchers.

- How to handle loop nests and loop-independent dependence with the compressed format, and
- How to manage both stride and non-stride patterns simultaneously.

The above problems are now discussed in this section.

4.2 Dynamic Detection of Strides

We define that an address stream is a *stride* as long as the stream can be expressed as $base + stride\_distance \times n$. SD^3 dynamically discovers strides and directly checks data dependences with strides and non-stride references. In order to detect strides, when observing a memory access, the profiler trains a stride detector for each PC (or any location identifier of a memory instruction) and decides whether the access is part of a stride or not. Because sources and sinks of dependences should be reported, we have a stride detector per PC. An address that cannot be represented as part of a stride is called a *point* in this paper.

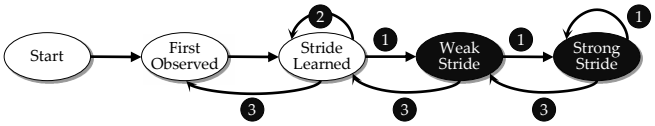


Fig. 6: Stride detection FSM. The current state is updated on every memory access with the following additional conditions: ❶ The address can be represented with the learned stride; ❷ Fixed-memory location access is detected; ❸ The address cannot be represented with the current stride.

Fig. 6 illustrates the state transitions in our stride detector. After watching two memory addresses for a given PC, a stride distance is learned. When a newly observed memory address can be expressed by the learned stride, FSM advances the state until it reaches the *StrongStride* state. The StrongStride state can tolerate a small number of stride-breaking behaviors. For memory accesses like `A[i][j]`, when the program traverses in the same row, we will see a stride. However, when a row changes, there would be an irregular jump in the memory address, breaking the learned stride. Having Weak/StrongStride states tolerates this behavior and increases the opportunity for finding strides.

We separately handle fixed-location memory accesses (i.e., stride distance is zero). If a newly observed memory access cannot be represented with the learned stride, it goes back to the FirstObserved state with the hope of seeing another stride behavior. Note that our stride detector does not always require strictly increasing or decreasing patterns. For example, a stream [10, 14, 18, 14, 18, 22, 18, 22, 26] is considered as a stride $10+4n$ ($0 \leq n \leq 4$). However, such non-strict strides may cause slight errors when calculating the occurrence count of data dependences. We discuss this issue in Section 4.7.

4.3 Stride-Based Dependence Checking Algorithm

Checking dependences is trivial in the pairwise method: we can exploit a hash table keyed by memory addresses, which enables fast searching whether a given memory address is dependent or not. However, the stride-based algorithm cannot use such simple conflict checking because a stride represents an *interval*. Hence, we first search potentially dependent strides and points by using the interval test, and then perform a new data-dependence test, *Dynamic-GCD*.

A naive solution for finding overlapping strides and points would be linear searching between strides and points, but this is extremely slow. Instead, we employ an *interval tree* based on a balanced binary search tree, Red-Black Tree [5]. Given an interval, we find *all* intersecting intervals in the tree, which means finding all overlapping strides and points.[4] Fig. 7 shows an example of an interval tree. Each node represents either a stride or point. Through a query, a stride of [86, 96] overlaps with [92, 192] and [96, 196].

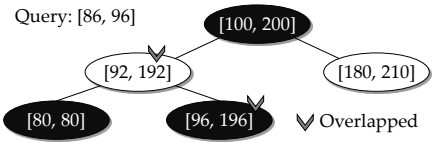


Fig. 7: Interval tree (based on a Red-Black Tree) for fast overlapping point/stride searching. Numbers are memory addresses. Black and white nodes represent Red-Black properties.

The next step is an actual data-dependence test between overlapping strides and points. We extend the well-known GCD (Greatest Common Divisor) test to the *Dynamic-GCD Test* in two directions: (1) We dynamically construct affined descriptors from address streams to use GCD test, and (2) we count the exact number of dependence occurrences (many static-time dependence test algorithms give a *may* answer along with *dependent* and *independent*).

```
1: for (int n = 0; n <= 6; ++n) {
2:   A[2*n + 10] = ...; // Stride 1 (Write)
3:   ... = A[3*n + 11]; // Stride 2 (Read)
4: }
```

Fig. 8: A simple example for DYNAMIC-GCD.

To illustrate the algorithm, consider the code in Fig. 8. We assume that the array `A` is the type of `char[]` and begins at address 10. Then, two strides will be created: (1) [20, 32] with the distance of 2 from line 2, and (2) [21, 39] with the distance of 3 from line 3. Our goal is to calculate the exact number of conflicting addresses in the two strides. The problem is then reduced to solving a Diophantine equation[5]:

$$2x + 20 = 3y + 21 \quad (0 \leq x,\ y \leq 6).$$

DYNAMIC-GCD is described in Algorithm 1, and we detail the steps of computations with Fig. 9:

1) Obtain the overlapped bounds and lengths as in Fig. 9: $low = 21$, $high = 30$, $length = 10$.
2) Check the existence of dependence by GCD, without considering bounds: To use GCD test on strides, $delta$

4. In the worst case, where all the nodes of an interval tree intersect a given input, it requires linear time to find overlapped strides and points. However, on average, an interval tree gives much faster searching time than a linear search. Further optimizations could be done in this step.

5. A Diophantine equation is an indeterminate polynomial equation in which only integer solutions are allowed. In our problem, we solve a linear Diophantine equation such as $ax + by = 1$.

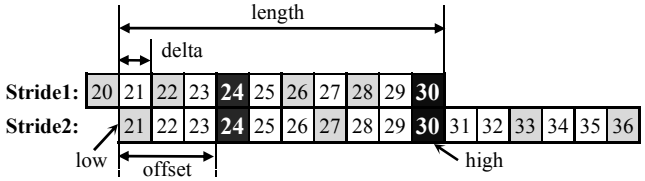


Fig. 9: Two strides in Fig. 8. Lightly shaded blocks indicate accessed memory locations, and black blocks are conflicting locations. The terms (length, delta, low, high, and offset) are explained in the below.

(the distance between low and the immediately following accessed address) is computed. It allows the two strides to be aligned to an imaginary common array that begins at low. Finally, do GCD test. If the test says no dependence, the algorithm halts. In Fig. 9, $delta$ is 1; GCD(2, 3) yields 1, which divides $delta$. Therefore, there *may* be dependences.

3) Compute the number of dependences while considering bounds: EXTENDED-EUCLID [5] yields the smallest solution of the Diophantine equation. In this example, the solution is 24. Then, $offset$ (the distance between low and the first solution) is obtained. Also, observe that the distance between solutions of the equation is LCM (Least Common Multiple) of two stride values. We finally compute the answer (Line 8, 9).
In this example, $offset$ is 3; the LCM of the strides is 6. We conclude that two (24 and 30) addresses are conflicting, and the strides are dependent.

Algorithm 1 DYNAMIC-GCD

INPUTS: Two strides: *low1*, *low2*, *high1*, *high2*, *stride1*, *stride2*
OUTPUT: Return the number of dependences of given strides
Require: $low1 \leq low2$, otherwise swap the strides.
1: Calculate *low*, *high*, and *length* as shown in Fig. 9.
2: $delta \leftarrow (stride1 - (low - low1) \mod stride1) \mod stride1$
3: **if** GCD$(stride1, stride2) \mod delta \neq 0$ **then**
4: **return** 0
5: **end if**
6: Call EXTENDED-EUCLID and obtain x and y. [5]
7: $offset \leftarrow ((stride2 \cdot y \cdot delta/GCD) + LCM) \mod LCM$
8: $result \leftarrow (len - (offset + 1) + LCM)/LCM$
9: $avg\_occur\_count \leftarrow total\_access\_in\_stride/distinct\_access\_in\_stride$
10: **return** $\max(0, result) \cdot avg\_occur\_count$

4.4 Summary of the Memory-Scalable SD³ Algorithm

The first part of SD³, a memory-scalable algorithm, is summarized in Algorithm 2. The algorithm is augmented on top of the pairwise algorithm and will be parallelized to decrease time overhead. Note that we still use the pairwise algorithm for memory references that do not have stride patterns. Algorithm 2 uses the following data structures:

- `STRIDE`: It represents a compressed stream of memory addresses from a PC. A stride consists of (1) the lowest and highest addresses, (2) the stride distance, (3) the size of the memory access, (4) the number of total accesses in the stride, and (5) read/write mode.
- `LoopInstance`: It represents a dynamic execution state of a loop including statistics and tables for data-dependence calculation (pending and history tables).
- `PendingPointTable` and `PendingStrideTable`: They capture memory references in the *current* iteration of a loop. `PendingPointTable` stores point (i.e., non-stride) memory accesses and is implemented as a hash table keyed by the memory address. `PendingStrideTable` remembers strides and is hashed by the PC address of the memory instruction. Note that a PC can generate multiple strides (e.g., `int A[N][M]`). Both pending tables also store *killed bits* to handle loop-independent dependences. See Fig. 10.
- `HistoryPointTable` and `HistoryStrideTable`: They remember memory accesses in *all* executed iterations of a loop so far. The structure is mostly identical to the pending tables except for the killed bits.
- `ConflictTable`: It holds discovered dependences for a loop throughout the program execution.
- `LoopStack`: It keeps the history of a loop execution like the callstack for function calls. A `LoopInstance` is pushed or popped as the corresponding loop is executed and terminated. It is needed to calculate data dependences across loop nests.

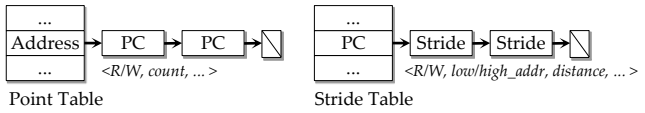


Fig. 10: Structures of point and stride tables: The structures of the history and pending tables are identical. They only differs some fields.

Algorithm 2 THE MEMORY-SCALABLE ALGORITHM

Note: new steps added on top of the pairwise method are underlined.

1: When a loop, L, starts, `LoopInstance` of L is pushed on `LoopStack`.
2: On a memory access, R, of L's i-th iteration, check the killed bit of R. If killed, report a loop-independent dependence, and halt the following steps.
Otherwise, store R in either `PendingPointTable` or `PendingStrideTable` based on the result of the stride detection of R. Finally, if R is a write, set its killed bit.
3: At the end of the iteration, check data dependences. Also, perform stride-based dependence checking. Report any found data dependences.
4: After Step 3, merge `PendingPointTable` with `HistoryPointTable`. Also, merge the stride tables. The pending tables including killed bits are flushed.
5: When L terminates, flush the history tables, and pop `LoopStack`. However, to handle loop nests, we propagate the history tables of L to the parent of L, if exist. Propagation is done by merging the history tables of L with the pending tables of the parent of L.
Meanwhile, to handle loop-independent dependences, if a memory address in the history tables of L is killed by the parent of L, this killed history is not propagated.

Although the essential parts of the algorithm are described in the above sections, the stride-based dependence calculation algorithm also needs a couple of new stride handling algorithms. The following three subsections elaborates these issues.

4.5 Merging Stride Tables for Loop Nests

In the pairwise method, we propagate the histories of inner loops to its upper loops to compute dependences in loop nests. However, introducing strides makes this propagation difficult. Steps 4 and 5 in Algorithm 2 require a *merge* operation of a history table and a pending table. Without strides (i.e., only points), this step is straightforward: we simply compute the union set of the two point hash tables, which takes a linear time, $O(max(size(\texttt{HistoryPointTable}), size(\texttt{PendingPointTable})))$. [6]

However, merging two stride tables is not trivial. A naive solution is just concatenating two stride lists. If this is done, the number of strides could be bloated, resulting in huge memory consumption. Hence, we try to do *stride-level* merging rather than a simple stride-list concatenation. The example is illustrated in Fig. 11.

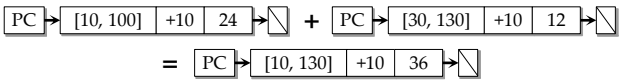


Fig. 11: Two stride lists from the same PC are about to be merged. The stride ([10, 100], +10, 24) means a stride of (10, 20, ..., 100) and total of 24 accesses in the stride. These two strides have the same stride distance. Thus, they can be merged, and the number of accesses is summed.

A naive stride-level merging requires quadratic time complexity. Here we again exploit the interval tree for fast overlapping testing. Nonetheless, we observed that tree-based searching still could take a long time if there is no possibility of stride-level merging. To minimize such waste, the profiler caches the result of the merging test. If a PC shows very little chance of having stride merges, SD^3 skips the merging test and simply concatenates the lists.

4.6 Handling Killed Addresses in Strides

We showed that maintaining *killed* addresses is very important to distinguish loop-carried and independent dependences. As discussed in Section 3.2, the pairwise method prevented killed addresses from being propagated to further steps. However, this step becomes complicated with strides because strides could be killed by the parent loop's strides or points.

Fig. 12 illustrates this case. A stride is generated from the instruction at line 6 when `Loop_5` is being profiled. After finishing `Loop_5`, its `HistoryStrideTable` is merged into `Loop_1`'s `PendingStrideTable`. At this point, `Loop_1` knows the killed addresses from lines 2 and 4. Thus, the stride at line 6 can be killed by either (1) a random point write at line 2 or (2) a write stride at line 4. We detect such killed cases when the history strides are propagated to the outer loop.

6. To make this merging faster, we employ *PC-set optimization*. However, this optimization may lead a slight lossy compression. This issue is discussed in Section 6.2.2.

```
1: for (int i = 0; i < N; ++i) {    // Loop_1
2:   A[rand() % N] = 10;           // Random kill on A[]
3:   for (int j = i; j >= 0; --j)  // Loop_3
4:     A[j] = i;                   // A write-stride
5:   for (int k = 0; k < N; ++k)   // Loop_5
6:     sum += A[k];                // A read-stride
7: }
```

Fig. 12: A stride from line 6 can be killed by either a point at line 2 or a stride at line 4.

Detecting killed addresses is essentially identical to finding conflicts, thus we also exploit an interval tree to handle killed addresses in strides.

Interestingly, after processing killed addresses, a stride could be one of three cases: (1) a shrunk stride (the range of stride address is reduced), (2) two separate strides, or (3) complete elimination. For instance, a stride [4, 8, 12, 16] can be shortened by killed address 16. If a killed address is 8, the stride is divided.

4.7 Lossy Compression in Strides

Our stride-based algorithm essentially uses a compression, which can be either *lossy* or *lossless*. If we only consider a strictly increasing or decreasing stride, SD^3 guarantees the perfect correctness of data-dependence profiling (i.e., results of SD^3 are identical to those of the pairwise method).

However, as discussed in Section 4.2, a stride like [10, 14, 18, 14, 18, 22, 18, 22, 26] is also considered as a stride in our implementation. In this case, our stride format cannot perfectly record the original characteristic of the memory stream. We only remember two facts: (1) a stride of $10 + 4i, (0 \leq i \leq 4)$ and (2) the total number of memory accesses in this stride is 9. Therefore, the stride format cannot precisely remember the occurrence count of each memory address. Such lossy compression may cause slight errors when DYNAMIC-GCD calculates the occurrence count of discovered data-dependence (Line 9 and 10 of Algorithm 1).

Suppose that this stride has a conflict at the address of 26. The address 26 is accessed only one time, but this information is lost. DYNAMIC-GCD estimates the occurrence count of this conflict by taking an average: 9 / 4 = 2.5, the total number of accesses in the stride is divided by the number of distinct addresses in the stride.

Nonetheless, such error does not noticeably affect the usefulness of our approach, because we still guarantee the correctness of the *existence* of data dependences. The error only happens in occurrence counts.

5 The SD^3 Algorithm Part II: Reducing Time Overhead by Parallelization

5.1 Overview of the Algorithm

It is well known that the runtime overhead of data-dependence profiling is very high. A typical method to reduce this overhead would be using sampling techniques. Unfortunately, we cannot use simple sampling techniques for our profiler because it mostly does trade-off between accurate results and low overhead. For example, a dependence pair could be missed due to

sampling, but this pair can prevent parallelization in the worst case. We instead solve the time overhead by *parallelizing* data-dependence profiling itself. In particular, we discuss the following problems:

- Which parallelization model is most efficient?
- How do the stride algorithms work with parallelization?

5.2 A Hybrid Parallelization Model of SD^3

We first survey parallelization models of the profiler that implements Algorithm 2. Before the discussion, we need to explain the structure of our profiler. Our profiler before parallelization is composed of the following three steps:

1) Fetching *events* from an instrumented program: Events include (1) *memory events*: memory reference information such as effective address and PC, and (2) *loop events*: beginning/iteration/termination of a loop, which is essential to implement Algorithm 2. Note that our profiler is an online tool. These events are delivered and processed on-the-fly.
2) Loop execution profiling and stride detection: We collect statistics of loop execution (e.g., trip count), and train the stride detector on every memory instruction.
3) Data-dependence profiling: Algorithm 2 is executed.

Our goal is to design an efficient parallelization model for the above steps. Three parallelization strategies would be candidates: (1) task-parallel, (2) pipeline, and (3) data-parallel. SD^3 exploits a hybrid model of pipeline and data-level parallelism.

With the task-parallel strategy, several approaches could be possible. For instance, the profiler may spawn concurrent tasks for each loop. During a profile run, before a loop is executed, the profiler forks a task that profiles the loop. This is similar to the shadow profiler [24]. This approach is not easily applicable to the data-dependence profiling algorithm because it requires severe synchronization between tasks due to nested loops. Therefore, we do not take this approach.

With pipelining, each step is executed on a different core in parallel. In a data-dependence profiler, the third step, the data-dependence profiling, is the most time consuming step. Hence, the third step will determine the overall speedup of the pipeline. However, we still can hide computation latencies of the first (event fetch) and the second (stride detection) steps from pipelining.

With the data-parallel method, the profiler distributes the collected memory references into different tasks based on a rule. A *task* performs data-dependence checking (Algorithm 2) in parallel with a *subset* of the entire input. It is essentially a SPMD (Single Program Multiple Data) style. Since this data-parallel method is the most scalable one and does not require any synchronizations (except for the final result reduction step, which is very trivial), we also use this model for parallelizing the data-dependence profiling step.

Fig. 13 summarizes the parallelization model of SD^3. We basically exploit pipelining, but the dependence profiling step, which is the longest, is further parallelized. To obtain even higher speedup, we also exploit *multiple* machines (See details in Section 6.2).

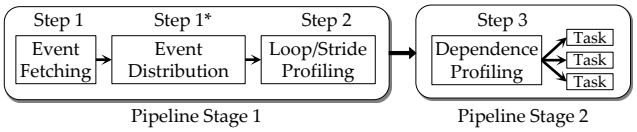


Fig. 13: SD^3 exploits both pipelining (2-stage) and data-level parallelism. Step 1* is augmented for the data-level parallelization.

Note that the *event distribution* step is introduced in the stage 1. Because of a SPMD-style parallelization at the stage 2, we need to prepare inputs for each task. In particular, we divide the address space in an *interleaved* fashion for better speedup, as shown in Fig. 14. The entire address space is divided by every 2^k bytes, and each subset is mapped to M tasks in an interleaved way. Each task only analyzes the memory references from its own range. A thread scheduler then executes M tasks on N cores.

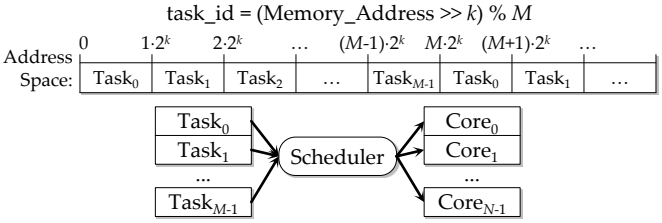


Fig. 14: Data-parallel model of SD^3 with the address-range size of 2^k, M tasks, and N cores: Address space is divided in an interleaved way. The above formula is used to determine the corresponding task id for a memory address. In our experimentation, the address-rage size is 128-byte ($k = 7$), and the number of tasks is the same as the number of cores ($M = N$).

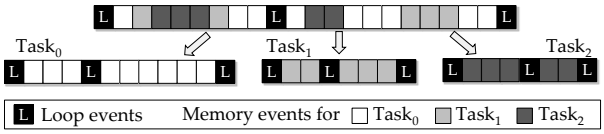


Fig. 15: An example of the event distribution step with 3 tasks: Loop events are duplicated for all tasks while memory events are divided depending on the address-range size and the formula of Fig. 14.

However, this event distribution, as illustrated in Fig. 15, is not a simple division of the entire input events: Memory events are distributed by our interleaved fashion. By contrast, loop events must be duplicated for the correctness of the data-dependence profiling because the steps of Algorithm 2 are triggered on a loop event.

5.3 Strides in Parallelized SD^3

Our stride-detection algorithm and Dynamic-GCD also need to be revised in parallelized SD^3 for the following reason. Stride patterns are detected by observing a stream of memory addresses. However, in the parallelized SD^3, each task can only observe memory addresses in its own address range. The problem is illustrated in Fig. 16. Here, the address space is divided for three tasks with the range size of 4 bytes. Suppose a stride *A* with the range of [10, 24] and the stride distance of 2. However, Task$_0$ can only see addresses in the ranges of [10, 14) and [22, 26). Therefore, Task$_0$ will conclude that there are

two different strides at [10, 12] and [22, 24] instead of only one stride. These broken strides bloat the number of strides dramatically.

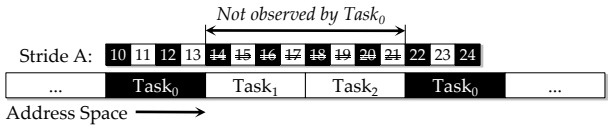


Fig. 16: A single stride can be broken by interleaved address ranges. Stride A will be seen as two separate strides in $Task_0$ with the original stride-detection algorithm.

To solve this problem, the stride detector of a task assumes that any memory access pattern is possible in out-of-my-region so that broken strides can be combined into a single stride. In this example, the stride detector of $Task_0$ assumes that the following memory addresses are accessed: 14, 16, 18, and 20. Then, the detector will create a single stride. Even if the assumption is wrong, the correctness is not affected: To preserve the correctness, when performing Dynamic-GCD, SD^3 excludes the number of conflicts in out-of-my-region.

5.4 Details of the Data-Parallel Model

5.4.1 Choosing a good address-range size

A key point in designing the data-parallel model is to obtain higher speedup via good load balancing. However, the division of the address space inherently makes a load unbalancing problem as memory accesses often show non-uniform locality. Obviously, having too small or too large address-rage size would worsen this problem. Hence, we use an interleaved division as discussed and then need to find a reasonably balanced address-range size.

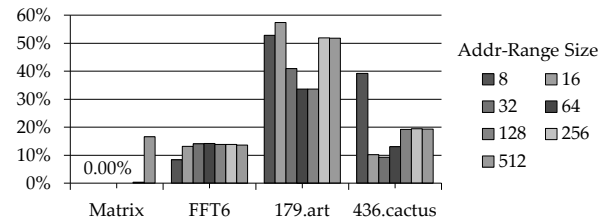


Fig. 17: Deviations of total number of memory accesses of eight tasks by the address-rage sizes (in bytes). Lower is better.

According to our experiment, shown in Fig. 17, as long as the range size is not too small or not too large, address-range sizes from 64 to 256 bytes yield well-balanced workload distribution. In our implementation, we choose 128 bytes.

5.4.2 Choosing an optimal number of tasks

Even if taking the interleaved approach, we cannot avoid the load unbalancing problem. To address this problem, we attempt to create sufficient tasks and employ the work-stealing scheduler [3], that is, exploiting fine-granularity task parallelism. At a glance, this approach would yield better speedup, but our data negated our hypothesis, as shown in Fig 18. We observed that no speedup was gained by this approach.

There are two reasons: (1) First, even if the quantity of the memory events is reduced, the number of stride may not be proportionally reduced. For example, in Fig. 16, despite the revised stride-detection algorithm, the total number of stride for all tasks is three; on a serial version of SD^3, the number of stride would have been one. Hence, having more tasks may increase the overhead of storing and handling strides, eventually resulting in poor speedup. (2) Second, the overhead of the event distribution would be significant as the number of tasks increase. Recall again that loop events are duplicated while memory events are distributed. This restriction makes the event distribution a complex and memory-intensive operation. On average, for SPEC 2006 with the train inputs, the ratio of the total size of loop events to the total size of memory event is 8%. Although the time overhead of processing a loop event is much lighter than that of a memory event, the overhead of transferring loop events could be serious as the number of tasks is increased.

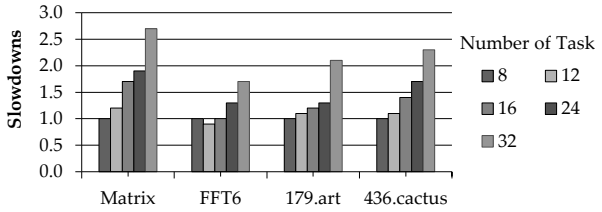


Fig. 18: Having more tasks than the number of cores (on a eight-core machine) exhibits *slowdowns* in our hybrid parallelization model.

Therefore, we let the number of tasks be identical to the number of cores. Although the data-dependence profiling is embarrassingly parallel, the mentioned challenges, handling strides and distributing events, hinder an optimal workload distribution and an ideal speedup.

6 Implementation

Building a profiler that implements SD^3 has many implementation challenges. We discuss important issues in this section. We believe the discussion would be informative to the implementation of other program analysis tools that exploit instrumentation.

We implement SD^3 on both Pin [22], a dynamic binary-level instrumentation toolkit and LLVM [20], a compiler framework. SD^3 algorithm itself is orthogonal to the choice of instrumentation mechanisms. We first discuss the common issues regardless of Pin and LLVM. In the end of this section, we elaborate issues specific to each instrumentation method.

6.1 Basic Architecture

Our profiler consists of *tracer* and *analyzer*, which is a typical producer and consumer architecture. These two modules are executed as separate processes.

- Tracer: It instruments a program, captures runtime execution traces (i.e., memory events and loop events), and transfers to the analyzer via an inter-process communication mechanism, using shared memory.
- Analyzer: It takes events from the tracer and performs SD^3 algorithm.

Note that our profiler must be an *online tool*. Because majority of loads and stores are instrumented, generated traces could be extremely huge, up to an order of 10 TB. Hence, we cannot simply use an offline approach. An example of such offline approach would be storing and compressing events (e.g., by using bzip) and then decompressing and analyzing the events. This approach is not effective at all, because compressing/decompressing traces take huge time.

One concern of this online approach would be the overhead of inter-process communication. We observed that the average amount of event transfer rate between the two processes was approximately 1 - 3 GB/s, which can be sufficiently handled by moderen computers. The size of the execution event is 12 bytes, and events are transferred as uncompressed.

This separation of tracer and analyzer enables two significant benefits. First, the pipeline parallelism, explained in Section 5.2, is easily achieved. Second, we can design the analyzer to be reused by different tracers. We separately implement tracers based on instrumentation mechanisms. We also define an abstracted communication layer between the single analyzer and multiple tracers, regardless of the choice of instrumentaion tools. Finally, such separation eases debugging of SD^3 algorithm.

6.2 Implementation of Analyzer

The analyzer first implements the data structures described in Section 4.4 and Algorithm 2, and we then parallelize by using Intel Threading Building Block (TBB) [13].

To obtain even better parallelism, we extend our profiler to work on *multiple machines*, based on a MPI-like execution model [9]. The same tracer, analyzer, and application are running in parallel on multiple machines, but each machine has equally divided workload. This is a simple extension of our data-parallel model, but applies across different machines. Our profiler also profiles multithreaded applications and provides per-thread profiling results.

6.2.1 False Positive and False Negative Issues

We discuss regarding false positives (i.e., reported as having dependences, but it was a false alarm) and false negatives (i.e., no dependences reported, but it has a dependence) in the data-dependence profiling.

False positives can occur if we take a bigger granularity in the memory instruction instrumentation, such as 8-byte granularity rather than a byte granularity. A data-dependence profiling in the speculative multithreading domain can take a large granularity to minimize overhead, but this approach suffers more false positive dependences [4].

In our implementation, first of all, the stride-based approach does not suffer from false positives. We correctly handle the size of memory access (e.g., whether `char`, `int`, or `double`) in the stride-based data structures and DYNAMIC-GCD.

For the pairwise method in which hash tables keyed by addresses, we always use 1-byte granularity, which means no false positives. However, it may have false negatives in a very unusual case, as shown in Fig. 19. Even if there was a 8-byte write at line 4, the 1-byte granularity policy only records the first byte of the access. Hence, the read from line 5 results in a missing data dependence. However, we believe such case is very unlikly to happen in well-written code. Of course, this false negative problem can be solved by a naive approach: inserting every bytes into a hash table on multiple-byte accesses. This approach obviously requires too more memory and time to be used.

```
1: void*   raw   = malloc(1024);
2: double* data1 = (double*)raw;
3: char*   data2 = (char*  )raw;
4: data1[0] = 1.0;        // Writing 8 bytes
5: char t   = data2[1];   // Reading only part of data1[0];
```

Fig. 19: A false negative case with 1-byte granularity.

False negatives can also happen because not all code can be executed with a specific input. This problem is discussed in Section 7.4.

6.2.2 PC-set optimization for the Pairwise Method

This PC-set optimization is only for the pairwise method, but this optimization can reduce both time and space overhead. Recall SD^3 still uses the pairwise method when a memory access does not show the stride behavior. As discussed in Section 4.5, the pairwise method merges two point tables when the current iteration finishes and when an inner loop is terminated.

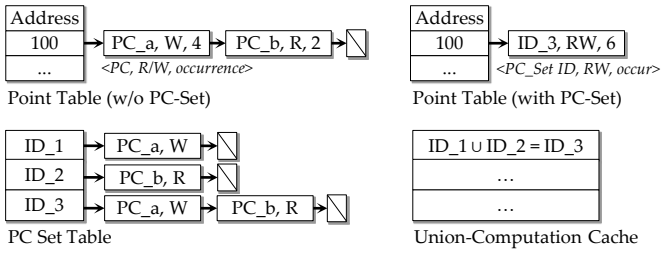


Fig. 20: PC set optimization for the fast PC-list merging. A PC list can be represented as a single PC-set ID, resulting in saving the memory. The union computation is be accelerated by the cache. However, this optimization can make an error in the frequency of dependence.

However, note that each entry of a point table has a *list* of PC, which is required to report PC-wise sources and sinks of dependences. Also, when merging two point tables, we need to compute an union set of two PC lists from two entries, which requires additional time. Fortunately, we observed that most of such union computation was repeated over a limited number of distinct PC lists. Hence, we introduce a global *PC-set table* that remembers all observed distinct PC lists, and a *cache* for the union computation. These two data structures, shown in Fig. 20, not only save the total memory consumption, but also avoid excessive computation time.

This PC-set optimization causes another lossy compression like the error discussed in Section 4.7. As illustrated in Fig. 20, introducing PC-set loses the occurrence count per each PC; the total occurrence count for the single PC-set is just saved. Hence, when reporting the frequency of the data dependence, the pairwise method may have an error. However, we also should note that this error is not for the existence of the dependence, but only for the slight frequency.

6.3 Implementation of Tracers

6.3.1 Issues in a Pin-based Tracer

A Pin-based tracer enables the data-dependence profiling at dynamic and binary level. This approach broadens the applicability of the tool, comparing to a compiler and source-code level approach. A dynamic instrumentation does not require a recompilation of a profilee. It is a great benefit if the application does not have full source code and requires different and complex tool chains.

The downside of Pin-based approach is that additional binary-level static analysis is needed to recover control flow graphs and loop structures, which is hard to implement. For example, recovering indirect branches and pinpointing the correct locations of loop entries and exits are challenge in the binary-level analysis.

Regarding the instrumentation of loads and stores, a binary executable typically has a lot of artifacts from push/pop on stacks and system function calls. Without eliminating such redundant loads and stores, results of a Pin-based profiler would have lots of dependences that are not useful for the parallelization hints. Some loads and stores also do not need to be instrumented if their dependences can be identified at static time, notably inductions and reductions. However, filtering such loads and stores selectively is also hard to implement. An alternative to this direct binary-level static analysis would be using a x86 binary translator to LLVM IR and then exploiting the LLVM framework [17].

6.3.2 Issues in a LLVM-based Tracer

Another choice of implementation of a tracer is using compiler-based instrumentation such as LLVM. Although LLVM also allows dynamic compilation and instrumentation, we use as a static and source-level instrumentation toolkit. LLVM provides very rich static-analysis infrastructure. Correct control flows and loop structures are already provided. Skipping inductions and reductions is relatively easy to implement inside of LLVM. Source-level instrumentation is easy to avoid binary-level artifacts. Idealy, some static analysis may be performed before the dynamic profiling to decrease the profiling overhead [7].

However, recompiling an application with instrumentation code is not always easy. It sometimes requires modifications in compiler toolchains and compiler driver code. The analyzer must need some information from the instrumentation phase, such as list of instrumented loops and memory instructions. As the instrumentation phase is separated from the runtime profiling, such information should be transfered via a persistent medium like a file.

7 Experimental Results

7.1 Experimentation Methodology

We use 22 SPEC 2006 benchmarks to report runtime overhead by running the *entire* execution of benchmarks with the reference input.[7] We instrument all memory loads and stores except for certain types of stack operations and some memory instructions in shared libraries. Our profiler collects details of data-dependence information as enumerated in Section 2.1. However, we only profile the top 20 hottest loops (based on the number of executed instruction). For the comparison of the overhead, we use the pairwise method. We also use seven OmpSCR benchmarks [1] for the input sensitivity problem.

Our experimental results were obtained on machines with Windows 7 (64-bit), 8-core with Hyper-Threading Technology, and 16 GB main memory. Memory overhead is measured in terms of the peak physical memory footprint. For results of multiple machines, our profiler runs in parallel on multiple machines but only profiles distributed workloads. We then take the slowest time for calculating speedup.

7.2 Memory Overhead of SD^3

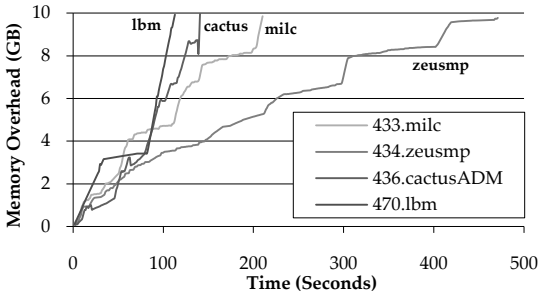


Fig. 23: Memory overhead of the pairwise for 4 SPEC 2006 benchmarks

Fig. 21 shows the absolute memory overhead of SPEC 2006 with the reference inputs. The memory overhead includes everything: (1) native memory consumption of a benchmark, (2) instrumentation overhead, and (3) profiling overhead. Among the 22 benchmarks, 21 benchmarks cannot be profiled with the pairwise method, which is still the state-of-the-art method, with a 12 GB memory budget. Many of the benchmarks (16 out of 22) consumed more than 12 GB even with the train inputs. Fig. 23 shows the memory consumption of the pairwise method on every second for 433.milc, 434.zeusmp, 435.lbm and 436.cactusADM. Within 500 seconds, these four benchmarks reached 10 GB memory consumption. We do not even know how much memory would be needed to complete the profiling with the pairwise method. We also tested 436.cactus and 470.lbm on a 24 GB machine, but still failed. Simply doubling memory size could not solve this problem.

However, SD^3 successfully profiled all the benchmarks. For example, while both 416.gamess and 436.cactusADM demand 12+ GB in the pairwise method, SD^3 requires only 1.06 GB (just 1.26× of the native overhead) and 1.02 GB (1.58× overhead), respectively. The geometric mean of the memory consumption of SD^3 (1-task) is 2113 MB while the overhead of native programs is 158 MB. Although 483.xalancbmk

7. Due to issues in instrumentation and binary-level analysis, we were not able to run the remaining 6 SPEC benchmarks. Please note that current LLVM implementation cannot instrument Fortran programs. The results in this paper is from the Pin-based profiler.

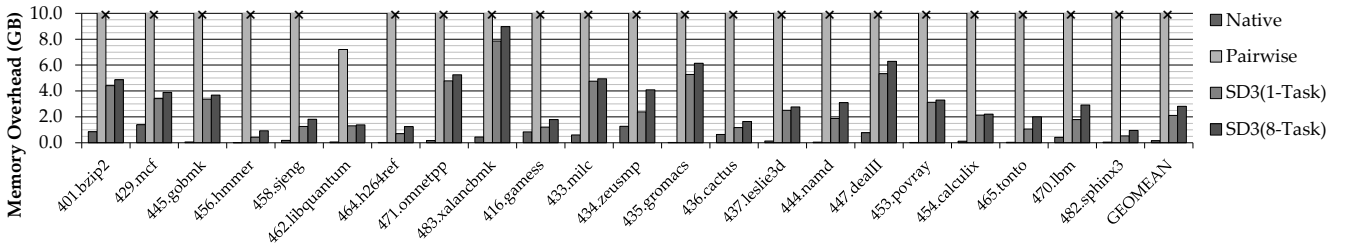


Fig. 21: Absolute memory overhead for SPEC 2006 with the reference inputs: 21 out of 22 benchmarks (× mark) need more than 12 GB in the pairwise method that is still the state-of-the-art algorithm of current tools. The benchmarks natively consume 158 MB memory on average.

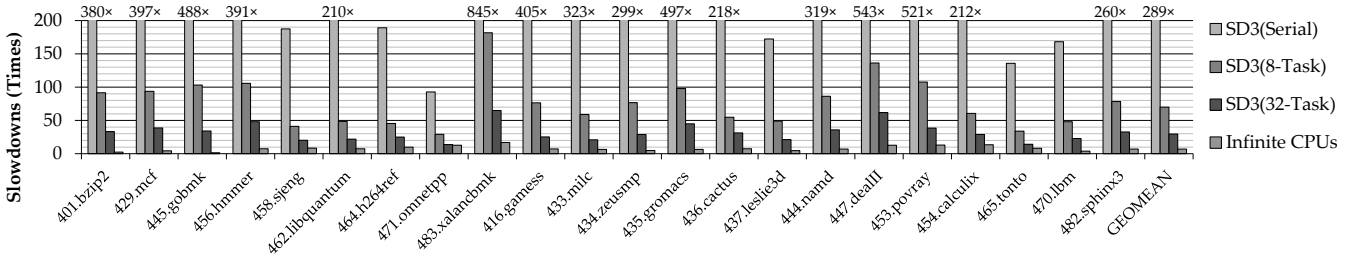


Fig. 22: Slowdowns (against the native run) for SPEC 2006 with the reference inputs: From left to right, (1)SD^3 (1-task on 1-core), (2) SD^3 (8-task on 8-core), (3) SD^3 (32-task on 32-core), and (4) estimated slowdowns of infinite CPUs. For all experiments, the address-range size is 128 bytes. The geometric mean of native runtime is 488 seconds on Intel Core i7 3.0 GHz.

needed more than 7 GB, we can conclude that the stride-based compression is very effective.

Parallelized SD^3 naturally consumes more memory than the serial version of SD^3, 2814 MB (8-task) compared to 2113 MB (1-task) on average. The main reason is that each task needs to maintain a copy of the information of the entire loops to remove synchronization. Furthermore, the number of total strides is generally increased compared to the serial version since each task maintains its own strides. However, SD^3 still reduces memory consumption significantly against the pairwise method.

7.3 Time Overhead of SD^3

The time overhead results of SD^3 is presented in Fig. 22. The time overhead includes both instrumentation-time analysis and runtime profiling overhead. The instrumentation-time overhead, such as recovering loops, is quite small. For SPEC 2006, this overhead is only 1.3 seconds on average. The slowdowns are measured against the execution time of native programs. As discussed in Section 5.4, the number of task is the same as the number of core in the experimentations.

As shown in Fig. 22, serial SD^3 shows a 289× slowdown on average, which is not surprising given the quantity of computations on every memory access and loop execution. Note that we do an exhaustive profiling for the top 20 hottest loops. The overhead could be improved by implementing better static analysis that allows us to skip instrumenting loads and stores that have proved not to make any data dependences.

When using 8 tasks on 8 cores, parallelized SD^3 shows a 70× slowdown on average, 29× and 181× in the best and worst cases, respectively. We also measure the speedup with four eight-core machines (total 32 cores). On 32 tasks with 32 cores, the average slowdown is 29×, and the best and worst cases are 13× and 64×, respectively, compared to the native execution time. Calculating the speedups over the serial SD^3, we achieve 4.1× and 9.7× speedups on eight and 32 cores, respectively.

Although the data-dependence profiling stage is embarrassingly parallel, our speedup is lower than the ideal speedup (4.1× speedup on eight cores). The first reason is that we have an inherent load unbalancing problem. The number of tasks is equal to the number of cores to minimize redundant loop handling and event distribution overhead. Note that the address space is statically divided for each task, and there is no simple way to change this mapping dynamically. Second, with the stride-based approach, processing time for handling strides is not necessarily decreased in the parallelized SD^3.

We also estimate slowdowns with infinite CPUs. In such case, each CPU only observes conflicts from a *single* memory address, which is extremely light. Therefore, the ideal speedup would be very close to the runtime overhead without the data-dependence profiling. However, some benchmarks, like 483.xalancbmk and 454.calculix, show 17× and 14× slowdowns even without the data-dependence profiling. The large overhead of the loop profiling mainly comes from frequent loop start/termination and deeply nested loops.

7.4 Input Sensitivity of Data-Dependence Profiling

One of the concerns of using a data-dependence profiler as a programming-assistance tool is the input sensitivity problem. We quantitatively measure the *similarity* of data-dependence profiling results from different inputs. A profiling result has a list of discovered dependence pairs (source and sink). We compare the discovered dependence pairs from a set of different inputs. Note that we only compare the top 20 hottest loops and ignore the frequency of the data-dependence pairs. We define similarity as follows, where R_i is the i-th result (i.e., a set of data-dependence pair):

$$\text{Similarity} = 1 - \sum_{i=1}^{N} \frac{\left|R_i - \bigcap_{k=1}^{N} R_k\right|}{|R_i|}$$

The similarity of 1 means all sets of results are exactly the same (no differences in the existence of discovered data-dependence pairs, but not frequencies). We first tested eight benchmarks in the OmpSCR [1] suite. All of them are small numerical programs, including FFT, LUReduction, and Mandelbrot. We tested them with three different input sets by changing the input data size or iteration count, but the input sets are sufficiently long enough to execute the majority of the source code. Our result shows that the data-dependence profiling results of OmpSCR were *not* changed by different input sets (i.e., Similarity = 1). Therefore, the parallelizability prediction of OmpSCR by our profiler has not been changed by the input sets we gave.

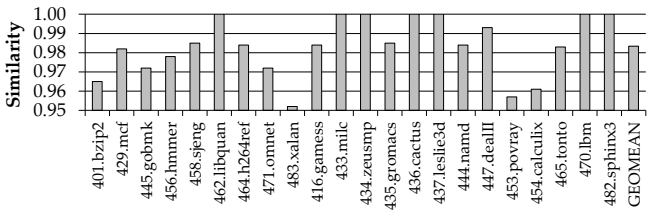


Fig. 24: Similarity of the results from different inputs: 1.00 means all results were identical (not the frequencies of dependence pairs).

We also tested SPEC 2006 benchmarks. We obtained the results from the reference and train input sets. Our data shows that there are very high similarities (0.98 on average) in discovered dependence pairs. Note that again we compare the similarity for only frequently executed loops. Some benchmarks show a few differences (as low as 0.95), but we found that the differences were highly correlated with the executed code coverage. In this comparison, we tried to minimize x86-64 specific artifacts such as stack operations of prologue/epilogue of a function.

A related work also showed a similar result. Thies et al. used a dynamic analysis tool to find pipeline parallelism in streaming applications with annotated code [29]. Their results showed that memory dependences between pipeline stages are highly stable and predictable over different inputs.

7.5 Discussion

We discuss two questions for designing SD^3 algorithm: the effectiveness of stride compression and samplings.

7.5.1 Opportunities for Stride Compression

We would like to see how many opportunities exist for stride compression. We easily expect that a regular and numerical program has a higher chance of stride compression than an integer and control-intensive program. Fig. 25 shows the distributions of the results from the stride detector when it profiles the entire memory accesses of SPEC 2006 with the train inputs. Whenever observing a memory access, our stride detector classifies the access as one of the three categories: (1) stride, (2) fixed-location access, or (3) point (i.e., non-stride). Clearly, SPEC FP benchmarks have a higher chance of stride compression than INT benchmarks: 59% and 28% of all the accesses show stride behaviors, respectively. Overall, 46% of the memory accesses are classified as strides. However, note that there are some benchmarks that have similar distributions of stride and non-stride accesses such as 444.namd. Thus, efficiently handling stride and point accesses simultaneously is important.

7.5.2 Sampling Technique

Instead of directly addressing the scalability issue, some previous work has tried to limit the overhead of data-dependence profiling using *sampling* [4]. However, simple sampling techniques are inadequate for the purpose of our data-dependence profiler: Any sampling technique can introduce inaccuracy. Inaccuracy in data-dependence profiling can lead to either false negatives or false positives (discussed in Section 6.2.1).

Some usage models of data-dependence profiling can tolerate inaccuracy. One such example is thread-level data speculation (TLDS) [4, 21, 28]. The reason is that in TLDS, incorrect speculations can be recovered by the rollback mechanism in hardware. Nevertheless, when we use dependence profiling to guide programmer-assisted parallelization for multicore processors without such rollbacks, the profile should be as accurate as possible.

```
307: for(item = 0 ; item < group->n_scheditems; item++)
308: {
309:   switch(group->scheditems[group->order[item]].type)
310:   {
311:     case sched_function: // call work function
...
322:     case sched_group:    // call work function
...
338:   }
339: }
```

Fig. 26: 436.cactusADM, ScheduleTraverse.c

Fig. 26 illustrates a case where a simple sampling could make a wrong decision on parallelizability prediction. The loop at line 307 of the figure is mistakenly reported as parallelizable. The reason was that the `work` function that generated dependences was called just one time, at the end of iteration. A simple sampling technique that randomly samples iterations in a loop could make this error.

8 Related Work

8.1 Dynamic Data-Dependence Analysis

One of the early works that used data-dependence profiling to help parallelization is Larus' parallelism analyzer `pp` [18]. `pp` detects loop-carried dependences in a program. The profiling algorithm is similar to the evaluated pairwise method. Larus analyzed six programs and showed that two different groups, numeric and symbolic programs, had different dependence behaviors. However, this work had huge memory and time overhead, as we demonstrated in this paper.

Tournavitis et al. [30] proposed a dependence profiling mechanism to overcome the limitations of automatic parallelization. However, in their work, they also used a pairwise-like method and did not discuss the scalability problem. Zhang et al. [36] proposed a data-dependence-distance profiler called Alchemist. Their tool is specifically designed for the *future* language constructor that represents the result

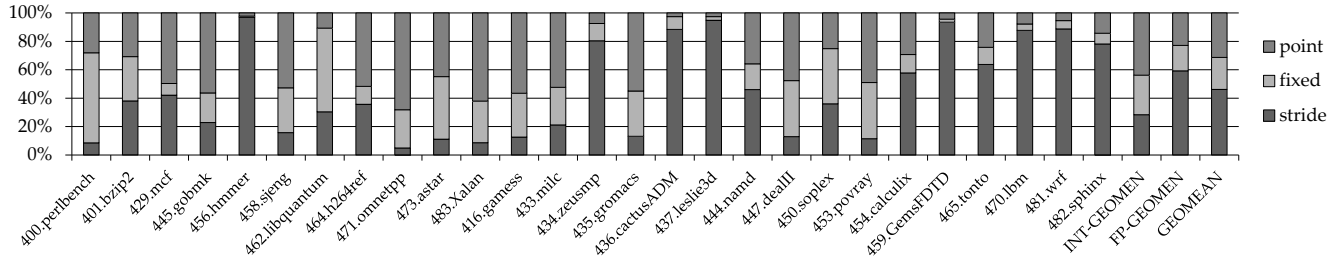


Fig. 25: Classification of stride detection for SPEC 2006 benchmarks with the train inputs: (1) point: memory references with non-stride behavior, (2) fixed: memory references whose stride distance is zero, and (3) stride: memory references that can be expressed in an affined descriptor.

of an asynchronous computation. Although they claimed there was no memory limitation in their algorithm, they evaluated very small benchmarks. The number of executed instructions of a typical SPEC 2006 reference run is on the order of 10^{12}, while that of the evaluated programs in Alchemist is less than 10^8. Praun et al. [32] proposed a notion of dependence density, the probability of existing memory-level dependencies among any two randomly chosen tasks from the same program phase. They also implemented a dependence profiler by using Pin and had a runtime overhead similar to our pairwise method.

8.2 Data-Dependence Profiling for Speculation

As discussed in Section 7.5, the concept of dependence profiling has been used for speculative hardware based optimizations. TLDS compilers speculatively parallelize code sections that do not have much data dependence. Several methods have been proposed [4, 8, 21, 28, 34], and many of them employ sampling or allow aliasing to reduce overhead. However, all of these approaches do not have to give accurate results like SD^3 since speculative hardware would solve violations in memory accesses.

8.3 Reducing Overhead of Dynamic Analysis

Shadow Profiling [24], SuperPin [33], PiPA [37], and Ha et al. [10] employed parallelization techniques to reduce the time overhead of instrumentation-based dynamic analyses. Since all of them focus on a generalized framework, they only exploit task-level parallelism by separating instrumentation and dynamic analysis. However, in our case, SD^3 further exploits data-level parallelism while reducing the memory overhead simultaneously.

A number of techniques that compress dynamic profiling traces have been proposed to save memory space [19, 23, 26, 35] as well as standard compression algorithms like bzip. Their common approach is using specialized data structures and algorithms to compress instructions and memory accesses that show specific patterns. METRIC [23], a tool to find cache bottlenecks, also exploits the stride behavior like SD^3. However, the fundamental difference is that their algorithm is only for compressing memory streams. SD^3 is not a simple compression algorithm; we present a number of algorithms that effectively calculates data dependence with the stride and non-stride formats.

9 Applications of SD^3 Algorithm

SD^3 is developed to solve the scalability for tools that assist programmers for easier manual parallelization from serial code. SD^3 algorithm is directly applicable to tools such as Intel Parallel Studio [12], CriticalBlue Prism [6], and Vector Fabric vfAnalyst [31]. As claimed in the Introduction, these tools currently show high overhead. DProf by Das and Wu [7] is a dependence profiler to assist parallelization, but also suffers from huge overhead. We have confirmed that some of the current tools may not distinguish loop-independent dependences from loop-carried dependences. SD^3 will provide more accurate data-dependence analysis algorithm.

This paper have focused on a precise algorithm to calculate data dependences in loop nests. However, we should note that SD^3 is easily extended to find data dependences among arbitrary function calls (including recursion). On a function entry, we assume that an imaginary loop that encloses the entire function body has been started, while the trip count is just one. Any data dependences between statements in the same function are detected as loop-independent dependences. Any data dependences across function calls are handled because SD^3 correctly calculates data dependences in loop nests.

A raw result from SD^3 algorithm is pairs of data dependences for a given program. If no data dependence reported, it obviously means that this code section can be embarrassingly parallelized. However, many serial code would have data dependences. SD^3 does not give hints on code transformation to avoid such data dependences. Unfortunately, the current tools also do not provide good hints to programmers. The interpretation of the discovered data dependences is the task of programmers at this time.

Our future work, Prospector [15], is to provide more useful hints on code transformation to programmers. As illustrated in an example of Prospector [15], by further analysis of discovered data dependences, we may classify some of dependences as reduction variables and assist to change the code. More sophisticated analyses would be possible. If discovered flow dependences do not dependent on order, it can be avoided by inserting critical sections. Jin et al. presented a similar idea recently [14], although their work is to fix concurrency bugs. If flow dependences should obey some computation order, we may provide hints on inserting synchronization such as condition variables and phasers [27]. Finally, if flow data dependences can be grouped into several steps, `DOACROSS` and pipelining could be advised.

10 Conclusions and Future Work

This paper proposed a new scalable data-dependence profiling technique called SD^3. Although data-dependence profiling is an important technique for helping parallel programming, it has huge memory and time overhead. SD^3 is the first solution that attacks both memory and time overhead at the same time. For the memory overhead, SD^3 not only reduces the overhead by compressing memory references that show stride behaviors, but also provides a new data-dependence checking algorithm with the stride format. SD^3 also presents several algorithms on handling the stride data structures. For the time overhead, SD^3 parallelizes the data-dependence profiling itself while keeping the effectiveness of the stride compression. SD^3 successfully profiles 22 SPEC 2006 benchmarks with the reference inputs.

In future work, we will focus on how such a scalable data-dependence profiler can actually provide advice on parallelizing legacy code, as discussed in Section 9. We hope that SD^3 can help many researchers to develop other dynamic tools to assist parallel programming.

References

[1] OmpSCR: OpenMP source code repository. http://sourceforge.net/projects/ompscr/.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP*, 1995.

[4] T. Chen, J. Lin, X.Dai, W. Hsu, and P. Yew. Data dependence profiling for speculative optimizations. In *Proc. of 14th Int'l Conf on Compiler Construction (CC)*, 2004.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[6] CriticalBlue. *Prism: an analysis exploration and verification environment for software implementation and optimization on multicore architectures*. http://www.criticalblue.com.

[7] D. Das and P. Wu. Experiences of using a dependence profiler to assist parallelization for multi-cores. In *IPDPS Workshops*, pages 1–8, 2010.

[8] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *PLDI*, 2004.

[9] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, Cambridge, MA, USA, 1994.

[10] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *OOPSLA*, 2009.

[11] Intel Corporation. *Intel Compilers*. http://software.intel.com/en-us/intel-compilers/.

[12] Intel Corporation. *Intel Parallel Studio*. http://software.intel.com/en-us/intel-parallel-studio-home/.

[13] Intel Corporation. *Intel Threading Building Blocks*. http://www.threadingbuildingblocks.org/.

[14] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *PLDI*, 2011.

[15] M. Kim, H. Kim, and C.-K. Luk. Prospector: Helping parallel programming by a data-dependence profiler. In *2nd USENIX Workshop on Hot Topics in Parallelism (HotPar '10)*, 2010.

[16] X. Kong, D. Klappholz, and K. Psarris. The I Test: An improved dependence test for automatic parallelization and vectorization. *IEEE Transactions on Parallel and Distributed Systems*, 2(3), 1991.

[17] A. Kotha, K. Anand, M. Smithson, G. Yellareddy, and R. Barua. Automatic parallelization in a binary rewriter. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, 2010.

[18] J. R. Larus. Loop-level parallelism in numeric and symbolic programs. *IEEE Trans. Parallel Distrib. Syst.*, 4(7), 1993.

[19] J. R. Larus. Whole program paths. In *PLDI*, 1999.

[20] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, 2004.

[21] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. Posh: a tls compiler that exploits program structure. In *PPoPP*, 2006.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[23] J. Marathe, F. Mueller, T. Mohan, S. A. Mckee, B. R. De Supinski, and A. Yoo. METRIC: Memory Tracing via Dynamic Binary Rewriting to Identify Cache Inefficiencies. *ACM Transactions on Programming Languages and Systems*, 29(2), 2007.

[24] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *CGO-5*, 2007.

[25] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[26] G. D. Price, J. Giacomoni, and M. Vachharajani. Visualizing potential parallelism in sequential programs. In *PACT-17*, 2008.

[27] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, 2008.

[28] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *ISCA*, 2000.

[29] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline

parallelism in C programs. In *MICRO-40*, 2007.

[30] G. Tournavitis, Z. Wang, B. Franke, and M. O'Boyle. Towards a holistic approach to auto-parallelization integrating profile-driven parallelism detection and machine-learning based mapping. In *PLDI*, 2009.

[31] VectorFabrics. *vfAnalyst: Analyze your sequential C code to create an optimized parallel implementation*. http://www.vectorfabrics.com/.

[32] C. von Praun, R. Bordawekar, and C. Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *PPoPP*, 2008.

[33] S. Wallace and K. Hazelwood. SuperPin: Parallelizing dynamic instrumentation for real-time performance. In *CGO-5*, 2007.

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

[35] X. Zhang and R. Gupta. Whole execution traces. In *MICRO-37*, 2004.

[36] X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *CGO-7*, 2009.

[37] Q. Zhao, I. Cutcutache, and W.-F. Wong. PiPA: pipelined profiling and analysis on multi-core systems. In *CGO-6*, 2008.