# Fast Data-Dependence Profiling by Skipping Repeatedly Executed Memory Operations

Zhen Li[1(✉)], Michael Beaumont[2], Ali Jannesari[1], and Felix Wolf[1]

[1] Technische Universität Darmstadt, 64289 Darmstadt, Germany
{li,jannesari,wolf}@cs.tu-darmstadt.de
[2] RWTH Aachen University, 52062 Aachen, Germany
michael.beaumont@rwth-aachen.de

**Abstract.** Nowadays, more and more program analysis tools adopt profiling approaches in order to obtain data dependences because of their ability of tracking dynamically allocated memory, pointers, and array indices. However, dependence profiling suffers from high time overhead. To lower the overhead, former dependence profiling techniques either exploit features of the specific program analyses they are designed for, or let the profiling process run in parallel. Although they successfully lowered the time overhead of dependence profiling by a certain amount, none of them have tried to solve the fundamental problem that causes the high time overhead: the memory operations that are repeatedly executed in loops. In most of the time, these memory operations lead to exactly the same data dependences. However, a profiling method has to profile all these memory operations over and over again in order to not miss a single dependence that may occur just once. In this paper, we present a method that allow a dependence profiling technique to skip memory operations that are repeatedly executed in loops without missing any single data dependence. Our method works with all types of loops and does not require any prepossessing like source annotation of the input code. Experiment results show that our method can lower the time overhead of data-dependence profiling by up to 52 %.

**Keywords:** Data-dependence · Profiling · Optimization · Program analysis · Parallel programming

## 1 Introduction

Extracting data dependences from programs serves as the foundation of many program analysis and transformation methods. Especially, since data dependence is one of the main factors that preventing parallelism, data-dependence analysis is the base of nearly all the tools that discover parallelism in parallel programming area. Tools for discovering parallelism [6,10,11,15,18,24] identify the most promising parallelization opportunities. Runtime scheduling frameworks [4,7,17,22] add more parallelism to programs by dispatching code sections in a more effective way. Automatic parallelization tools [1,8,13,25] transform


G. Wang et al. (Eds.): ICA3PP 2015, Part IV, LNCS 9531, pp. 583–596, 2015.
DOI: 10.1007/978-3-319-27140-8_40

sequential into parallel code automatically. Method that suggests parallel patterns [9] helps programmer to choose the most promising pattern for parallelizing code. All the tools and methods mentioned above have in common the fact that they rely on data-dependence information to achieve their goals.

Data dependences can be obtained in two main ways: static and dynamic analysis. Static approaches determine data dependences without executing the program. Although they are fast and even allow fully automatic parallelization in some cases [1,8], they lack the ability to track dynamically allocated memory, pointers, and dynamically calculated array indices. This usually makes their assessment conservative, limiting their practical applicability. In contrast, dynamic dependence profiling captures only those dependences that actually occur at runtime. Although dependence profiling is inherently input sensitive, the results are still useful in many situations, which is why such profiling forms the basis of many program analysis tools [3,6,10,11,15]. Moreover, input sensitivity can be addressed by running the target program with changing inputs and computing the union of all collected dependences.

However, a serious limitation of data-dependence profiling is high time overhead. It may significantly prolong the analysis, sometimes requiring an entire night [19]. This is because dependence profiling requires all memory operations to be instrumented and records of all accessed memory locations to be kept. Many solutions have been proposed to lower the overhead. The first solution is to limit the scope to the subset of the dependence information needed for the analysis they have been created for, sacrificing generality and, hence, discouraging reuse. The second solution is sampling, also tries to analyze a subset of all the memory operations but without losing generality. Based on sampled memory operations combined with a probabilistic model, the second solution profiles data dependence with a sacrifice of accuracy. The last solution is to let the data-dependence profiling process run in parallel. This is possible because some data dependences related to one memory address do not affect other dependences related to another memory address. It does not lose generality or accuracy, but it surely needs much more effort to implement.

An observation is that many memory operations in loops are repeatedly executed. In most of the time they lead to always the same data dependences, but still need to be analyzed over and over again just because of some special data dependences that rarely occur. None of the solutions mentioned above tried to deal with this problem. In this paper, we present a method that allow a dependence profiling technique to skip memory operations that are repeatedly executed in loops without missing any data dependence. Our method works with all types of loops, and allows nesting. Furthermore, our method can be applied in combination with the existing overhead-reducing techniques mentioned above. Experiments results on applications from NAS Parallel Benchmarks 3.3.1 [5] and Starbench parallel benchmark suite [2] show that our method can lower the time overhead of data-dependence profiling by up to 52 %.

The remainder of the paper is organized as follows. First, we summarize related work in Sect. 2. Then, we introduce the work flow of data-dependence

profiling in Sect. 3, providing a background of our method. In Sect. 4, we describe the details of skipping memory operations in loops. Evaluation of our method and a discussion on the characteristics of skipped memory operations are presented in Sect. 5. Finally, we conclude the paper and outline future prospects in Sect. 6.

## 2 Related Work

In previous dynamic data-dependence profiling techniques, their overhead was reduced through three major ways: tailoring the profiling technique to a specific analysis, sampling memory operations, or parallelizing the profiling process.

Using dependence profiling, Kremlin [6] determines the length of the critical path in a given code region. Based on this knowledge, it calculates a metric called self-parallelism, which quantifies the parallelism of the region. Instead of pair-wise dependences, Kemlin records only the length of the critical path. Alchemist [24], a tool that estimates the effectiveness of parallelizing program regions by asynchronously executing certain language constructs, profiles dependence distance instead of detailed dependences. Although these approaches profile data dependences with low overhead, the underlying profiling technique has been tailored to the specific analysis, and has difficulty in supporting other program analyses.

Another solution to decrease the profiling overhead is to use approximate representation rather than instrument every memory operation. Previous work [20] tried to ignore memory operations in a code section when it had been executed more than $2^{32-k}$ times. However, when setting $k = 10$, only 33.7 % of the memory operations are covered, which can lead to significant inconsistency in profiled data dependences. Vanka and Tuck [21] profiled data dependencies based on signature and also compared the accuracy under different sampling rates. In this work, sampling was done in function level. A sampling rate of $M$ means the next $M - 1$ invocations of a function will be skipped. When decreasing the sampling rate from 1 to 100, an obvious drain of accuracy was observed.

There are also approaches that reduce the time overhead of dependence profiling through parallelization. For example, SD3 [12] exploits pipeline and data parallelism to extract data dependences from loops. DiscoPoP [16] distributes all the memory operations of a program among a number of worker threads based on the accessed address, and a redistribution table is used to ensure balanced workload. Multi-slicing [23] leverages compiler support for parallelization. Before execution, the compiler divides the profiling job into multiple profiling tasks through a series of static analyses. All these approaches successfully reduced the time overhead of data-dependence profiling without losing generality or accuracy. However, they still analyze all memory operations. At a certain time, the parallelism exist among different memory addresses cannot be exploited further by increasing the number of worker threads, and the huge number of memory operations that need to be processed sequentially dominates the profiling overhead.

Like Kremlin, Alchemist, and former work [20,21], our method also profiles only a subset of all the memory operations of a program. Unlike these methods,

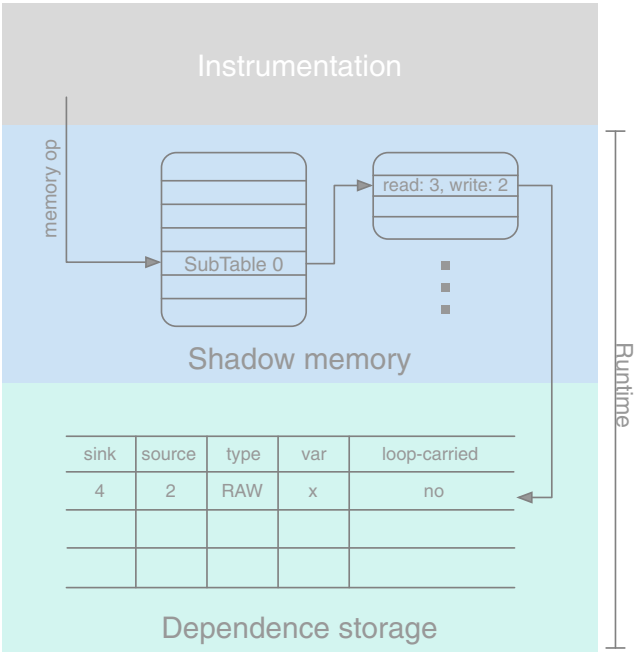



**Fig. 1.** Work flow of data-dependence profiling.

our approach does not lose generality or accuracy. The skipped memory operations are those repeatedly executed and lead to identical data dependences. Theoretically, our approach can work with any code sections that are executed more than once.

## 3 Background

A profiling techniques usually contains two parts: an instrumentation component that inserts analysis functions into the target code following specific rules, and a runtime library that implements the analysis functions and data structures. In data-dependence profiling, the instrumentation component inserts analysis functions for every memory operation. Instrumented code will be linked against the runtime library and executed. The runtime library is further divided into two components. The first component is called *shadow memory*. During runtime, the analysis functions keep tracking each memory locations accessed in the target application, and maintain access status of each memory location in a separate memory space. The second component is data-dependence storage, where data dependences are built and stored when the statuses in shadow memory are changed.

Figure 1 shows the work flow of data-dependence profiling. Among the three phases, instrumentation can be done statically, and time overhead of instrumentation is usually negligible. The main time overhead are caused by the remaining

two phases: updating shadow memory and building dependence. Both shadow memory and dependence storage are typically implemented based on table-like data structures where each memory address or data dependence has an entry. Given the truth that the number of memory operations and data dependences are usually very large, the overhead is mainly incurred by searching, updating, and inserting elements to the data structures. As a result, data-dependence profiling typically slows the program down by a factor ranging from 100 to 150. [10]

However, not every memory operation has to be processed through all the three phases. Let us take the loop shown in Fig. 2 as an example. After profiling two iterations of the loop, data dependences are complete. Table 1 shows the dependences. *Source* and *sink* are the source code locations of the former and the latter memory operations, respectively. *Type* is the dependence type, including read after write (RAW), write after read (WAR), and write after write (WAW). *Variable* is the variable that causing a dependence. When source and sink of a dependence belong to different iterations of a loop, we call the dependence a *loop-carried* dependence.

```
1  while (k > 0) {
2      sum += k * 2;
3      k--;
4  }
```

**Fig. 2.** A simple loop where data dependences will not change over iterations.

**Table 1.** Data dependences of the loop shown in Fig. 2.

| ID | Sink | Source | Type | Variable | Loop-carried |
|---|---|---|---|---|---|
| 1 | 2 | 2 | write after read (WAR) | sum | no |
| 2 | 3 | 1 | write after read (WAR) | k | no |
| 3 | 3 | 2 | write after read (WAR) | k | no |
| 4 | 3 | 3 | write after read (WAR) | k | no |
| 5 | 1 | 3 | read after write (RAW) | k | yes |
| 6 | 2 | 2 | read after write (RAW) | sum | yes |
| 7 | 2 | 3 | read after write (RAW) | k | yes |
| 8 | 3 | 3 | read after write (RAW) | k | yes |

Among the dependences shown in Table 1, dependence 1–4 can be obtained within the first iteration, and dependence 5–8 will be added once the second iteration is done. After that, no more data dependence will be built, no matter how many iterations the loop has. In this case, profiling these memory operations over and over again is just a waste of time. It may be necessary to keep updating statuses in shadow memory for correctness, but we definitely do not want to touch dependence storage after data dependences for a code section are complete. In the next section, we show how we skip these memory operations after the dependences are fully obtained to accelerate the profiling process.

## 4 Approach

An abstract analysis function for a memory operation looks like this:

```
mem_op(accessType, accessInfo, addr).
```

For a memory operation, `accessType` can be either read or write. It does not change over time. In practice, two analysis functions will be created for read and write operations, respectively. Necessary information needed to update shadow memory are stored in `accessInfo`, and passed into the analysis function. Usually, `accessInfo` is the identifier of the associate memory operation. For example, the address of the instruction, the source line location, the variable name, or a combination of such information. Depending on concrete implementation, `accessInfo` may or may not be unique to each memory operation. However, for one memory operation, its `accessInfo` does not change. `addr` is the memory address accessed by the memory operation. It can change if the address is referred by pointers.

### 4.1 Condition on `addr`

If a memory operation can be safely skipped, the memory address it accesses must not change over time. For simplicity, we create a variable called `lastAddr` for each memory operation storing the memory address accessed by the memory operation last time. And we require

```
addr == lastAddr
```

to be a necessary condition if a memory operation can be safely skipped. `lastAddr` should be initialized with an address which is rarely accessed, like $0 \times 0$.

When the condition on `addr` holds, it only means that the current memory operation has been profiled before. It does not mean all data dependences that are related to the current memory operation have been obtained. Again, let us take the loop shown in Fig. 2 as an example. After applying the condition on `addr`, all the memory operations in the first iteration will be profiled, and dependence 1–4 in Table 1 are obtained. However, from the second iteration, memory operations are skipped because the addresses they access do not change. Thus, we name the condition on `addr` a necessary condition, and we still need other conditions to decide if a memory operation can be skipped.

### 4.2 Condition on `accessInfo`

The key to cover all data dependences is to decide when to resume profiling once the profiling has been paused. Our solution is to have a mechanism that allows a memory operation be notified if the access status of its memory address has changed, so that the memory operation must be profiled again.

In order to track the access status of a memory address, the shadow memory stores `accessInfo` of the last read operation and the last write operation to

the address. We call them `statusRead` and `statusWrite`, respectively. We then create two variables `lastStatusRead` and `lastStatusWrite` for each memory operation, storing the `accessInfo` of the last read operation and the last write operation to the memory address when the memory operation was profiled *last* time, respectively. Then we require

```
statusRead == lastStatusRead &&
statusWrite == lastStatusWrite
```

to be another necessary condition if a memory operation can be safely skipped. Both `lastStatusRead` and `lastStatusWrite` should be initialized with impossible values for `accessInfo`.

When the condition on `accessInfo` holds, it means that the access status of the memory address has been seen before. We say "has been seen before" because the address may change, and the access status of the current memory address may just coincidentally be the same as the access status of another address. This is very likely to happen when `accessInfo` is not unique to each memory operation. However, combing the two conditions on `addr` and `accessInfo` will give the sufficient condition if a memory operation can be safely skipped: a memory operation has been profiled before, and the access status of its memory address has not changed since it was profiled last time.

When the conditions do not hold anymore, it means either the memory operation accesses a different memory address, or the access status of the memory address has changes. No matter which situation it is, the memory operation must be profiled again in order to cover new data dependences.

### 4.3 Example

In this section, we show how our method works on a simple example, and a special case where a memory operation can be skipped even without updating its status in shadow memory.

```
1  loop:
2      op1:  write x
3      op2:  read  x
4      op3:  read  x
5      op4:  write x
6  end
```

**Fig. 3.** A loop containing for memory operations on the same memory address.

Figure 3 shows a loop with four memory operations (`op1`–`op4`). All the memory operations access the same memory address `x`. We show memory operations instead of source code so that the profiling process can be clearly illustrated. Data dependences of the loop shown in Fig. 3 are listed in Table 2.

**Table 2.** Data dependences of the loop shown in Fig. 3.

| ID | Sink | Source | Type | Variable | Loop-carried |
|---|---|---|---|---|---|
| 1 | op2 | op1 | read after write (RAW) | x | no |
| 2 | op3 | op1 | read after write (RAW) | x | no |
| 3 | op4 | op3 | write after read (WAR) | x | no |
| 4 | op1 | op4 | write after write (WAW) | x | yes |

**Table 3.** Changing process of values of `lastStatusRead` and `lastStatusWrite` in the profiling process on the loop shown in Fig. 3.

| Op | lastStatusRead | | | | lastStatusWrite | | | |
|---|---|---|---|---|---|---|---|---|
| | init | 1st | 2nd | 3rd | init | 1st | 2nd | 3rd |
| write x | — | 0 | 3 | S | — | 0 | 4 | S |
| read x | — | 0 | 3 | S | — | 1 | 1 | S |
| read x | — | 2 | S | S | — | 1 | S | S |
| write x | — | 3 | S | S | — | 1 | S | S |

**Table 4.** Changing process of the statuses in shadow memory in the profiling process on the loop shown in Fig. 3.

| | init | op1 | op2 | op3 | op4 | op1 | op2 | op3 | op4 |
|---|---|---|---|---|---|---|---|---|---|
| statusRead | 0 | 0 | 2 | 3 | 3 | 3 | 2 | 3 | 3 |
| statusWrite | 0 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 4 |

The changing process of values stored in `lastStatusRead` and `last StatusWrite` for each memory operation is shown in Table 3. "1st", "2nd"", and "3rd" refer to the first, the second, and the third iteration of the loop, respectively. An "S"means the memory operation is skipped, otherwise the memory operation is processed and the value of `lastStatusRead` and `lastStatusWrite` are updated.

The changing process of the accessing status of `x` in shadow memory is shown in Table 4. We adopt the most common design, where for each memory address the last read operation and the last write operation to the address are stored. In both Tables 3 and 4, we use "1" for op1, "2" for op2, and so fort.

Let us examine the profiling process step by step. In the beginning, `last StatusRead` and `lastStatusWrite` are initialized to "–", `statusRead` and `statusWrite` are 0, and `lastAddr` is `0 × 0`. Now comes `op1`. Since `addr` is not equal to `lastAddr`, `op1` is processed. Statuses in shadow memory are loaded into `lastStatusRead` and `lastStatusWrite`, which are both 0 in case of `op1`. Then `op1` updates shadow memory. `statusWrite` of `x` is now 1.

The same process happens to `op2`. The difference is that when `op2` is executed, `statusRead` and `statusWrite` of `x` has been changed to 0 and 1, respectively. With `statusWrite` is no longer zero, a read-after-write (RAW)

dependence from op2 to op1 is built, which is the first dependence shown in Fig. 2. The profiling process continues, and dependence 2, 3 are built when op3 and op4 are profiled.

Now the profiling process enters the second iteration, and op1 comes again. Although the condition on addr holds this time, the condition on AccessInfo fails. The last time op1 was profiled, the last read operation (stored in lastStatusRead) and the last write operation (in lastStatusWrite) to x were 0. After the first iteration is completed, they are 3 and 4. op1 must be profiled again in order to cover new dependences. Thus, the last data dependence in Table 2 is built. The same situation also happens to op2, but it only leads to a read-after-read (RAR) dependence, which is ignored in most of the data-dependence profilers.

Both condition holds when op3 is executed again, and it is skipped. No dependence instance is built, and no query to the dependence storage. Note that shadow memory is still updated for correctness. From then on, all further memory accesses to x in the same loop are skipped, and no dependence is missed. The dependence storage is touched only four times, exactly as the number of dependences the loop contains.

**Special Case.** When the loop contains only op1, op2, and op3, statusWrite to x will be always 1. This is a special case where the following condition holds:

```
currentWrite == statusWrite == lastStatusWrite.
```

In this case, a write operation can be skipped without updating shadow memory. The same applies for read operation as well.

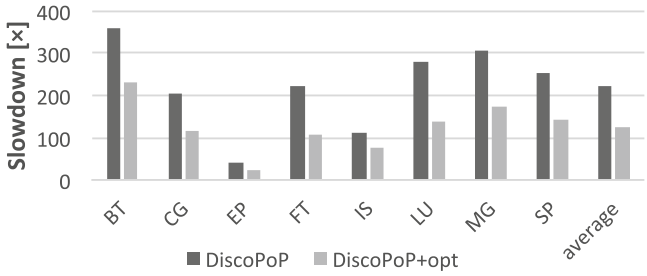
## 5 Evaluation

We implemented our method in the data-dependence profiler [16] of DiscoPoP [14,15]. The profiler contains several different implementations of shadow memory. In this paper, we choose an implementation where statusRead and statusWrite of a memory address are stored in two separate sets called *readSet* and *writeSet*, respectively. Both of the two sets are non-approximate representation, meaning no false positives or false negatives will be built.

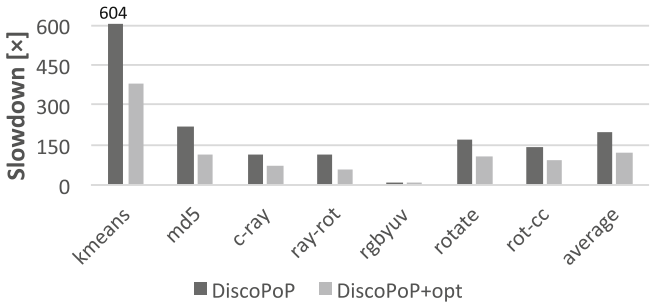
We conducted a range of experiments to evaluate the effectiveness of our method. Test cases are the NAS Parallel Benchmarks 3.3.1 [5] (NAS), a suite of programs derived from real-world computational fluid-dynamics applications, and a few applications from the Starbench parallel benchmark suite [2], which covers programs from diverse domains, including image processing, information security, machine learning and so on.

### 5.1 Time Overhead

Figure 4 shows the slowdowns of the data-dependence profiler on NAS benchmarks and *kmeans* from Starbench before (dp) and after (dp+opt) applying the




(a) Slowdown on NAS.




(b) Slowdown on Starbench.

**Fig. 4.** Slowdowns of the data-dependence profiler of DiscoPoP on NAS and Starbench benchmarks with (DiscoPoP+opt) and without (DiscoPoP) skipping repeatedly executed memory operations.

mechanism of skipping memory operations that are repeatedly executed in loops. As it shown, our method reduces the slowdown of data-dependence profiling on all of the test cases. The highest slowdown reduction shows in *FT* (52.0 %), and the lowest shows in *rot-cc* (31.1 %). On average, our method reduces the time overhead of data-dependence profiling by 41.3 %. The outputs after applying our method were compared to the original ones using *diff* tool, and no difference is observed.

Whether our method reduces the time overhead of data-dependence profiling on an application depends on the computation pattern of the application. Theoretically, the more work done in loops (or other repetitive manner), the more effective our method will be. If a program does not have any code sections that are executed more than once, which is obviously very uncommon for a real-world application, our method should actually bring a minor time overhead due to condition checking. In test cases *FT*, *LU*, and *CG*, the biggest hot spots are all loops. Applying our method on these test cases give reductions on slowdown of 52 %, 51 %, and 44 %, respectively.

Memory access pattern is another factor that can affect the effectiveness of our method. In the worst case, accessed memory addresses change in every iteration, which means the profiling process cannot be paused. This usually

**Table 5.** Statistics of memory operations that lead to data dependence but skipped on NAS benchmarks and *kmeans* from Starbench.

| Benchmark | read | | write | | read+write | |
|---|---|---|---|---|---|---|
| | total | skipped [%] | total | skipped [%] | total | skipped [%] |
| BT | 743 969 748 | 71.94 | 104 153 401 | 22.66 | 848 123 149 | 65.89 |
| CG | 562 665 608 | 79.20 | 82 428 819 | 92.32 | 645 094 427 | 80.88 |
| EP | 1 268 263 496 | 96.75 | 528 633 275 | 89.00 | 1 796 896 771 | 94.47 |
| FT | 1 034 144 426 | 99.68 | 274 436 113 | 99.53 | 1 308 580 539 | 99.65 |
| IS | 26 061 226 | 82.69 | 10 596 042 | 73.53 | 36 657 268 | 80.04 |
| LU | 368 187 710 | 87.09 | 36 303 260 | 41.92 | 404 490 970 | 83.04 |
| MG | 66 160 096 | 82.60 | 5 876 449 | 53.88 | 72 036 545 | 80.26 |
| SP | 450 997 264 | 83.54 | 51 853 149 | 44.31 | 502 850 413 | 79.50 |
| kmeans | 1 124 603 733 | 65.27 | 225 500 303 | 87.97 | 1 350 104 036 | 69.06 |
| md5 | 3 908 055 | 91.05 | 1 368 725 | 97.99 | 5 276 780 | 92.85 |
| c-ray | 1 251 777 658 | 64.77 | 264 217 429 | 48.35 | 1 515 995 087 | 61.91 |
| ray-rot | 500 462 138 | 56.48 | 133 222 408 | 47.65 | 633 684 546 | 54.62 |
| rgbyuv | 25 639 777 | 89.28 | 15 977 310 | 85.32 | 41 617 087 | 87.76 |
| rotate | 328 610 773 | 89.17 | 53 662 659 | 56.59 | 382 273 432 | 84.60 |
| rot-cc | 427 139 027 | 91.67 | 76 733 411 | 57.34 | 503 872 438 | 86.44 |
| average | — | 82.08 | — | 66.56 | — | 80.06 |

happens when computation is based on array or matrix. Results on test cases *BT*, *IS*, *rotate*, and *rot-cc* are affected due to this problem.

### 5.2 Skipped Memory Operations

In the second experiment, we get statistics of the memory operations that lead to data dependence but skipped in each test case. As most of the data-dependence profilers do, read-after-read (RAR) dependences are not profiled in our experiment.

Table 5 shows the statistics. In each column group, "percent" gives the percent of memory operations skipped of the type specified for the group. As it is shown, on average 80.06 % of the memory operations that lead to data dependences are skipped. It is surprising that the full data dependence set of an application can be obtained by profiling only 20 % of its memory operations (or even less because those do not lead to dependences are skipped already). The results give us an insight of how much time were wasted in a classic data-dependence profiler that profiles identical data dependences over and over again.

Although on average about 80 % of the memory operations that lead to data dependence are skipped, the slowdown reductions shown in Fig. 4 never achieve 60 %. There are two reasons for this. Firstly, in most cases, skipping a memory operation means skipping the data dependence building phase. Overhead is still incurred by updating shadow memory. The second reason is that profiling a write operation is more complex than profiling a read, and the percentage of

skipped write operations (66.56 %) is less than the percentage of read (82.08 %). Profiling a write operation needs to check both WAW and WAR dependences, while profiling read operation only needs to check RAW.

### 5.3 Memory Overhead

Our method introduces a minor overhead on memory consumption of data-dependence profiling because of the variables created for condition check. However, compared to the memory overhead of shadow memory, the memory overhead of our method can be ignored. In our experiments, one 64-bit integer (`lastAddr`) and two 32-bit integers (`lastStatusRead` and `lastStatusWrite`) are created for each *distinct* memory operation. However, the number of distinct memory operations is usually small comparing to the number of total memory operations due to loops and other code blocks that are repeatedly executed. For example, *kmeans* has $10^9$ memory operations in total and iterates 300 times. Thus, the number of distinct memory operations in *kmeans* is roughly $3 \times 10^6$. With 16 Bytes memory overhead each, our method results in about 50 MB memory consumption. The memory overhead of shadow memory, however, is almost ten times of that. Memory consumption of the state-of-the-art data-dependence profilers [12,16] ranges from several hundred mega bytes to several giga bytes. Using 10 % memory more to reduce the time overhead by 30–50 % is definitely a bargain.

## 6 Conclusion

Data-dependence profiling has a huge time overhead because it applies heavy analysis to every memory operation of the target program. Existing solutions to reduce the number of memory operations needed to be analyzed includes static analysis and sampling. However, the number of data dependences that can be determined statically is usually limited. Sampling, on the other side, skips memory operations according to certain pre-defined rules with no respect to the memory access pattern of the target program.

In this paper, we proposed a fast data-dependence profiling method that can skip memory operations repeatedly executed in loops. By storing a short profiling history for each memory operation, our method recognizes memory operations that have been recently profiled and skips them, and, which is more important, resumes profiling when the access pattern changes. According to the experiment results, our method reduces the time overhead of data-dependence profiling by 42.5 % on average. Furthermore, in contrast to sampling approaches, our method ensures consistent state in shadow memory, lowering the time overhead without losing accuracy. Finally, our method can cooperate with existing overhead-lowering techniques for data-dependence profiling like static analysis and parallelization.

We plan to develop a fast data-dependence profiler with the help of both former techniques of reducing overhead like parallelization and the method presented in this paper. We are also interested in applying our method to profilers

that built on top of virtual machines, where the original code without instrumentation can be scheduled into execution when all its memory operations are marked as skipped.

## References


1. Amini, M., Goubier, O., Guelton, S., Mcmahon, J.O., Pasquier, F.X., Pan, G., Villalon, P.: Par4All: From convex array regions to heterogeneous computing. In: Proceedings of the 2nd International Workshop on Polyhedral Compilation Techniques, IMPACT 2012 (2012)
2. Andersch, M., Juurlink, B., Chi, C.C.: A benchmark suite for evaluating parallel programming models. In: Proceedings 24th Workshop on Parallel Systems and Algorithms, PARS 2011, pp. 7–17 (2011)
3. Atre, R., Jannesari, A., Wolf, F.: The basic building blocks of parallel tasks. In: Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores, COSMIC 2015, pp. 3:1–3:11. ACM, New York (2015)
4. August, D.I., Huang, J., Beard, S.R., Johnson, N.P., Jablin, T.B.: Automatically exploiting cross-invocation parallelism using runtime information. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, pp. 1–11. IEEE Computer Society (2013)
5. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The NAS parallel benchmarks. Int. J. Supercomput. Appl. **5**(3), 63–73 (1991)
6. Garcia, S., Jeon, D., Louie, C.M., Taylor, M.B.: Kremlin: rethinking and rebooting gprof for the multicore age. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 458–469. ACM (2011)
7. Govindarajan, R., Anantpur, J.: Runtime dependence computation and execution of loops on heterogeneous systems. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, pp. 1–10. IEEE Computer Society (2013)
8. Grosser, T., Groesslinger, A., Lengauer, C.: Polly - performing polyhedral optimizations on a low-level intermediate representation. Parallel Process. Lett. **22**(04), 1250010 (2012)
9. Huda, Z.U., Jannesari, A., Wolf, F.: Using template matching to infer parallel design patterns. ACM Trans. Archit. Code Optim. **11**(4), 64:1–64:21 (2015)
10. Ketterlin, A., Clauss, P.: Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In: Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 45, pp. 437–448. IEEE Computer Society (2012)
11. Kim, M., Kim, H., Luk, C.K.: Prospector: discovering parallelism via dynamic data-dependence profiling. In: Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism, HOTPAR 2010 (2010)
12. Kim, M., Kim, H., Luk, C.K.: SD3: A scalable approach to dynamic data-dependence profiling. In: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 43, pp. 535–546. IEEE Computer Society (2010)

13. Lee, S.I., Johnson, T., Eigenmann, R.: Cetus - an extensible compiler infrastructure for source-to-source transformation. In: Rauchwerger, L. (ed.) Languages and Compilers for Parallel Computing. Lecture Notes in Computer Science, vol. 2958, pp. 539–553. Springer, Heidelberg (2004)
14. Li, Z., Atre, R., Ul-Huda, Z., Jannesari, A., Wolf, F.: DiscoPoP: A profiling tool to identify parallelization opportunities. In: Niethammer, C., Gracia, J., Knüpfer, A., Resch, M.M., Nagel, W.E. (eds.) Tools for High Performance Computing 2014, 1st edn, pp. 1–10. Springer International Publishing, Switzerland (2015)
15. Li, Z., Jannesari, A., Wolf, F.: Discovery of potential parallelism in sequential programs. In: Proceedings of the 42nd International Conference on Parallel Processing, PSTI 2013, pp. 1004–1013, vol. 13. IEEE Computer Society (2013)
16. Li, Z., Jannesari, A., Wolf, F.: An efficient data-dependence profiler for sequential and parallel programs. In: Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium, IPDPS 2015, pp. 484–493 (2015)
17. Ottoni, G., Rangan, R., Stoler, A., August, D.I.: Automatic thread extraction with decoupled software pipelining. In: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 38, pp. 105–118. IEEE Computer Society (2005)
18. Rul, S., Vandierendonck, H., De Bosschere, K.: Function level parallelism driven by data dependencies. SIGARCH Comput. Archit. News **35**(1), 55–62 (2007)
19. Rul, S., Vandierendonck, H., De Bosschere, K.: A profile-based tool for finding pipeline parallelism in sequential programs. Parallel Comput. **36**(9), 531–551 (2010)
20. Serebryany, K., Potapenko, A., Iskhodzhanov, T., Vyukov, D.: Dynamic race detection with LLVM compiler. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 110–114. Springer, Heidelberg (2012)
21. Vanka, R., Tuck, J.: Efficient and accurate data dependence profiling using software signatures. In: Proceedings of the 10th International Symposium on Code Generation and Optimization, CGO 2012, pp. 186–195. ACM, New York (2012)
22. Ye, J.M., Chen, T.: Exploring potential parallelism of sequential programs with superblock reordering. In: Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems, HPCC 2012, pp. 9–16. IEEE Computer Society (2012)
23. Yu, H., Li, Z.: Multi-slicing: a compiler-supported parallel approach to data dependence profiling. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012, pp. 23–33. ACM (2012)
24. Zhang, X., Navabi, A., Jagannathan, S.: Alchemist: a transparent dependence distance profiling infrastructure. In: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2009, pp. 47–58. IEEE Computer Society (2009)
25. Zhao, B., Li, Z., Jannesari, A., Wolf, F., Wu, W.: Dependence-based code transformation for coarse-grained parallelism. In: Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores, COSMIC 2015, pp. 1–10. ACM, New York (2015)