To Appear in the Proceedings of the
43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), December 2010.

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

Minjang Kim Hyesoon Kim
School of Computer Science
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{*minjang, hyesoon*}*@cc.gatech.edu*

Chi-Keung Luk
Technology Pathfinding and Innovations
Software and Services Group
Intel Corporation
Hudson, MA 01749
chi-keung.luk@intel.com

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.

Keywords-**profiling, data dependence, parallel programming, program analysis, compression, parallelization.**

I. Introduction

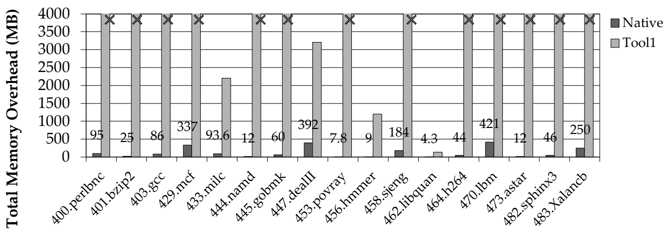
As multicore processors are now ubiquitous in mainstream computing, parallelization has become the most important approach to improving application performance. However, specialized software support for parallel programming is still immature although a vast amount of work has been done in supporting 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 [10], CriticalBlue Prism [6], and Vector Fabric vfAnalyst [26] 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 [21] and Banerjee's inequality test [14] 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 automatic parallelizing compilers often failed to parallelize simple embarrassingly parallel loops written in C/C++. It also had limited success in irregular data structures due to pointers and indirect memory accesses.

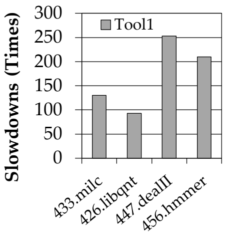
Rather than entirely relying on static analysis, dynamic analysis using *data-dependence profiling* is an alternative or a complementary approach to address the limitations of the static approach since all memory addresses are resolved in runtime. Data-dependence profiling has already been used in parallelization efforts like speculative multithreading [4, 7, 17, 23, 29] and finding parallelism [15, 22, 25, 27, 31]. 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.

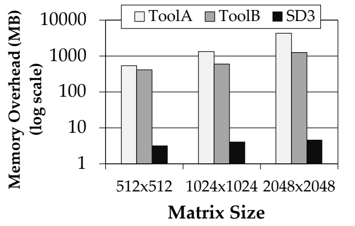
As a concrete example, Figure I demonstrates the scalability issues of our baseline algorithm. We believe that this baseline algorithm is still state-of-the-art and very similar to the current tools. Figures I(a) and I(b) show the memory and time overhead, respectively, when profiling 17 SPEC 2006 C/C++ 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. The runtime overhead is between 80× and 270× for the four benchmarks



(a) Memory overhead of Tool1

(✕: Data-dependence profiling takes more than 10 GB memory.)





(b) Time overhead of Tool1 (c) Memory overhead of matrix addtion

Figure 1. Overhead of our baseline algorithm and current tools.

that worked. While both time and memory overhead are severe, the latter will stop further analysis. The culprit is the dynamic data-dependence profiling algorithm used in the baseline, which we call the *pairwise method* [4, 15]. This algorithm needs to store all outstanding memory references in order to check dependences, which can easily lead to memory bloats.

Another example that clearly shows the memory scalability problem is a simple matrix addition program that allocates three N by N matrices ($A = B + C$). As shown in Figure I(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. The failed benchmarks in Figure I(a) are successfully profiled by SD^3.
2) *Reducing runtime overhead with parallelization:* We show that data-dependence profiling itself can be effectively parallelized. We demonstrate 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.

II. Background on Data-Dependence Profiling

A. The Usage Model of Data-Dependence Profiler.

Before describing SD^3, we briefly describe the usage model of our dynamic data-dependence profiler, as shown in Figure 2.

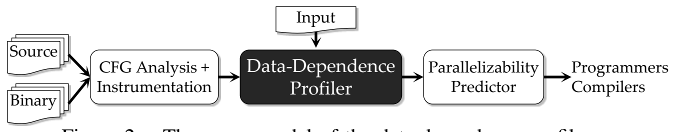


Figure 2. The usage model of the data-dependence profiler.

The profiler takes a program in either source code or binary and profiles with a representative input. In the first step, the profiler performs a basic control-flow analysis such as loop extraction and instrumentation. In the second step, it dynamically profiles data dependence. In the third step, it does post-processing of the profiling result such as identifying parallelizable code sections, especially parallelizable loops, and suggests how to parallelize them. If a code section is not easily parallelizable, we report detailed information of the discovered data dependences. This final data helps programmers on parallelization and also can be used by of compiler optimizations [25]. Among the three steps, obviously the data-dependence profiler is the bottleneck of the scalability problem, and we focus on this.

The data-dependence profiler provides all or some of the following information:

- **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**.

B. 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 I shows an example of the weaknesses of the static analysis. We tried automatic parallelization on the OmpSCR benchmarks [1], which consist of short mathematical kernel code, by Intel C/C++ compiler (Version 11.0) [9].

Programmer# column shows the number of loops that the programmer manually parallelized. *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. The result shows that only four loops (out of 13) were successfully parallelized. Pointer aliasing was the main cause of the limitations. Finally, *Profiler#*

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

Table I
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

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 the given input. As we will show in Section VI-E, 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 parallelize 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 the inherent input-sensitivity problem.

III. THE BASELINE ALGORITHM: PAIRWISE METHOD

Before presenting SD^3, 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. In this paper, we focus on data dependences within loop nests because loops are major parallelization targets. Our algorithm can be easily extended to find data dependences in arbitrary program structures.

A. Basic Idea of the Pairwise Method

In the big picture, 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 solve (1) nested loops and (2) loop-carried/independent dependences.

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

Figure 3. A simple example of data-dependence profiling.

B. Checking Data Dependences in a Loop Nest

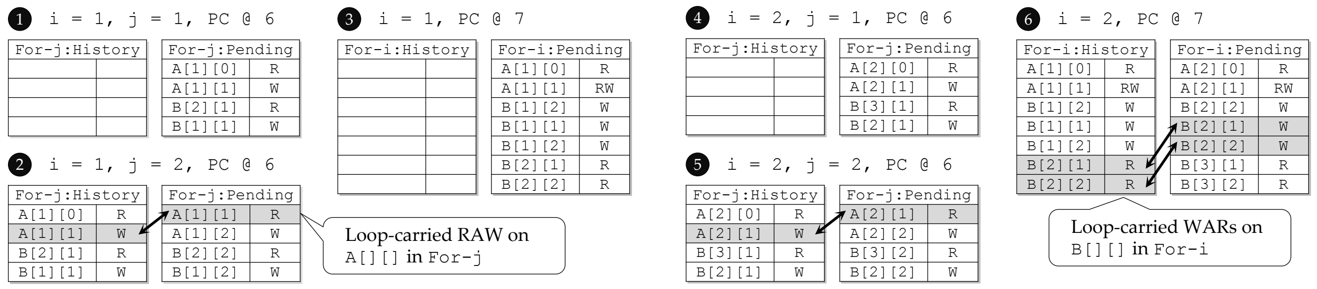
We explain the pairwise algorithm with a simple loop nest example in Figures 3 and 4. Note that we intentionally use a simple example. This code may be easily analyzed by compilers, but the analysis could be difficult if (1) dynamically allocated arrays are passed through deep and complex procedure call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

C. Loop-carried and 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.



For-j:History		For-j:Pending	
		A[1][0]	R
		A[1][1]	W
		B[2][1]	R
		B[1][1]	W

For-j:History		For-j:Pending	
A[1][0]	R	A[1][1]	R
A[1][1]	W	A[1][2]	W
B[2][1]	R	B[2][2]	R
B[1][1]	W	B[1][2]	W

For-i:History		For-i:Pending	
		A[1][0]	R
		A[1][1]	RW
		B[1][2]	W
		B[1][1]	W
		B[1][2]	W
		B[2][1]	R
		B[2][2]	R

For-j:History		For-j:Pending	
		A[2][0]	R
		A[2][1]	W
		B[3][1]	R
		B[2][1]	W

For-j:History		For-j:Pending	
A[2][0]	R	A[2][1]	R
A[2][1]	W	A[2][2]	W
B[3][1]	R	B[3][2]	R
B[2][1]	W	B[2][2]	W

For-i:History		For-i:Pending	
A[1][0]	R	A[2][0]	R
A[1][1]	RW	A[2][1]	RW
B[1][2]	W	B[2][2]	W
B[1][1]	W	B[2][1]	W
B[1][2]	W	B[2][2]	W
B[2][1]	R	B[3][1]	R
B[2][2]	R	B[3][2]	R

Figure 4. Snapshots of the pending and history tables of each iteration when the pairwise method profiles Figure 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;
6:    match();
7:    if (pass_flag == 1)
8:      do_something();
9:    ...

11: void match() {
12:   ...
13:   if (condition)
14:     pass_flag = 1;
15:   ...
16: }
```

Figure 5. Loop-independent dependence on `pass_flag` in 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.

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.

D. 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 III-B, 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 IV, we solve this problem by using *compression*.

Profiling time overhead is also critical since almost all the memory loads and stores are instrumented. We attack this overhead by *parallelizing* the data-dependence profiling itself. We present our solution in Section V.

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

A. Overview of the Algorithm

The basic idea of solving this memory-scalability problem is to store memory references as *compressed* formats. Since many memory references show stride patterns[3], our profiler can also compress memory references with a *stride* format (e.g., `A[a*i + 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,
- 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.

[2]Because `pass_flag` is a global variable, a loop-carried WAW is reported, but it can be removed by privatizing the variable.

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

B. Dynamic Detection of Strides

We define that an address stream is a *stride* if it can be expressed as $base + distance \times i$. 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 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. The address that cannot be represented as part of a stride is called a *point* in this paper.

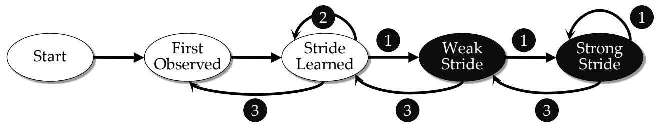


Figure 6. Stride detection FSM. The current state is updated on every memory access with the following additional conditions: ❶ The address can be represented with the learned stride; ❷ Fixed-memory location access is detected; ❸ The address cannot be represented with the current stride.

Figure 6 illustrates the state transitions in the 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.

C. New Dependence Checking Algorithm with Strides

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 using the interval test and then perform a new data-dependence test, *Dynamic-GCD*.

A naive solution for finding overlapping strides would be linear searching between strides, but this is extremely slow. Instead, we employ an *interval tree* based on a balanced binary search tree, Red-Black Tree [5]. Figure 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].

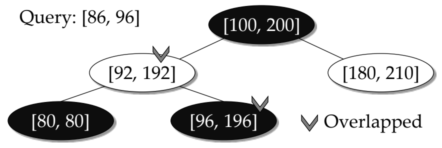


Figure 7. Interval tree (based on a Red-Black Tree) for fast overlapping point/stride searching. Numbers are memory addresses. Black and white nodes represent Red-Black properties.

The next step is an actual data-dependence test between overlapping strides. 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 the 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: }
```

Figure 8. A simple example for Dynamic-GCD.

To illustrate the algorithm, consider the code in Figure 8. We assume that the array `A` is `char[]` type and begins at address 10. 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[4]:

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

The solutions are 24 and 30, which means exactly two memory addresses are conflicting. The detail of Dynamic-GCD can be found at an extended version of this paper [13].

D. Summary of the Memory-Scalable SD^3 Algorithm

The first part of SD^3, a memory-scalable algorithm, is summarized in Algorithm 1. 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 1 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, and (4) the number of total accesses in the stride.
- `LoopInstance`: It represents a dynamic execution state of a loop including statistics and tables for data-dependence calculation (pending and history tables).

[4]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 $ax + by = 1$.

- `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 indicate killed addresses. See Figure 9.
- `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.

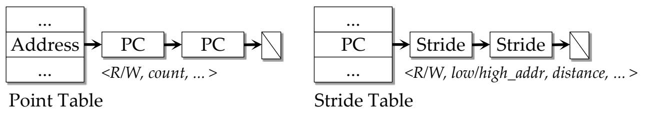


Figure 9. Structures of point and stride tables: The structures of the history and pending tables are identical. They only differs some fields.

In Algorithm 1, new steps added on top of the pairwise algorithm are underlined.

E. 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 1 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, $O(max(size($`HistoryPointTable`$),$ $size($`PendingPointTable`$)))$.[5]

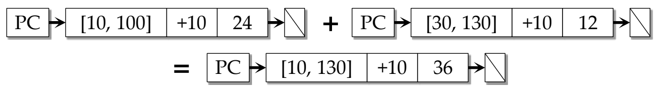


Figure 10. Two stride lists from the same PC are about to be merged. The stride ([10, 100], +10, 24) means a stride of (10, 20, ..., 100) and total of 24 accesses in the stride. These two strides have the same stride distance. Thus, they can be merged, and the number of accesses is summed.

[5]Each entry of a point table has a list of PC, which is required to report PC-wise sources and sinks of dependences. However, merging two PC lists while considering duplicate PCs takes linear time. Instead we have a single *PC-set* per entry along with the global PC-set lookup table [13].

Algorithm 1 THE MEMORY-SCALABLE ALGORITHM

```
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.
```

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 Figure 10.

However, 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.

F. 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 III-C, 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.

Figure 11 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

```
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: }
```

Figure 11. A stride from line 6 can be killed by a point at line 2 and a stride at line 4.

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. 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.

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

A. Overview of the Algorithm

It is well known that the runtime overhead of data-dependence profiling is very high. The main reason is that *all* memory loads and stores except for stack operations (e.g., `push` and `pop`) are instrumented. A typical method to reduce this overhead is 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?

B. A Hybrid Parallelization Model of SD^3

We first survey parallelization models of the profiler that implements Algorithm 1. 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 1. 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 1 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 [20]. 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 1) 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.

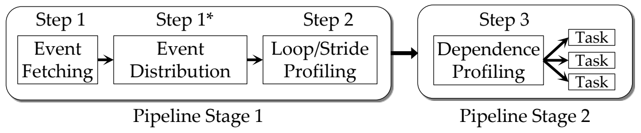


Figure 12. SD^3 exploits both pipelining (2-stage) and data-level parallelism. Step 1* is augmented for the data-level parallelization.

Figure 12 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. We apply the same data-parallel approach, which was used in the pipeline stage 2, on multiple machines.

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 Figure 13. 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.

However, this event distribution, as illustrated in Figure 14, is not a simple division of the entire input events:

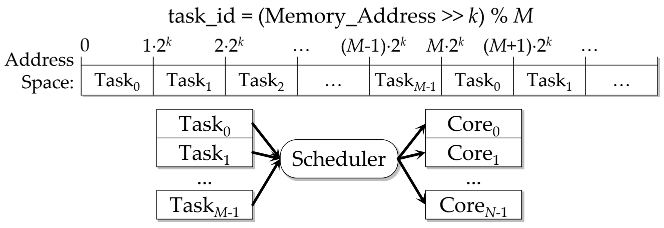


Figure 13. Data-parallel model of SD^3 with the address-range size of 2^k, M tasks, and N cores: Address space is divided in an interleaved way. The above formula is used to determine the corresponding task id for a memory address. In our experimentation, the address-rage size is 128-byte ($k = 7$), and the number of tasks is the same as the number of cores ($M = N$).

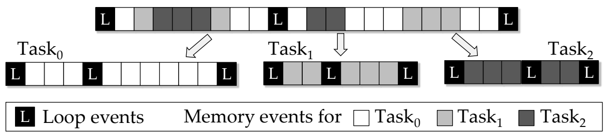


Figure 14. An example of the event distribution step with 3 tasks: Loop events are duplicated for all tasks while memory events are divided depending on the address-range size and the formula of Figure 13.

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 1 are triggered on a loop event.

C. 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 Figure 15. 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.

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.

D. Details of the Data-Parallel Model

A key point in designing the data-parallel model is to obtain higher speedup via good load balancing. However,

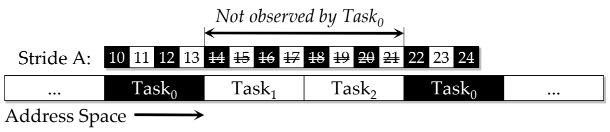


Figure 15. A single stride can be broken by interleaved address ranges. Stride A will be seen as two separate strides in Task$_0$ with the original stride-detection algorithm.

this division of the address space inherently makes a load unbalancing problem because 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 and need to find a reasonably balanced address-range size. According to our experiment, 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 [13].

Even if taking this interleaved approach, we still have a load unbalancing problem. To address this problem, we attempted to generate sufficient tasks and employed the work-stealing scheduler [3]. At a glance, this approach would yield better speedup, but our initial data negated our hypothesis [13]. 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 Figure 15, 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.

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.

VI. Experimental Results

A. Implementation of SD^3

We implement SD^3 on top of Pin [18], a dynamic binary instrumentation framework for x86 and x86-64. Performing data-dependence profiling at binary level rather than via compiler-based instrumentation broadens the applicability of the tool as it can be applied to executable generated by

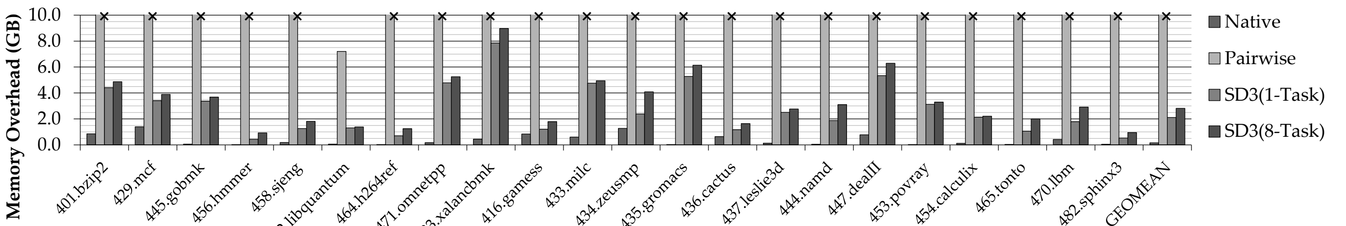


Figure 16. Absolute memory overhead for SPEC 2006 with the reference inputs: 21 out of 22 benchmarks (× mark) need more than 12 GB in the pairwise method that is still the state-of-the-art algorithm of current tools. The benchmarks natively consume 158 MB memory on average.

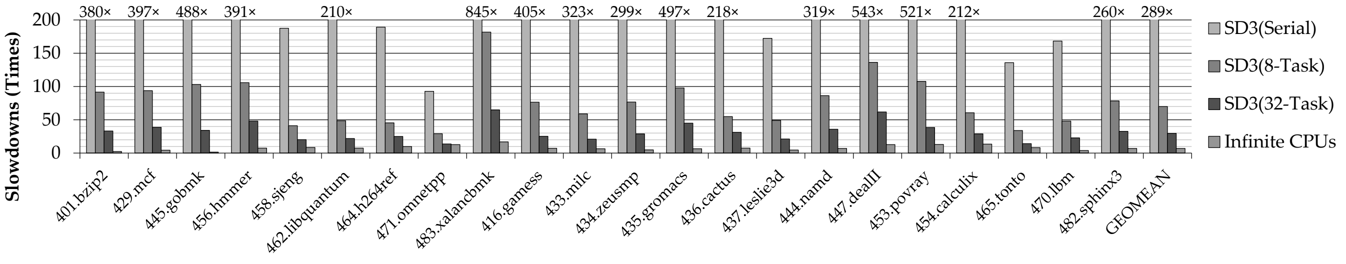


Figure 17. Slowdowns (against the native run) for SPEC 2006 with the reference inputs: From left to right, (1)SD^3 (1-task on 1-core), (2) SD^3 (8-task on 8-core), (3) SD^3 (32-task on 32-core), and (4) estimated slowdowns of infinite CPUs. For all experiments, the address-range size is 128 bytes. The geometric mean of native runtime is 488 seconds on Intel Core i7 3.0 GHz.

different compilers or those that do not have the source code. The downside is that this requires additional static analysis to recover control flows and loop structures from a binary executable, which is not easy to implement.

We use Intel Threading Building Block (TBB) [11] for our parallelization model. We also extend our profiler to work on multiple machines, by a similar way to our data-level parallel model. Note that our profiler must be an *online tool* because generated traces could be extremely huge (10 GB - 10 TB). Our profiler also profiles multithreaded applications and provides per-thread profiling results.

B. Experimentation Methodology

We use 22 SPEC 2006 benchmarks[6] to report runtime overhead by running the *entire* execution of benchmarks with the reference input. 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 II-A. 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.

[6]Due to issues in instrumentation and binary-level analysis, we were not able to run the remaining 6 SPEC benchmarks.

C. Memory Overhead of SD^3

Figure 16 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. 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 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.

D. Time Overhead of SD^3

We now present the time overhead results. As discussed in Section V-D, the number of task is the same as the number of core in the experimentations.[7] The slowdowns are measured against the execution time of native programs. 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.

As shown in Figure 17, 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]Precisely speaking, running 8-task on 8-core is only for the data-parallel part. Our profiler actually requires one more thread for the event fetch, distribution, and stride profiling as shown in Figure 12.

E. Input Sensitivity of Dynamic 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.

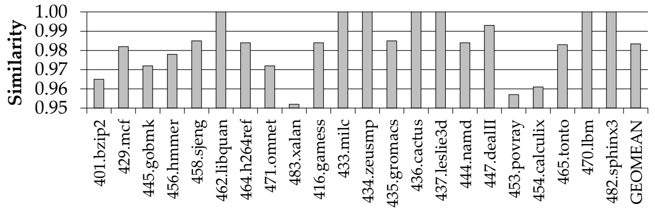


Figure 18. Similarity of the results from different inputs: 1.00 means all results were identical (not the frequencies of dependence pairs).

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 dynamic analysis for streaming applications to find pipeline parallelism with annotated source code [24]. Their results showed that streaming applications had stable and regular data flows.

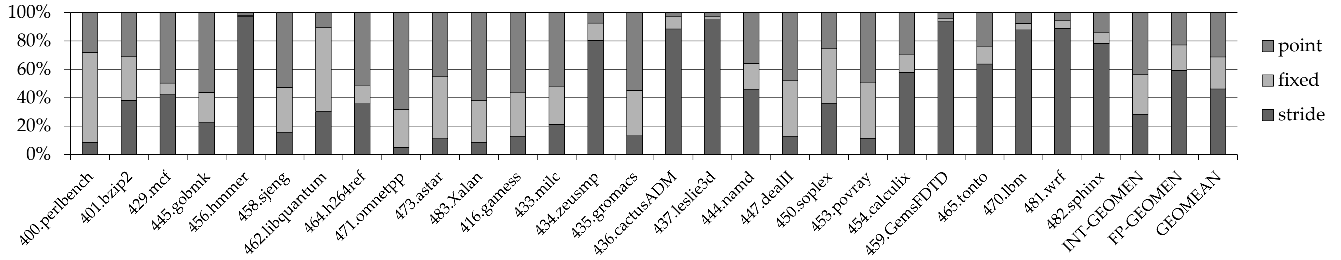


Figure 19. Classification of stride detection for SPEC 2006 benchmarks with the train inputs: (1) point: memory references with non-stride behavior, (2) fixed: memory references whose stride distance is zero, and (3) stride: memory references that can be expressed in an affined descriptor.

F. Discussion

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

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. Figure 19 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.

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 (there is a dependence, but reported as none) or false positives (there is no dependence, but reported as having one). In some usage models of data-dependence profiling, inaccuracy can be tolerated. One such example is thread-level data speculation (TLDS) [4, 17, 23]. 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. We have found examples where a simple sampling would make a wrong decision on parallelizability prediction [13]. We currently do not use any sampling techniques in our SD^3.

VII. Related Work

A. Dynamic Data-Dependence Analysis

One of the early works that used data-dependence profiling to help parallelization is Larus' parallelism analyzer `pp` [15]. `pp` detects loop-carried dependences in a program with a particular input. 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. [25] 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. [31] proposed a data-dependence-distance profiler called Alchemist. Their tool is specifically designed for the *future* language constructor that represents the result 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. [27] 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.

B. Dependence Profiling for Speculative Parallelization

As discussed in Section VI-F, 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, 7, 17, 23, 29], 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.

C. Reducing Overhead of Dynamic Analysis

Shadow Profiling [20], SuperPin [28], PiPA [32], and Ha et al. [8] 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 [16, 19, 22, 30] 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 [19], 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.

VIII. 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 [12]. We hope that SD^3 can help many researchers to develop other dynamic tools to assist parallel programming.

IX. Acknowledgments

Special thanks to Nagesh B. Lakshminarayana for helping earlier experimentation. We thank Geoffrey Lowney, Robert Cohn, and John Pieper for their feedback and support; Paul Stravers for providing an experimental result; Puyan Lotfi, Pranith D. Kumar, and HParch members for their comments; finally, many valuable comments from the anonymous reviewers who help this paper. Minjang Kim and Hyesoon Kim are supported by NSF CCF0903447, NSF/SRC task 1981, Intel Corporation, and Microsoft Research.

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] 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.

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

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

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

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

[12] 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.

[13] M. Kim, H. Kim, and C.-K. Luk. SD^3: A scalable approach to dynamic data-dependence profiling. Technical Report TR-2010-001, Atlanta, GA, USA, 2010.

[14] 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.

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

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

[17] 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.

[18] 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.

[19] 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.

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

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

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

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

[24] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *MICRO-40*, 2007.

[25] 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.

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

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

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

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

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

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

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