SD^3: An Efficient Dynamic Data-Dependence Profiling Mechanism

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

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 program analysis technique to exploit parallelism in serial programs. More specifically, manual, semi-automatic, or automatic parallelization can use the outcomes of data-dependence profiling to guide *where* and *how* to parallelize in a program.

However, state-of-the-art data-dependence profiling techniques consume extremely huge resources 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 with a typical resource budget or only report very limited information.

In this paper, we propose an efficient approach to data-dependence profiling that can address 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 22 SPEC 2006 benchmarks 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.

We also demonstrate the usefulness of SD^3 by showing manual parallelization followed by data dependence profiling results.

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 on supporting parallel programming. For example, automatic parallelization has been researched for decades, but it was successful in only limited domains. Parallelization is still a burden for programmers.

Recently, several tools, including Intel Parallel Advisor [12] and Vector Fabric Pareon (previously, 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 whether two tasks access the same memory location and at least one of them is a write. Two data-independent tasks can be safely executed in parallel without synchronization.

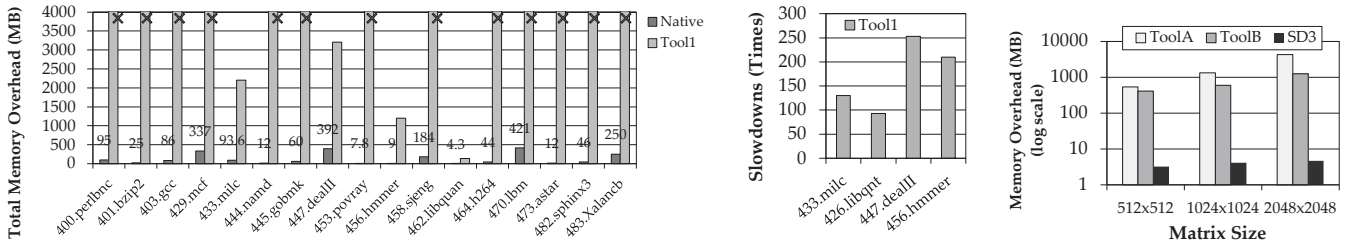
Traditionally, data-dependence analysis has been done *statically* by compilers techniques, such as the GCD test [24] and Banerjee's inequality test [16], especially for array-based data accesses. This static analysis is limited by pointer analysis in languages with arbitrary pointers and dynamic allocations such as C/C++. However, precise pointer analysis is undecidable for these languages [4]. We observed that state-of-the-art production-automatic parallelizing compilers often fail to parallelize simple, embarrassingly parallel loops written in C/C++. The compilers also had limited success in irregular data structures due to pointer analysis 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-only approaches since all memory addresses are resolved in runtime. Data-dependence profiling has already been used in parallelization efforts like speculative multithreading [5, 8, 20, 27, 34] and finding potential parallelism [9, 17, 25, 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 a number of research works have focused on using data-dependence profiling, almost no work exists on addressing the performance and overhead issues.

As a concrete example, Fig. 1 shows the dependence profiling overhead of the two commercial tools, obtained on May 2010. Because no detailed profiling algorithms of the tools are publicly available, we implement our own baseline algorithm called *the pairwise method*. We believe the pairwise method is very similar to the algorithms of these commercialized tools.

Fig. 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 bench-

• *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 addition

Fig. 1: Overhead of the current commercial dependence profilers. "Tool1," "ToolA," and "ToolB" are used to anonymize commercial tools. We profiled the top 20 hottest loops and their inner loops (if any) in the experimentations of (a) and (b).

marks, only four benchmarks were successfully analyzed; the rest of the benchmarks failed because of insufficient physical memory. The runtime overhead is between an 80× and 270× slowdown for the four benchmarks that worked. While both time and memory overhead are severe, the latter will stop further analysis. The culprit is the data-dependence profiling algorithms used in these tools. The 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 overhead problem is a simple matrix addition program that allocates three N × N matrices for $A = B + C$. As shown in Fig. 1(c), the current tools require an order of gigabytes of additional memory as the matrix size increases. In contrast, our method, SD^3, needs only less than 10 MB memory.

In this paper, we address these memory and time overhead problems by proposing an efficient 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 a new data-dependence calculation algorithm:* We demonstrate that SD^3 significantly reduces the memory consumption of data-dependence profiling. SD^3 is not a simple compression technique; we should address several issues to achieve memory-efficient profiling. The failed benchmarks in Fig. 1(a) are successfully profiled by SD^3 on a 12 GB machine.
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

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 it with a representative

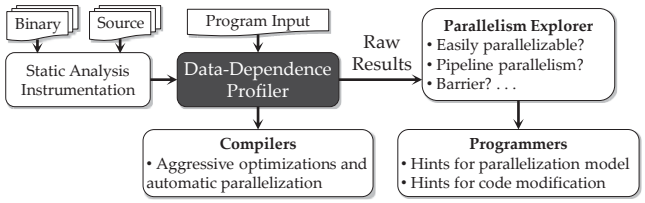


Fig. 2: Examples of data-dependence profiler applications.

input. A raw result from our dependence profiler is a list of discovered data-dependence pairs. 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),
- **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 the transformation of the serial code. 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 overhead problem, and we focus on this in the paper.

3 The Baseline Pairwise Method

We describe our baseline algorithm, *the pairwise method*. 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 description of the algorithm by focusing on data dependences within *loop nests* because loops are major parallelization targets. Note that the previous algorithms [5, 17] may be similar to the pairwise method, but they did not present a solid baseline algorithm for SD^3.

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 the *history table*, respectively. Each loop has its own pending and history tables 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 explained the pairwise algorithm with a loop example in [15].

The pairwise algorithm handles a loop across function calls and recursion via the loop stack (See Section 4.3). It also easily finds dependences between functions. When a function starts, we assume that a loop, which encompasses the whole function body with zero trip count, has been started, such as `do {func_body();} while(0);`. Then, loop-independent dependences in this imaginary loop will be dependences in the function.

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 for judging the parallelizability of a loop. While loop-independent dependences do not prevent parallelizing a loop by `DOALL`, loop-carried flow dependences generally prohibit parallelization except for `DOACROSS` or pipelining.

To handle loop-independent dependence, we introduce a *killed* address, which is very similar to the *kill* set in data-flow analysis. We mark an address as killed once the memory address is written in an iteration. Then, subsequent accesses to the killed address within the same iteration are no longer stored in tables and reported as loop-independent dependences. We provide an example from SPEC 179.art in [15].

3.3 Problems of the Pairwise Method

The pairwise method needs to store all distinct memory references within a loop invocation. As a result, the memory requirement per loop is obviously increased as the memory footprint is increased. The memory requirement could be even worse because of nested loops. In the detailed description of the pairwise method [15], one of the important steps is *propagation*. For example, the history references of inner loops propagate to their upper loops, which is implemented as merging history and pending tables. Hence, only when the top-most loop finishes can all the history references within the loop nest be flushed. 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. In turn, whole distinct memory references often need to be stored along with PC addresses throughout the program execution. In Section 4, we solve this problem using *compression*.

Profiling time overhead is also critical since an extreme number of 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 A Memory-Efficient Algorithm of SD^3

The basic idea of solving the memory overhead problem is to store memory references as a *compressed* format. Since many memory references show stride patterns, 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 an efficient data-dependence profiler. We need to address the following challenges for SD^3:

- How to detect stride patterns dynamically (4.1),
- How to perform data-dependence checking with the compressed format *without* decompression (4.2),
- How to check dependences efficiently with both stride and non-stride patterns (4.4), and
- How to handle loop nests and loop-independent dependence with the compressed format (4.5, 4.6).

4.1 Dynamic Detection of Strides

We define an address stream as a *stride* as long as the stream can be expressed as $base + stride\_distance \cdot n$. 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 or not the access is part of a stride. Because the 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. 3: 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 (stride); ❷ The address cannot be represented with the current stride (point).

Fig. 3 illustrates that the state transitions in our stride 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. When a row changes, however, there could be an irregular jump in the memory address, breaking the learned stride. Having Weak/StrongStride states tolerates a few non-stride accesses so that no point references are recorded in the table.

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. 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 a stride $10 + 4 \cdot n$ ($0 \leq n \leq 4$). Note that such non-strict strides may cause slight errors when calculating the occurrence count of data dependences (See Section 4.7). We do not further combine multiple strides from a PC into a single stride even if these strides are generated by two- (or more) dimensional accesses.

4.2 Stride-Based Dependence Checking Algorithm

Checking dependences is trivial in the pairwise method: we exploit a hash table keyed by memory addresses, which enables fast searching whether or not a given memory address is dependent. Unfortunately, the stride-based algorithm cannot use such simple dependence checking because a stride represents an *interval*. We also need to efficiently find dependence among both strides and points. This section first introduces algorithms to find conflicts within two strides (or a stride and a point). Section 4.4 then discusses how SD^3 implements efficient stride-based dependence checking.

The key point in the new algorithm is to find conflicts of two strides. We attack the problem through two steps: (1) finding overlapped strides and points and (2) performing a new data-dependence test, DYNAMIC-GCD, to calculate the exact conflicts. For the first step, the overlapping test, we employ an *interval tree*, which is based on the Red-Black Tree [6]. The test finds *all* overlapping strides and points in a tree for a given input.[1] Fig. 4 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].

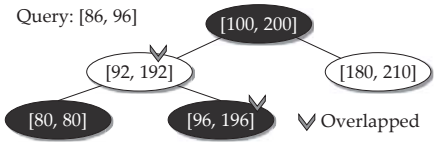


Fig. 4: Interval tree (based on a Red-Black Tree) for fast overlapping point/stride searching. Numbers are memory addresses. Black and white nodes represent Red-Black properties.

The next step is an actual data-dependence test between two 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: }
```

Fig. 5: A simple example for DYNAMIC-GCD.

To illustrate the algorithm, consider the contrived program in Fig. 5. We assume that array `A` is a 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. DYNAMIC-GCD returns the exact number of conflicting addresses in the two strides. The problem is reduced to solving a Diophantine equation:[2]

$$2x + 20 = 3y + 21 \quad (0 \leq x,\ y \leq 6). \tag{1}$$

DYNAMIC-GCD, described in Algorithm 1, solves this equation. We detail the computation steps using Fig. 6:

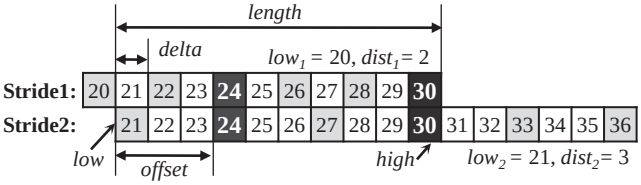


Fig. 6: Two strides in Fig. 5. Lightly shaded boxes indicate accessed locations, and black boxes are conflicting locations. The terms (length, delta, low, high, and offset) are explained in the following paragraphs.

1) Sort and obtain the overlapped bounds and lengths: Let $low_1 \leq low_2$; otherwise swap the strides. In Fig. 6, the bounds are $low = 21$, $high = 30$, and $length = 10$.
2) Check the existence of the dependence by the GCD test: We only have the runtime stride information. To use the GCD test, we transform the strides as if we have the common array base such as A[$dist_1 \cdot x + delta$] and A[$dist_2 \cdot y$], where $delta$ is the distance between low and the immediately following accessed address in Stride1. Then, we can use the GCD test. In Fig. 6, $delta$ is 1; the GCD of 2 and 3 is 1, which divides $delta$. Therefore, the strides *may* be dependent.
3) Count the exact dependences within the bound: To do so, we first compute the smallest conflicting point in the bound (24 in Fig. 6) by using EXTENDED-EUCLID [6], which returns x and y in $ax + by = \gcd(a, b)$, where a is $-dist_1$, and b is $dist_2$. $offset$ is then defined as the distance between low and this smallest conflicting point, which is 3 in Fig. 6. Observe that the difference between two adjacent conflicting points is the least common multiple of $dist_1$ and $dist_2$ (6 in Fig. 6). Then, we can count the exact number of dependences: two addresses (24 and 30) are conflicting. The strides are dependent.

4.2.1 *Clarification of Equation* (1) *and Figure 5*

We discussed DYNAMIC-GCD with the code of Fig. 5. However, this code does *not* actually create the two strides as shown in Fig. 6 and Eq. (1). Because the loop is single-level, the code will only check dependences between two pending points (the write at line 2 and the read at line 3) against the history strides as the loop iterates. On `i`-th iteration (assuming zero-based index), the two points, $2i + 20$ (the write at line 2) and $3i + 21$ (the read at line 3), are being checked with

1. In the worst case, where all the nodes of an interval tree intersect an input, linear time is required. On average, an interval tree is still faster than a simple linear search. Section 4.4 introduces a further optimization.

2. 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 $ax + by = 1$.

Algorithm 1 DYNAMIC-GCD

Inputs: Two stride: $(low_1, high_1, dist_1)$, $(low_2, high_2, dist_2)$
Output: Return the number of dependences of the two strides.

Require: $low_1 \leq low_2$; otherwise swap the strides.
1: Calculate low, $high$, and $length$ as shown in Fig. 6.
2: $delta \leftarrow (dist_1 - ((low - low_1) \mod dist_1)) \mod dist_1$
3: $gcd \leftarrow$ The greatest common divisor of $dist_1$ and $dist_2$
4: **if** $(delta \mod gcd) \neq 0$ **then**
5: **return** 0
6: **end if**
7: $x, y \leftarrow$ EXTENDED-EUCLID$(-dist_1, dist_2)$
8: $lcm \leftarrow$ The least common multiple of $dist_1$ and $dist_2$
9: $offset \leftarrow ((dist_2 \cdot y \cdot delta/gcd) + lcm) \mod lcm$
10: $result \leftarrow (length - (offset + 1) + lcm)/lcm$
11: **return** $\max(0, result)$

the two history strides, $2k + 20$ and $3k + 21$, $(0 \leq k < i)$.[3] Therefore, there is no moment when Eq. (1) is performed. We intentionally used an incorrect code example to explain DYNAMIC-GCD easier.

```
1: for (int i = 0; i <= 7; ++i) {
2:   A[2*i + 10] = ...;   // Stride 1 (Write)
3:   for (int j = 0; j <= 6; ++j) {
4:     ... = A[3*j + 11]; // Stride 2 (Read)
5:   }
6: }
```

Fig. 7: A correct program that will shows Fig. 6 and Equation (1).

Fig. 7 contains the correct code that will exactly show the case of Fig. 6 and Eq. (1). When `i` is 7 and the inner loop just finishes, the history stride table of `for-j` is now propagated to the pending stride table of `for-i`. After the propagation and when the current iteration (`i` = 7) finishes, we will finally see the dependence checking depicted in Fig. 6.

4.3 Overview of the Memory-Efficient SD^3 Algorithm

The first part of SD^3, a memory-efficient algorithm, is presented in Algorithm 2. The algorithm augments the pairwise algorithm and will be parallelized to decrease time overhead. We still use the pairwise algorithm for memory references that do not have stride patterns. Algorithm 2 uses the following data structures, which are depicted in Fig. 8:

• **`POINT`**: This represents a non-stride memory access from a PC. This structure has (1) PC address (or location ID); (2) the number of total accesses from this PC; (3) the pointer to the next `POINT` (if any), which is needed because a memory address can be touched by different PCs; and (4) miscellaneous information including the read/write mode and the last accessed iteration number to calculate dependence distance. Section 6.2.3 introduces optimizations to reduce the overhead of handling the list structure.

• **`STRIDE`**: This represents a compressed stream of memory addresses from a PC. This structure has (1) the lowest and highest addresses; (2) the stride distance; (3) the number of total accesses in this stride; and (4) the pointer to the next `STRIDE` because a PC can create multiple strides that cannot be combined. Notice that miscellaneous fields such as RW mode and the last accessed iteration number are stored along with the PC because these fields are common per PC.

• **`LoopInstance`**: This represents a dynamic execution state of a loop, including statistics and data structures for data-dependence calculation (the pending and history tables).

• **`Pending{Point|Stride}Table`**: These tables capture memory references in the *current* iteration of a loop. `PendingPointTable` is a hash table, where the key is memory address for fast conflict checking, and the value is `POINT`. `PendingStrideTable` associates `STRIDE` with the originating PC address. Both pending tables have *killed bits* to handle loop-independent dependences. These tables employ *DAS-ID* (dynamic allocation-site ID) optimization to minimize the search space in dependence checking. The necessary structure change is discussed in the following section.

• **`History{Point|Stride}Table`**: This holds memory accesses in *all* executed iterations of a loop so far. The structure equals the pending tables except for killed bits.

• **`LoopStack`**: This 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 in loop nests that may have function calls and recursion.

Although the big picture of the algorithm is described, the stride-based algorithm also needs to address several challenges. The following four subsections elaborate these issues.

Algorithm 2 THE MEMORY-EFFICIENT 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 reference, 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 (Section 4.1). If R is a write, set its killed bit.
3: At the end of the iteration, do the stride-based dependence checking (Sections 4.2 and 4.4). Report any found dependences.
4: After Step 3, merge the pending and history point tables. Also, merge the stride tables (Section 4.5). Finally, the pending tables, including killed bits, are flushed.
5: When L terminates, flush the history tables, and pop the `LoopStack`. To handle loop nests, we propagate the history tables of L to the parent of L, if they 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 (Section 4.6), this killed history is not propagated.

4.4 Optimizing Stride-Based Dependence Checking

Fig. 8 summarizes the table structures and stride-based dependence-checking algorithm. When the current iteration

3. In our implementation, the very first few points until learning a stride still exist in the point table. Hence, strictly speaking, there are a few history points. However, we can safely assume that no such transient point exists to explain the DYNAMIC-GCD.

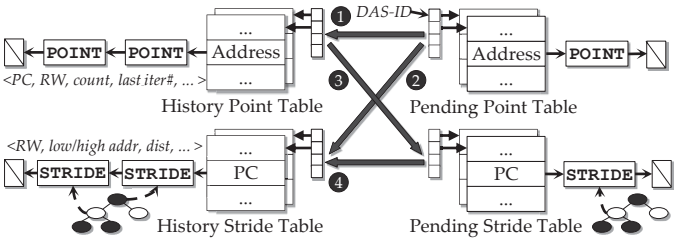


Fig. 8: Structures of the point and stride tables and four cases of the dependence checking: This figure shows the moment when the current iteration finishes. A stride table has an associated interval tree. The tables support DAS-ID (dynamic allocation-site ID) to minimize the checking space. Our implementation uses an additional hash table, where the key is DAS-ID and the value is either point or stride sub-table that only contains memory references from the same DAS-ID.

finishes, the two pending tables are checked against the two history tables. Because it has both point and stride tables, the dependence checking now requires four sub-steps for every pair of the tables, shown as the four large arrows in the figure.

Arrow ❶ is the case of the pairwise method. Arrows ❷ and ❸ show the case of checking a stride table and a point table. We take *every* address in the point table and check the conflict against the stride table using the associated interval tree. Arrow ❹ is the most complex step. *Every* stride in the pending stride table needs to be enumerated and checked against the history stride table. This enumerating and checking could take a long time, especially for a deep nested loop. Therefore, in order to reduce this enumeration and potentially huge search space, we introduce *dynamic allocation-site* optimization.

This optimization is based on the fact that a memory access on a variable or a structure must have its associated allocation site. Memory accesses from different allocation sites will *never* conflict in a correct program. Once allocation sites are known, we need to check only dependences among memory accesses within the same allocation site, reducing search space significantly. In particular, we focus on heap accesses because they are the main target of the analysis. To obtain allocation site IDs on heap accesses effectively, we *dynamically* track allocated heap regions and issue an ID on each heap region. The DAS-ID optimization is implemented as follows:

1) Instrument heap functions (e.g., `malloc` and `delete`).
2) On heap allocation, retrieve the allocated memory range, and issue a DAS-ID by an simply increasing counter. Store this pair of the allocated range and the DAS-ID into a global table. The table is an interval tree that allows a fast query of the associated DAS-ID for a given memory access.
3) On heap deallocation, delete the corresponding node.
4) On load and store, fetch the address, and query the table to obtain the corresponding DAS-ID of the access. Store the memory reference to either a point or stride table reserved for this DAS-ID only.

We finally update the point and stride table structure by adding an additional hash table layer, where the key type is DAS-ID and the value is either a point or a stride table. The valued point and stride tables now only contain memory references from the same DAS-ID.

4.5 Merging Stride Tables for Loop Nests

In the pairwise method, we propagate the histories of inner loops to their upper loops to compute dependences in loop nests. 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), merging tables is straightforward: we simply compute the union set of the two point hash tables.[4]

On the other hand, merging two stride tables is not trivial. A naive solution is to just concatenate two stride lists. If this is done, the number of strides could be bloated, resulting in increased memory consumption. Alternatively, we try to do *stride-level* merging rather than a simple stride-list concatenation. An example is illustrated in Fig. 9.

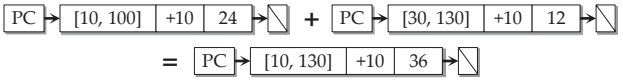


Fig. 9: Two stride lists from the same PC are about to be merged. The stride ([10, 100], +10, 24) means a stride of (10, 20, ..., 100) and 24 total accesses in the stride. These two strides have the same stride distance. Thus, they can be merged, and the number of accesses is summed.

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 in history counters per PC. 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 discussed that maintaining *killed* addresses is very important to distinguish loop-carried and independent dependences. The pairwise method prevented killed addresses from being propagated to further steps (to the next iteration). This step becomes complicated with strides because strides could be killed by the parent loop's strides or points.

```
1: for (int i = 0; i < N; ++i) {    // Loop_1
2:   A[rand() % N] = 10;            // Random kill
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. 10: A stride at line 6 can be killed by either a point at line 2 or a stride at line 4.

Fig. 10 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

4. The point table merging must perform the union of two PC-lists, each of which is from the pending tables and from history tables, respectively. To make PC-list merging faster, we employ PC-set optimization (Section 6.2.3).

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 between strides and points. We use the same dependence-checking algorithm.

Interestingly, after processing killed addresses, a stride could be one of three cases: (1) a shrunk stride (the range of stride addresses 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 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, which means SD^3 results are identical to the pairwise method results.

Section 4.1 discussed that a stride like [10, 14, 18, 14, 18, 22, 18, 22, 26] is also considered a stride in our implementation. In this case, our stride format cannot perfectly record the original characteristic of the stream. We only remember two facts: (1) a stride of $10 + 4 \cdot n, (0 \leq n \leq 4)$ and (2) the total number of memory accesses in this stride is 9. The stride format cannot precisely remember the occurrence count of each memory address. Such lossy compression may cause slight errors when DYNAMIC-GCD calculates.

Suppose that this stride has a conflict at address 26. Address 26 is accessed only one time, but this information has been lost. For the compensation, we add a correction on the result of DYNAMIC-GCD by taking the average occurrence count of each reference: $\lceil 9/5 \rceil = 2$, the total accesses in the stride 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.

5 An Time Efficient Algorithm of SD^3

5.1 Overview of the Algorithm

The time overhead of data-dependence profiling is very high. A typical method to reduce the time overhead would be to use sampling techniques. Unfortunately, simple sampling techniques are not desirable because they mostly trade off accurate results (for a given input) for 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 attack 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 Parallelization Model of SD^3: A Hybrid Approach

We first survey parallelization models of the profiler that implements Algorithm 2. Before the discussion, we need to explain the structure of our profiler briefly. 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. Our profiler is an online tool. Events are 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.

To find an optimal parallelization model for SD^3, three parallelization strategies are considered: (1) task-parallel, (2) pipeline, and (3) data-parallel. Our approach is using 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 [23]. This approach is not easily applicable to the data-dependence profiling algorithm because it requires severe synchronization between tasks due to nested loops. We do not take this approach.

Pipelining enables each step to be executed on a different core in parallel. We have three steps, but the third step, the data-dependence profiling, is the most time-consuming step. Although the third step determines the overall speedup, we still can hide computation latencies of the first (event fetch) and the second (stride detection) steps from pipelining.

Regarding the data-parallel method, first notice that SD^3 itself is embarrassingly parallel. Checking data dependences for a particular address requires only information on this address; no information from the other addresses is needed. We take a SPMD (Single Program Multiple Data) style to exploit this data-level parallelism. A set of *task* perform Algorithm 2 concurrently, but each task only processes a *subset* of the entire input. This data-parallel method is the most scalable one and does not require any synchronizations except for the trivial final result reduction step. We also use this model.

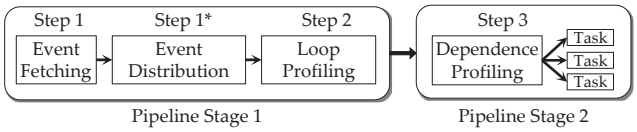


Fig. 11: SD^3 exploits both pipelining (2-stage) and data-level parallelism. Step 1* is augmented for the data-level parallelization.

From this survey, our solution is a hybrid model: we basically exploit pipelining, but the dependence profiling step, which is the longest, is further parallelized by a SPMD style. Fig. 11 summarizes the parallelization model of SD^3. To obtain even higher speedup, we may exploit *multiple* machines (See details in Section 6.2). However, there are several issues for an efficient parallelization, which will be now discussed.

5.3 Event Distribution for Parallel Processing

The *event distribution* step is introduced in stage 1. For the SPMD parallelization at stage 2, we need to prepare inputs

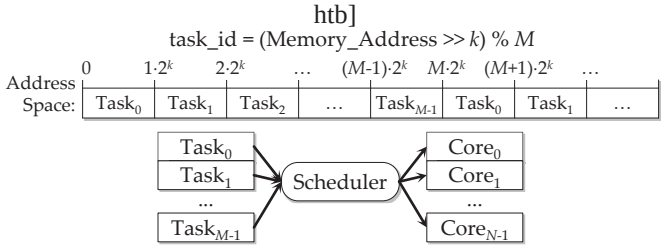


Fig. 12: Data-parallel model of SD^3 with the address-range size of 2^k, M tasks, and N cores: Address space is divided in an interleaved manner. The above formula is used to determine the corresponding task id for a memory address. In our experimentation, the address-range size is 128 bytes ($k = 7$), and the number of tasks is the same as the number of cores ($M = N$).

for each task. In particular, we divide the address space in an *interleaved* fashion for better speedup, as shown in Fig. 12.

The entire address space is divided every 2^k bytes, and each subset is mapped to M tasks in an interleaved manner. Each task analyzes only the memory references from its own range. A thread scheduler then executes M tasks on N cores.

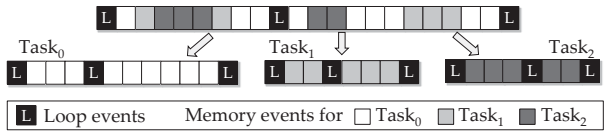


Fig. 13: An example of the event distribution step with three tasks: Loop events are duplicated for all tasks, while memory events are divided depending on the address-range size and the formula of Fig. 12.

The event distribution, as depicted in Fig. 13, is not a simple division of the entire input events: Memory events are distributed by the interleaving. By contrast, loop events must be duplicated for the correctness of the parallelized SD^3 (i.e., the result should be identical to the serial SD^3) because the steps of Algorithm 2 are triggered on a loop event.

5.4 Strides in Parallelized SD^3

Our stride-detection algorithm and DYNAMIC-GCD need a revision in the parallelized SD^3. Stride patterns are detected by observing a stream of memory addresses. In the parallelized SD^3, however, each task can only observe memory addresses in its own address range. This problem is illustrated in Fig. 14.

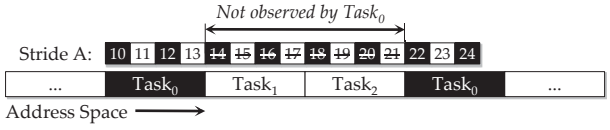


Fig. 14: A single stride can be broken by interleaved address ranges. Stride A will be seen as two separate strides in Task$_0$ with the original stride-detection algorithm.

Here, the address space is divided for three tasks, with a range size of 4 bytes. Consider a stride *A* with a range of [10, 24] and a stride distance of 2. 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 fragmented strides dramatically bloat the number of strides.

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 fragmented 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.

5.5 Details of the Data-Parallel Model

5.5.1 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 creates a load imbalance problem, as memory accesses often show non-uniform locality. Obviously, having too small or too large an address range would worsen this problem. Hence, we use an interleaved division as discussed and then need to find a reasonably balanced address-range size. According to our experiment (not shown in the paper), as long as the range size is not too small or too large, address-range sizes from 64 to 256 bytes yield well-balanced workload distribution. In our implementation, we choose 128 bytes.

5.5.2 Choosing an optimal number of tasks

Even when taking the interleaved approach, we cannot avoid the load imbalance problem. To address this issue, 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 (not shown in the paper). We observed that no speedup was gained using this approach.

There are two reasons for this: First, even if the number of the memory events is reduced, the number of strides may not be proportionally reduced. For example, in Fig. 14, despite the revised stride-detection algorithm, the total number of strides for all tasks is three; on a serial version of SD^3, the number of strides would have been one. Hence, having more tasks may increase the overhead of storing and handling strides, eventually resulting in poor speedup. Second, the overhead of the event distribution would be significant as the number of tasks increases. 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 events 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 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 build SD^3 using both Pin [21], a dynamic binary-level instrumentation toolkit, and LLVM [19], a compiler framework. We discuss the motivation for using Pin and LLVM and several important issues.

6.1 Basic Architecture

Our profiler consists of a *tracer* and an *analyzer*, a typical producer and consumer architecture:

- Tracer: This instruments a program, captures runtime execution traces (i.e., memory events and loop events), and transfers the traces to the analyzer via shared memory. A tracer is built on either Pin or LLVM.
- Analyzer: This takes events from the tracer and performs the SD^3 algorithm, which is orthogonal to tracers.

Our profiler must be an *online tool*. Because the majority of loads and stores could be instrumented, generated traces could be extremely large, up to an order of 10 TB. We cannot simply use an offline approach with such traces. An example of such an offline approach would be storing and compressing events (e.g., using bzip) and then decompressing and analyzing the events. This approach is not effective at all because compressing/decompressing traces takes most of the time. One concern of this online approach would be the overhead of inter-process communication. We observed that the average event transfer rate between the two processes was approximately 1 - 3 GB/s, which can be sufficiently handled by modern computers. The size of the execution event is 12 bytes (on x86-64), and events are transferred without compression.

This separation of tracer and analyzer results in two significant benefits. First, pipeline parallelism, explained in Section 5.2, is easily achieved. Second, the analyzer can 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 instrumentation toolkits.

6.2 Implementation of Analyzer

The analyzer first implements the data structures described in Section 4.3 and Algorithm 2, and we then parallelize using Intel Threading Building Block (TBB) [13]. To obtain even better parallelism, we extend our profiler to work on *multiple machines*, using an MPI-like execution model [10]. The same tracer, analyzer, and application are running in parallel on multiple machines, but each machine has an equally divided workload. This is a simple extension of our data-parallel model but applies across different machines.

6.2.1 Importance of programming techniques

Many programming techniques are extremely essential to improve the performance of the pairwise and SD^3 significantly, other than the key algorithms. Specialized data structures should be implemented rather than using general data structures in C++ STL. Customized memory allocation is also critical because the memory reference structures (`POINT` and `STRIDE`) are frequently allocated and removed.

6.2.2 False Positive and False Negative Issues

For the implementation of the analyzer, we should consider the false positive (reported as having dependences, but it was a false alarm) and false negative (no dependences reported, but it has a dependence) issues in data-dependence profiling.

False negatives can occur when not all code can be executed with a specific input. Section 7.4 discusses this problem. False positives can also occur if we take a larger granularity in the memory instruction instrumentation, such as 8-byte or cache-line granularity rather than a byte granularity. Data-dependence profiling in the speculative multithreading domain can use a large granularity to minimize overhead, but this approach suffers from more false positive dependences [5]. In our implementation, the stride-based approach does not suffer from false positives. We correctly handle the size of the 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 are keyed by addresses, we always use 1-byte granularity, which does not suffer from any false positives. False negatives, however, can occur in a very unusual case, shown in Fig. 15.

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

Fig. 15: A false negative case with 1-byte granularity: both `pd` and `pc` are the alias of `raw`, but their access types are different.

Even if there is an 8-byte write at line 4, the 1-byte granularity policy records only the first byte of the access. The read from line 5 results in a missing data dependence. We believe such a case is very unlikely to occur in well-written code. This problem can be resolved by our DAS-ID optimization. The heap accesses at line 4 and 5 both have the same allocation site at line 1. We then easily detect aliased accesses by the different access types.

6.2.3 PC-set optimization for the Pairwise Method

To reduce both time and space overhead in the pairwise method, we introduce PC-set optimization. Recall that SD^3 still uses the pairwise method when a memory access does not show 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.

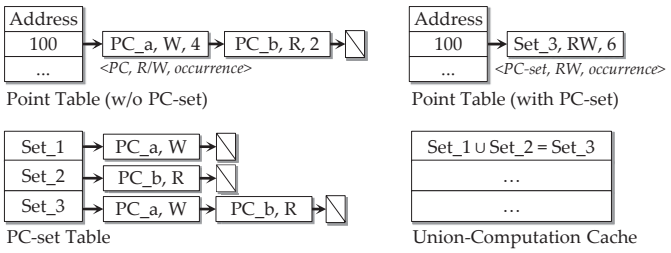


Fig. 16: PC-set optimization for fast PC-list merging. A PC list can be represented as a single PC-set ID, resulting in saving the memory. PC-list merging can also be accelerated by a cache.

The same memory address may be accessed by multiple PC locations. Since we report PC-wise sources and sinks of dependences, each entry of a point table must have a *PC list*, which was implemented as a list of `POINT`. Having a PC list creates two challenges: (1) space overhead and (2) overhead on merging two PC lists (computing a union of two lists). Fortunately, we observed that the number of distinct PC lists for SPEC 2006 was not significant: the geometric mean is only 6,242. We also learned that most of the union computation was repeated.

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. 16, not only save the total memory consumption, but also avoid excessive computation time. The hit ratio of the union cache is 95% on average.

This PC-set optimization causes another lossy compression like the error discussed in Section 4.7. As illustrated in Fig. 16, introducing a 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 a minor error. Again, no error exists on judging the existence of dependences.

6.3 Implementation of Tracers

We first implemented SD^3 on Pin [15]. We discuss challenges for Pin-based SD^3 and the motivations for LLVM-based SD^3.

6.3.1 Issues in a Pin-based Tracer

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

The downside of the Pin-based approach is that additional binary-level static analysis is needed to recover control flow graphs and loop structures, which is difficult to implement. For example, recovering indirect branches (e.g., jump tables for switch-case) and pinpointing the correct locations of loop entries and exits are challenges in binary-level analysis.

Regarding the instrumentation of loads and stores, an x86 binary executable typically has a lot of artifacts from push/pop in stacks and system function calls. Without eliminating such redundant loads and stores, results of a Pin-based profiler would have a lot 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. Filtering such loads and stores selectively is also difficult. These challenges motivate the use of an LLVM-based SD^3.

6.3.2 Issues in an LLVM-based Tracer

Using compiler-based instrumentation such as LLVM may address the issues in the Pin-based tracer. LLVM provides a very rich static-analysis infrastructure, including correct control flows and loop structures. Furthermore, LLVM solves many challenges in binary-level instrumentation. For example, skipping inductions and reductions is relatively easy to implement since all the data-flow information is retained, unlike with binaries. Further static analysis may be performed before dynamic profiling to decrease the profiling overhead [7]. For example, all memory loads and stores whose data dependences can be identified at static time could be excluded as profiling candidates. Our implementation in this paper skips induction and reduction variables (both basic and derived ones) and some read-only accesses.

The greatest downside of using an LLVM-based tracer is that it requires recompilation. Recompiling an application with instrumentation code is not always easy. It sometimes requires modifications in compiler tool chains and compiler driver code. The analyzer needs some information from the instrumentation phase, such as a list of instrumented loops and memory instructions. As the instrumentation phase is separated from the runtime profiling, such information should be transferred via a persistent medium like a file.

7 Experimental Results

7.1 Experimentation Methodology

We use 22 (out of 29) SPEC CPU2006 benchmarks [2] to report runtime overhead by running the *entire* execution of benchmarks with the reference input. Seven SPEC benchmarks were not profiled successfully due to several implementation issues. Among the successfully profiled 22 benchmarks, we observed that a few loops from functions that parse input files caused runtime errors due to incorrect binary-level loop instrumentation. We excluded such erroneous loops.

We should note that we intentionally used highly optimized binaries (`-O3` of Intel compilers) in the experimentation to reduce excessive profiling time overhead. It is true that a result from a highly optimized binary is virtually useless when we want to map the result to the source code by using debugging information. Many variables, statements, loops, and functions could be moved or eliminated by aggressive optimization. In practice, one should profile an unoptimized binary to obtain a human-readable result. However, the difference of native execution time between unoptimized and optimized binaries could be 10 times. The difference in memory overhead is not worse as the time overhead because memory accesses to local stacks are mostly optimized. The purpose of the experimentation is to measure the overhead, not to see actual dependence profiling results. For the later purpose, we recommend using unoptimized code with the train or test inputs.

We instrument all memory loads and stores except for certain types of stack operations and corner cases. Our profiler collects details of data-dependence information as enumerated in Section 2. We profile the 20 hottest loops (based on the number of executed instructions) *and* their inner loops. For comparing the overhead, we use the pairwise method. We also use seven OmpSCR benchmarks [1] to evaluate the input-sensitivity problem.

Our experimental results were obtained on machines with Windows 7 (64-bit), 8-core with Hyper-Threading Technology,

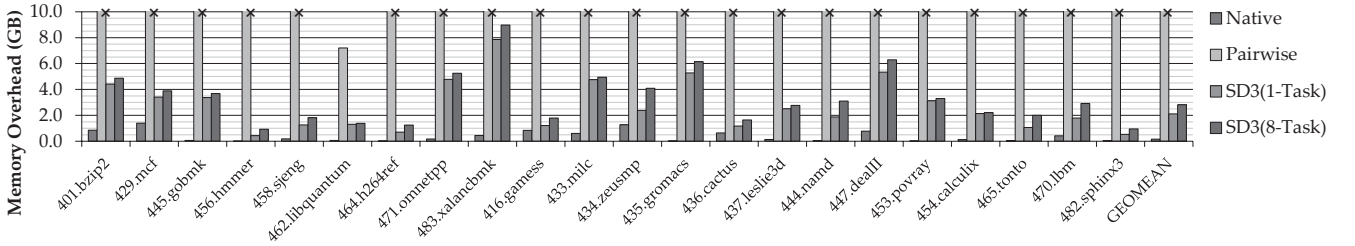


Fig. 17: Absolute memory overhead for SPEC 2006 with optimized (`-O3`) binaries and the reference inputs: 21 out of 22 benchmarks (✕ mark) need more than 12 GB in the pairwise method. The benchmarks natively consume 158 MB memory on average.

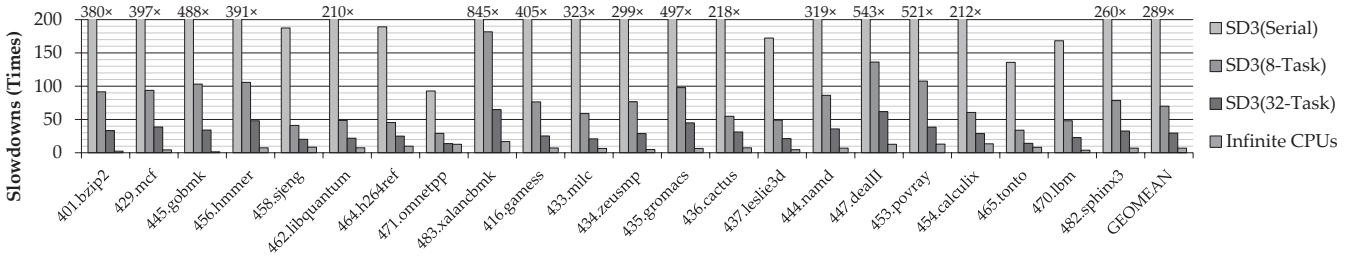


Fig. 18: Slowdowns (against the native run) for SPEC 2006 with optimized (`-O3`) binaries and the reference inputs: From left to right, (1)SD^3 (1-task on 1-core), (2) SD^3 (8-task on 8-core), (3) SD^3 (32-task on 32-core), and (4) estimated slowdowns of infinite CPUs. For all experiments, the address-range size is 128 bytes. The geometric mean of native runtime is 488 seconds on Intel Core i7 3.0 GHz.

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.

Currently, the LLVM implementation cannot instrument Fortran programs. The results in this paper are from the Pin-based profiler, but the LLVM-based profiler shows a similar performance.

7.2 Memory Overhead of SD^3

Fig. 17 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) dynamic profiling overhead. Among 22 benchmarks, 21 benchmarks cannot be profiled by the pairwise method on a 16 GB memory system. Sixteen out of 22 benchmarks consumed more than 12 GB even with the train inputs.

Fig. 19 shows the memory consumption of the pairwise method in 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. 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.

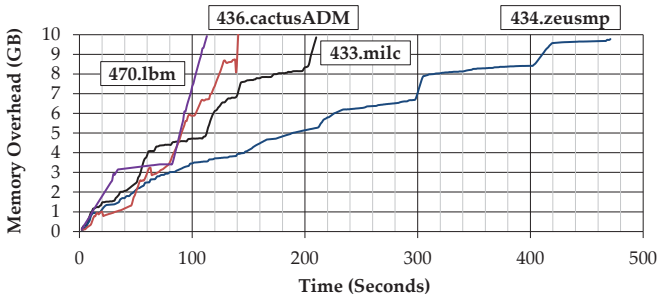


Fig. 19: Memory overhead of the pairwise for 4 SPEC 2006 benchmarks.

SD^3 successfully profiled 22 benchmarks on a 12GB machine although a couple of benchmarks needed 7+ GB. 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. Nonetheless, SD^3 still reduces memory consumption significantly compared to the pairwise method.

7.3 Time Overhead of SD^3

The time overhead results of SD^3 are presented in Fig. 18. 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 5.5, the number of tasks is the same as the number of cores in the experimentation.

As shown in Fig. 18, serial SD^3 shows a 289× slowdown on average, which is not surprising given the amount of computations on every memory access and loop execution. The overhead could be improved by implementing better static analysis that allows us to skip instrumenting loads and stores that have proved not to create any data dependences. As discussed in Sections 6.2.1 and 6.2.3, implementation techniques are critical to boost the baseline performance. Otherwise, a profiler could easily show a thousand slowdown.

When using eight tasks on eight 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.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 imbalance problem. The number of tasks is equal to the number of cores to minimize redundant loop handling and event distribution overhead. However, 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, the processing time for handling strides is not necessarily decreased in the parallelized SD^3.

We also estimate slowdowns with infinite CPUs. In such cases, each CPU observes only conflicts from a *single* memory address, which is extremely low. Therefore, the ideal speedup would be very close to the runtime overhead without the data-dependence profiling. 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. We compare only 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):

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

A 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 long enough to execute the majority of the source code. We found that the profiling results of OmpSCR were *not* changed by different input sets (i.e., Similarity = 1).

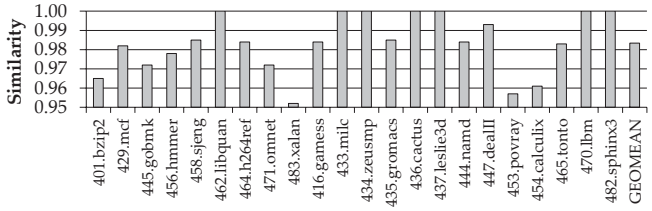


Fig. 20: Similarity of the results from different inputs: 1.00 means all results were identical (not the frequencies of dependence pairs).

Fig. 20 shows the similarity results of SPEC 2006, which are obtained from the reference and train input sets. Our data show that there are very high similarities (0.98 on average) in discovered dependence pairs. Recall that 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 minimize x86-64 artifacts such as stack operations in functions.

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 [28]. Their results showed that memory dependences between pipeline stages are highly stable and predictable over different inputs.

7.5 Discussion: Do Simple Samplings Work?

Using *sampling* would be an obvious option to limit the overhead of dependence profiling [5]. However, simple sampling techniques are inadequate for the purpose of our data-dependence profiler for two reasons: (1) Any sampling technique may introduce incorrect results, either false negatives or false positives (See Section 6.2.2); (2) Sampling may not be effective to solve the overhead problem.

• **Correctness:** Some usage models of the dependence profiling can tolerate inaccuracy, notably thread-level data speculation (TLDS) [5, 20, 27]. The reason is that in TLDS, incorrect speculations can be recovered by the rollback mechanism in hardware. However, when using dependence profiling to guide programmer-assisted parallelization for conventional multicore processors, we claim that the profile should be as correct as possible. In this context, "correctness" means that a profiling is correct for only a given input. We cannot prove the correctness for all possible inputs using a dynamic approach.

Fig. 21 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 potentially parallelizable. The reason is that the `work` function that generated dependences was called just one time: at the end of the iteration. A simple sampling technique that randomly samples iterations in a loop may make this error.

```
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. 21: 436.cactusADM, ScheduleTraverse.c

• **Overhead:** A simple sampling technique also does not solve the overhead problem inherently. To demonstrate the overhead problem, we compare the following three simple sampling techniques and SD^3 for the Jacobi benchmark in OmpSCR: (1) `Burst sampling`: Dependence profiling is triggered every N instructions and then lasts for M consecutive memory instructions. We fix N to be 5,000 and M to be 1,000 in this evaluation; (2) `Early stopping`: For any given loop, we profile up to N% of its total invocation count and M% of its total iteration count; (3) `Major-loops only`: We perform dependence profiling only on hot loops (at least 5% of the total execution time).

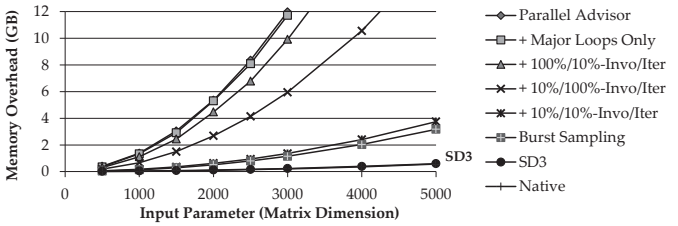


Fig. 22: Memory overhead for Jacobi in OmpSCR by different profiling mechanisms and policies.

Fig. 22 shows the results. When the matrix is 3000 by 3000, the pairwise method requires more than 12 GB of memory. Even when the samplings are used, the memory overhead is still in the order of GBs. In contrast, SD^3 requires only 50 MB because it can compress most of the memory references.

Of course, sophisticated and adaptive sampling techniques could solve the overhead problem. To the best of our knowledge, no advanced sampling is devised to reduce the overhead while maintaining the correctness. Most sampling techniques used in TLDS are not desirable for our purpose.

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` [17]. The profiling algorithm is similar to the evaluated pairwise method but suffers from huge memory and time overhead. Tournavitis et al. [30] proposed a dependence profiling mechanism to overcome the limitations of automatic parallelization. They also used a pairwise-like method and did not discuss the overhead problem explicitly.

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 [5, 8, 20, 27, 34], and many of them employ sampling or allow aliasing to reduce overhead. All of these approaches do not have to give accurate results like SD^3, assuming speculative hardware.

8.3 Reducing Overhead of Dynamic Analysis

Shadow Profiling [23], SuperPin [33], PiPA [37], and Ha et al. [11] 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. SD^3 further exploits data-level parallelism while reducing the memory overhead at the same time.

A number of techniques that compress dynamic profiling traces as well as standard compression algorithms like bzip have been proposed to save memory space [18, 22, 25, 35]. Their common approach is to use specialized data structures and algorithms to compress instructions and memory accesses that show specific patterns. METRIC [22], a tool to find cache bottlenecks, also exploits the stride behavior like SD^3. While METRIC only presents the compression algorithm, SD^3 introduces a number of algorithms that effectively calculate data dependence with the stride and non-stride formats.

9 Applications of SD^3 Algorithm

In this section, we demonstrate the usefulness of SD^3 with our proposed profiling tool, *Prospector* [14]. Prospector is built on top of SD^3. It performs loop and data dependence profiling and provides guidelines for how programmers can manually parallelize their applications.

We demonstrate an actual usage of Prospector with SPEC 179.art, summarized in Fig. 23. Although 179.art could be easily parallelized by TLS techniques, production compilers cannot automatically parallelize it. A typical programmer who does not know the algorithm detail would easily spend a day or more on parallelizing. Table 1 shows the result with the train input. The detailed steps are as follows:

• Prospector finds that the loop `scan_recognize:5` in Fig. 23 is the hottest loop with 79% execution coverage, only one invocation, and 20 iterations. Every iteration has almost an equal number of executed instructions (implying good balance). The loop could be a good parallelization candidate.

• Regarding dependence profiling, the loop has *no* loop-carried *flow* dependences except a reduction variable and an induction variable (i.e., loop counter variables).

TABLE 1: Profiling result of the loop, `scan_recognize:5`, in 179.art

Loop Profiling	79% execution coverage; 1 invocation and 20 iterations; Standard deviation of iteration lengths: 3.5%
Dependence Profiling	Loop-carried WAWs on `f1_layer` ... Temporary variables on `i`, `k`, `m`, `n` Induction variable on `j` at line 5 Reduction variable on `highest_confidence`
	No Loop-carried RAWs: *may be parallelizable*

```
 1: void scan_recognize(startx, starty, endx, endy, stride)
 2: {
 3:   ...
 4:   #pragma omp for private (i,k,m,n)
 5:   for (j = starty; j < endy; j += stride)
 6:     for (i = startx; i < endx; i += stride){
 7:       ..
 8:       pass_flag = 0;
 9:       match();
10:       if (pass_flag == 1) {
11:         if (set_high[tid][0] == TRUE) {
12:           highx[tid][0] = i, highy[tid][0] = j;
13:           set_high[tid][0] = FALSE;
14:         }
15:         if (set_high[tid][1] == TRUE) {
16:           ...
18:     }  // End of for-i
...
30: void match()
31: {
32:   reset_nodes();
34:   while (!matched) {
35:     ...
48:     int match_cnfd = simtest2();
41:     if ((match_cnfd) > rho) {
43:        pass_flag = 1;
44:        if (match_cnfd > highest_confidence[tid][winner]){
45:          highest_confidence[tid][winner] = match_cnfd;
46:          set_high[tid][winner] = TRUE;
47:        }
48:     ...

70: void reset_nodes()
71: {
72:   for (i=0;i<numf1s;i++) {
73:     f1_layer[tid][i].W = 0.0;
74:     Y[tid][i].y = 0.0;

Need parallel reduction
```

Fig. 23: Simplified parallelization steps of 179.art on multicore by Prospector's result: (1) Privatize global variables; (2) Insert OpenMP pragmas; (3) Add a reduction code (not shown here)

• `scan_recognize:5` has many loop-carried output dependences on global variables such as `f1_layer` and `Y`. These variables are not automatically privatized to threads, so we perform *privatization*, explicitly allocating thread-local private copies of these global variables. (Not shown in Fig. 23)

• Temporary variables (in the scope of the loop:5) are also found such as `i` at line 6. In this code, we need to insert `i` to OpenMP's private list. We classify a set of dependences as a temporary variable if the very first time access in an iteration is a write (initialization) and then is followed by loop-independent flow dependences (uses).

• A potential reduction variable (in the scope of the loop:5), `highest_confidence`, is identified. The variable is intended to calculate the maximum, which is the commutative operation. We manually modify the code to obtain local results and compute the final answer. (Not shown in Fig. 23) However, detecting a reduction from raw dependence results has a several challenges such as verifying commutativity property. We will investigate this problem more in our future work.

Programmers do not need to know the very details of 179.art. By interpreting Prospector's results, programmers can easily understand and finally parallelize. It is true that Prospector cannot prove the parallelizability. However, when compilers cannot automatically parallelize the loop, programmers must prove its correctness by hand or verify it empirically using exhaustive tests. In such cases, we claim Prospector and SD^3 are very valuable for programmers. Recent commercial tools [12, 31] and research that use profiling to assist parallelization [9, 26, 28, 29] also advocate this claim.

10 Conclusions and Future Work

This paper proposed a new efficient data-dependence profiling technique called SD^3. SD^3 is the first solution that attacks both the memory and the time overhead of data-dependence profiling at the same time. For the memory overhead, SD^3 not only reduces the overhead by compressing memory references that show stride behavior, but also provides a new data-dependence checking algorithm with the stride format. SD^3 presents several algorithms on handling the stride data structures. For the time overhead, SD^3 exploits pipeline and data-level parallelism in our data-dependence profiling itself while keeping the effectiveness of the stride compression. Several issues for higher speedups were discussed. SD^3 successfully profiles top 20 loops and their inner loops of 22 optimizeSPEC CPU2006 benchmarks with optimized binaries and the reference inputs.

In future work, we will focus on how such an efficient data-dependence profiler can actually provide advice on parallelizing legacy code as discussed in Section 9. Also, advanced static analysis can help further to reduce the overhead.

11 Acknowledgements

Special thanks to Puyan Lotfi and Hyojong Kim for their supports for the implementation of LLVM-based SD^3. We thank Geoffrey Lowney, Robert Cohn, and John Pieper for their feedback and support; Paul Stravers for providing an experimental result; Pranith D. Kumar, and HPArch members for their comments; finally, many valuable comments from the anonymous reviewers who help this paper. Minjang Kim, Nagesh B. Lakshminarayana, and Hyesoon Kim are supported by National Science Foundation (NSF) CCF0903447, NSF/SRC task 1981, NSF CAREER award 1139083, Intel Corporation, and Microsoft Research. Any opinions, fndings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily refect those of NSF, Microsoft, or Intel.

References

[1] *OmpSCR: OpenMP Source Code Repository*. http://sourceforge.net/projects/ompscr/.

[2] Standard Performance Evaluation Corporation, *SPEC CPU2006*. http://www.spec.org/cpu2006/.

[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 *Proceedings of the fifth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPOPP '95, 1995.

[4] V. T. Chakaravarthy. New results on the computability and complexity of points–to analysis. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, 2003.

[5] T. Chen, J. Lin, X. Dai, W.-C. Hsu, and P.-C. Yew. Data dependence profiling for speculative optimizations. In *Compiler Construction*, Lecture Notes in Computer Science. 2004.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[7] D. Das and P. Wu. Experiences of using a dependence profiler to assist parallelization for multi-cores. In *IPDPS Workshops*, 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 *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, 2004.

[9] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. In

Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, PLDI '11, 2011.

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

[11] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, 2009.

[12] Intel Corporation. *Intel Parallel Advisor*. http://software.intel.com/en-us/articles/intel-parallel-advisor/.

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

[14] M. Kim, H. Kim, and C.-K. Luk. Prospector: Helping parallel programming by a data-dependence profile. In *The 2nd USENIX conference on Hot topics in parallelism*, HotPar '10, 2010.

[15] M. Kim, H. Kim, and C.-K. Luk. SD^3: A scalable approach to dynamic data-dependence profiling. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, 2010.

[16] X. Kong, D. Klappholz, and K. Psarris. The I test: An improved dependence test for automatic parallelization and vectorization. *IEEE Trans. Parallel Distrib. Syst.*, 2, July 1991.

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

[18] J. R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, 1999.

[19] 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*, CGO '04, 2004.

[20] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: a TLS compiler that exploits program structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '06, 2006.

[21] 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 *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, 2005.

[22] 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 Trans. Program. Lang. Syst.*, 29, April 2007.

[23] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, 2007.

[24] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 1997.

[25] G. D. Price, J. Giacomoni, and M. Vachharajani. Visualizing potential parallelism in sequential programs. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, 2008.

[26] S. Rul, H. Vandierendonck, and K. De Bosschere. A profile-based tool for finding pipeline parallelism in sequential programs. *Parallel Comput.*, 36, September 2010.

[27] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, 2000.

[28] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, 2007.

[29] G. Tournavitis and B. Franke. Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, 2010.

[30] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, 2009.

[31] Vector Fabrics. *Pareon: Optimize applications for multicore phones, tablets and x86.* http://www.vectorfabrics.com/.

[32] C. von Praun, R. Bordawekar, and C. Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, 2008.

[33] S. Wallace and K. Hazelwood. SuperPin: Parallelizing dynamic instrumentation for real-time performance. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, 2007.

[34] P. Wu, A. Kejariwal, and C. Caşcaval. Languages and compilers for parallel computing. chapter Compiler-Driven Dependence Profiling to Guide Program Parallelization. 2008.

[35] X. Zhang and R. Gupta. Whole execution traces. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, 2004.

[36] X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, 2009.

[37] Q. Zhao, I. Cutcutache, and W.-F. Wong. PiPA: pipelined profiling and analysis on multi-core systems. In *Proceedings of the 6th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, 2008.

Minjang Kim is a PhD candidate in the College of Computing at Georgia Tech. His research interests are parallel programming tools and compilers. He received a BS in computer science and a BS in naval architecture and ocean engineering, both from Seoul National University. In 2008, he obtained an MS in computer science from Georgia Tech.

Nagesh B Lakshminarayana is a 3rd year PhD student in the College of Computing at Georgia Tech. His research interests are in the field of Computer Architecture. He obtained his Master's degree in computer science from Georgia Tech in 2009 and his Bachelor's degree in computer science and engineering from RV College of Engineering, India in 2003.

Hyesoon Kim is an assistant professor in the School of Computer Science at Georgia Institute of Technology. Her research interests include high-performance energy-efficient heterogeneous architectures, and programmer-compiler-microarchitecture interaction. She received a BA in mechanical engineering from Korea Advanced Institute of Science and Technology (KAIST), an MS in mechanical engineering from Seoul National University, and an MS and a PhD in computer engineering at The University of Texas at Austin.

Chi-Keung Luk is a senior staff engineer at Intel. His research interests include parallel programming tools and techniques, compilers, and virtualization. Luk has a Ph.D. in computer science from the University of Toronto.