Accelerating Data-Dependence Profiling with Static Hints

Mohammad Norouzi[1(✉)], Qamar Ilias[1], Ali Jannesari[2], and Felix Wolf[1]

[1] Technische Universitaet Darmstadt, Darmstadt, Germany
{norouzi,wolf}@cs.tu-darmstadt.de, ilias.qamar@gmail.com
[2] Iowa State University, Ames, IA, USA
jannesari@iastate.edu

Abstract. Data-dependence profiling is a program-analysis technique to discover potential parallelism in sequential programs. Contrary to purely static dependence analysis, profiling has the advantage that it captures only those dependences that actually occur during execution. Lacking critical runtime information such as the value of pointers and array indices, purely static analysis may overestimate the amount of dependences. On the downside, dependence profiling significantly slows down the program, not seldom prolonging execution by a factor of 100. In this paper, we propose a hybrid approach that substantially reduces this overhead. First, we statically identify persistent data dependences that will appear in any execution. We then exclude the affected source-code locations from instrumentation, allowing the profiler to skip them at runtime and avoiding the associated overhead. At the end, we merge static and dynamic dependences. We evaluated our approach with 38 benchmarks from two benchmark suites and obtained a median reduction of the profiling time by 62% across all the benchmarks.

1 Introduction

Data-dependence analysis is a prerequisite for the discovery of parallelism in sequential programs. Traditionally, compilers such as PLUTO [1] perform it statically with the goal of auto-parallelizing loops. However, lacking critical runtime information such as the value of pointers and array indices, purely static dependence analysis may overestimate the amount of dependences. This is why auto-parallelization has not succeeded much beyond the confines of the polyhedral model [2], a theoretical framework for the optimization and, in particular, parallelization of loops that satisfy certain constraints.

Recently, many tools [3–7] emerged that avoid some of the limits of purely static analysis. They abandon the idea of fully automatic parallelization and instead point the user to likely parallelization opportunities, based on data dependences captured at runtime. They counter the inherent input sensitivity of such a dynamic approach by running the program with several representative inputs and by providing weaker correctness guarantees, although their suggestions more than often reproduce manual parallelization strategies. In addition,

R. Yahyapour (Ed.): Euro-Par 2019, LNCS 11725, pp. 17–28, 2019.
https://doi.org/10.1007/978-3-030-29400-7_2

they observed that data dependences in frequently executed code regions that are subject to parallelization do not change significantly with respect to different inputs [4–6]. Nonetheless, high runtime overhead, caused by the need to profile memory accesses during execution, makes them hard to use. Optimizations such as sampling loop iterations for profiling [8], parallelizing the data-dependence profiler itself [5,9], and skipping repeatedly executed memory operations [10] lower the overhead only to a certain degree. To reduce the overhead more substantially, we take a fundamentally different route. Leveraging the power of prior static dependence analysis, we exclude those memory accesses from profiling whose data dependences can already be determined at compile time.

Overall, we follow a hybrid approach. First, we run a static analyzer, PLUTO in our case, to identify those data dependences that every program execution must respect. We then run the dependence profiler but refrain from instrumenting all memory-access instructions that correspond to these dependences, allowing the profiler to skip them at runtime and avoid the associated overhead. Furthermore, we transform all data dependences regardless of how they have been obtained - whether statically or dynamically - into a unified representation and merge them into one output. Here, we focus on reducing the profiling overhead. How to use the acquired data dependences to identify parallelization potential is addressed in related work [4,5,7] and beyond the scope of this paper. In a nutshell, we make the following specific contributions:

- A hybrid approach to the extraction of data dependences that combines the advantages of static and dynamic techniques
- An implementation as an extension of the data-dependence profiler of DiscoPoP [7], although our approach is generic enough to it be implemented in any data-dependence profiler
- An evaluation with 38 programs from two benchmark suites, showing a median reduction of the profiling time by 62%

The remainder of the paper is organized as follows. We discuss related work in Sect. 2. Section 3 presents our approach, followed by an evaluation in Sect. 4. Finally, we review our achievements in Sect. 5.

2 Related Work

Profiling of memory accesses is a common technique to identify data dependences [4,5,7], but suffers from high runtime overhead, not seldom causing a slowdown of a factor of 100 or more. A typical method to reduce runtime overhead is sampling [8], although it does not apply well to data-dependence profiling. A data dependence is made of two distinct memory accesses and omitting only one of them is enough to miss a dependence or introduce spurious dependences.

But there are further optimizations available to lower the profiling overhead. For example, Parwiz [4], a parallelism discovery tool, coalesces contiguous memory accesses. This lowers the profiling overhead, but only for a subset of the memory accesses. Kremlin [11], another parallelization recommender system, profiles

data dependences only within specific code regions. To save memory overhead, SD3 [5], a dependence profiler, compresses memory accesses with stride patterns. Moreover, it reduces the runtime overhead by parallelizing the profiler itself. DiscoPoP [7] is a parallelism discovery tool that includes a generic data-dependence profiler [9], which serves as the basis for our implementation. The original version of the profiler converts the program into its LLVM-IR representation, after which it instruments all memory access instructions. A runtime library tracks the memory accesses during execution. To reduce the memory and runtime overhead, it records memory accesses in a signature hash table. Moreover, it skips repeatedly executed memory operations. Like SD3, it runs multiple threads to reduce the runtime overhead further. Because of its favorable speed with an average slowdown of 86, we implemented our approach in DiscoPoP, although it is generic enough to improve the efficiency of any profiler. The main difference to the optimizations pursued in other tools is the hybrid combination of dynamic and static dependence analysis.

To obtain data dependences statically, we use PLUTO [1], an auto-parallelizing compiler for polyhedral loops. PLUTO annotates the beginning and end of a code section containing a polyhedral loop. The annotated area is called a SCoP (Static Control Part) and fulfills certain constraints. It has a single entry and a single exit point and contains only (perfectly-nested) loops with affine linear bounds [2]. With PLUTO extracting data dependences from SCoPs, we accelerate subsequent dependence profiling by excluding memory-access operations that appear in SCoPs from instrumentation, cutting the SCoP-related profiling overhead.

Another hybrid-analysis framework was proposed by Rus et al. [12]. It targets the automatic parallelization of loops whose parallelization is not obvious at compile time. Based on the results of static analysis, they formulate conditions and insert them into the source code. These conditions evaluate at runtime whether a loop can be parallelized or not. In contrast to their work, our contribution happens at a lower level, where we just collect data dependences, with the goal of increasing the profiling speed.

3 Approach

Below, we explain our hybrid approach to identify data dependences. Figure 1 shows the basic workflow. Dark boxes highlight our contribution in relation to the previously isolated static and dynamic dependence analyses. First, we extract data dependences statically. Based on these dependences, we identify memory-access instructions that can be eliminated from profiling. The precise elimination algorithm is explained in Sect. 3.1. The dynamic data-dependence analysis will then skip these instructions during the profiling process. Finally, we transform all data dependences we have found – whether of static or dynamic origin – into a unified representation, whose details we describe in Sect. 3.2, and merge them into a single output file. Before we proceed to the evaluation in Sect. 4, we also discuss the relation between the set of data dependences extracted by the hybrid and the purely dynamic approach in Sect. 3.3.

Algorithm 1. Exclusion of memory-access instructions from instrumentation

```
for each function f ∈ program do
    SCoPSet = PLUTO.getSCoPs(f)
    for each SCoP s ∈ SCoPSet do
        varSet = getVariables(s)
        for each variable var ∈ varSet do
            instrument(firstLoadInst(var,s))
            instrument(lastLoadInst(var,s))
            instrument(firstStoreInst(var,s))
            instrument(lastStoreInst(var,s))
```

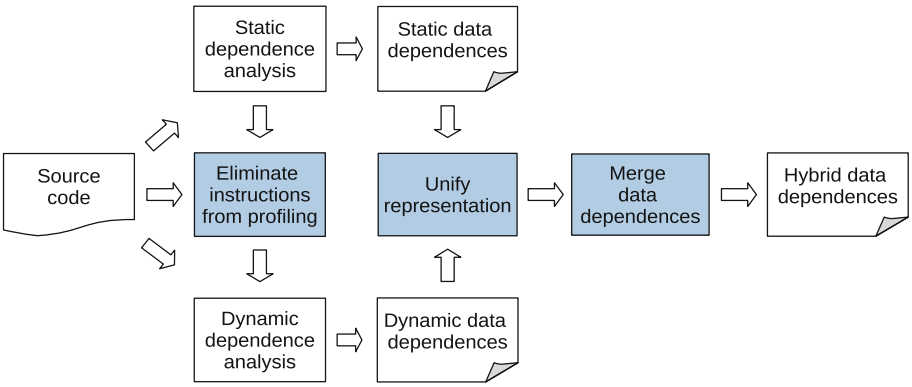


Fig. 1. The workflow of our hybrid data-dependence analysis. Dark boxes show our contributions.

3.1 Reduced Instrumentation

We exclude specific memory-access instructions from instrumentation that appear inside source code locations from which PLUTO can extract data dependences statically. Algorithm 1 shows the details and can be best understood when following the examples in Fig. 2.

We first let PLUTO annotate the target program with SCoP directives. In the example, lines 10 and 65 contain the annotations. Then, we traverse the source code and mark the variables inside a SCoP. For each variable, we determine its boundary instructions: the first and the last read and write operation. The first read and write of the array variable a appear in lines 15 and 20 and the last read and write in lines 55 and 60, respectively. We instrument only these boundary instructions and mark all other memory-access operations on a variable for exclusion. The dark box shows the section to be left out for variable a.

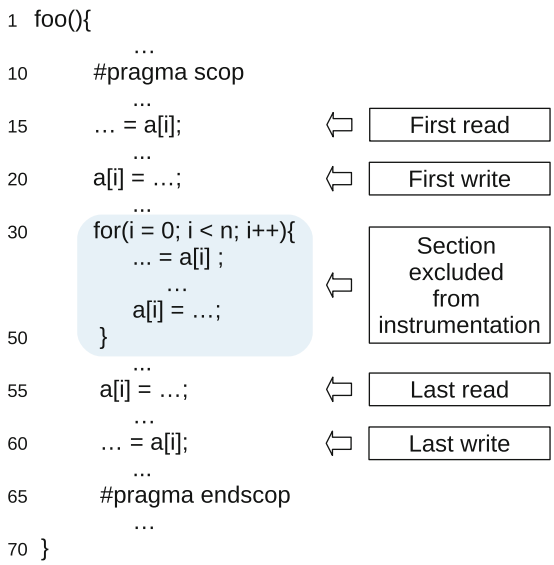


Fig. 2. A SCoP and the memory-access instructions excluded from instrumentation.

If a profiler fails to instrument one of the boundary instructions, it will report false positive and negative data dependences. False positives are data dependences that are reported but do not exist in the program. Conversely, false negatives are data dependences that exist in the program but are not reported by the profiler. False positive or negative data dependences that are reported when the boundary instructions are skipped can adversely influence parallelization recommendations that span across multiple SCoPs. The opportunities inside a SCoP, however, are not affected because PLUTO extracts all the data dependences relevant to its parallelization. We profile the boundary instructions not to miss any data dependences that a purely dynamic method would obtain. In addition, this avoids false positives and negatives and helps assess parallelization potential that stretches across SCoPs. Figures 3a and b show situations that create false negatives. If we exclude the first read in Fig. 3a, the read-after-write (RAW) dependence between the first read inside the SCoP and the last write preceding it is not reported. If the first write is eliminated, two types of false negatives will happen: on the one hand, the write-after-read (WAR) between the first write and the read before the SCoP (Fig. 3b), and the write-after-write (WAW) between the first write and the write before the SCoP on the other. Moreover, if we do not instrument the last read operation on a variable (Fig. 3a), the WAR between the last read and the write after the SCoP will be ignored. If we exclude the last write, however, dependences of two types will not be reported: the RAW between the last write and the read after the SCoP (Fig. 3b) and the WAW between the last write and the write after the SCoP. Of course, these considerations apply only to live-out loop variables that are accessed both inside and outside the loop.

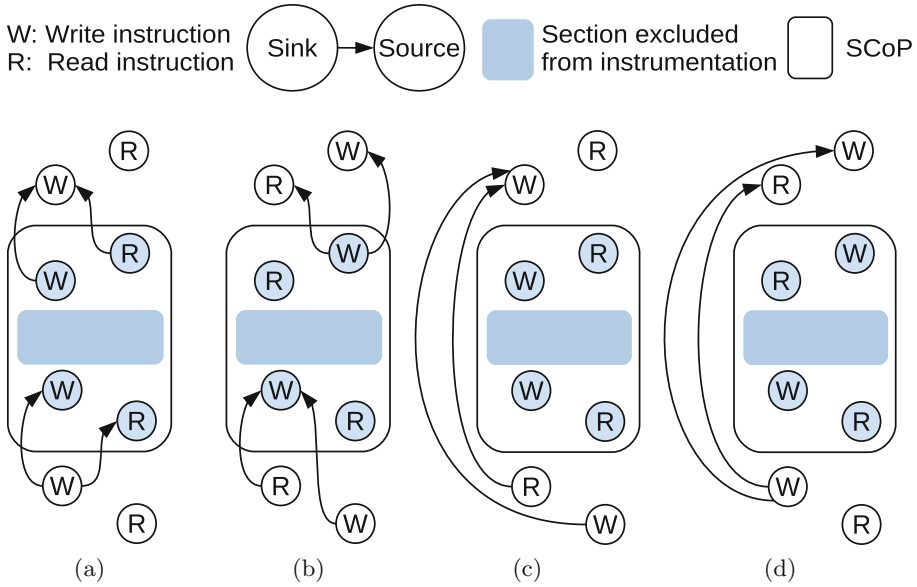


Fig. 3. Situations that create false negative (a and b) and false positive (c and d) data dependences when the first and last read and write instructions in a SCoP are not instrumented (shown in dark circles).

Figures 3c and d show situations that create false positives. Three types of false positives are reported if the boundary instructions are not instrumented. Figure 3c shows a false positive RAW between the last write preceding the SCoP and the first read succeeding it. Figure 3d shows a WAR that will be reported falsely between the last read before the SCoP and the first write after it. Finally, the write operations before and after the SCoP, in both figures, create false positive WAW dependences.

Our analysis excludes memory-access instructions that exist in polyhedral loops. In the worst case, if there are no polyhedral loops in a program, all instructions are instrumented and thus, the hybrid approach falls back to the purely dynamic approach. The overhead of the hybrid approach, in this case, is not reduced in comparison with the purely dynamic approach.

```
1        1:60 NOM {RAW 1:60|i} {WAR 1:60|i}
2        1:63 NOM {RAW 1:59|temp1} {RAW 1:67|temp1}
3        1:64 NOM {RAW 1:60|i}
4        1:65 NOM {RAW 1:59|temp1} {RAW 1:67|temp1} {WAR 1:67|temp2}
5        1:66 NOM {RAW 1:59|temp1} {RAW 1:65|temp2} {RAW 1:67|temp1}
6        1:67 NOM {RAW 1:65|temp2} {WAR 1:66|temp1}
7        1:70 NOM {RAW 1:67|temp1}
8        1:74 NOM {RAW 1:41|block}
```

Fig. 4. A fragment of unified data dependences extracted from a sequential program.

Algorithm 2. Transformation of data dependences identified by PLUTO into the unified representation.

```
for each SCoP scop ∈ SCoPSet do
    fileID = findFileID(scop)
    depSet = PLUTO.getDeps(scop)
    for each dependence dep ∈ depSet do
        varName = getVarName(dep)
        sourceLine = findSourceLine(dep)
        sinkLine = findSinkLine(dep)
        depType = getDataType(dep)
        print(fileID : sinkLine depType fileID : sourceLine|varName)
```

3.2 Unified Representation

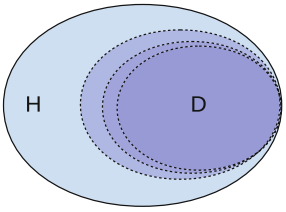
A data dependence exists if the same memory location is accessed twice and at least one of the two accesses is a write. Without loss of generality, one of the accesses occurs earlier and one later during sequential execution. To store data dependences, static and dynamic tools use different representations, which we unify in this paper. A sample of unified data dependences is shown in Fig. 4. We write a data dependence as a triple `<sink, type, source>`. `type` is the dependence type (i.e., RAW, WAR, or WAW). Because they are irrelevant to parallelization and, strictly speaking, do not even constitute a dependence according to our definition above, most data-dependence profilers do not profile read-after-read (RAR) dependences, which is why we do not report them either. `sink` and `source` are the source code locations of the later and the earlier memory access, respectively. `sink` is specified as a pair `<fileID:lineID>`, while `source` is specified as a triple `<fileID:lineID|variableName>`. We assign a unique fileID to each file in a program. Existing profilers, including Parwiz, DiscoPoP, SD3, and Intel Pin [13], already display data dependences in terms of source-code files, line numbers, and variable names. Thus, transforming their output to our unified representation requires little effort.

PLUTO, in contrast, assigns a unique ID to each source-code statement in a SCoP and reports data dependences based on these IDs. We use Algorithm 2 to transform the output of PLUTO into the unified representation. First, we find the fileID of each SCoP, before we retrieve the set of data dependences in a SCoP from PLUTO. We use the IDs to identify the statements in which the source and sink of a data dependence appear. Then, we read the source code of the file to find the line number of the statements. Finally, we determine the type of the data dependence and the name of the variable involved in it. Unfortunately, PLUTO does not report data dependences for loop index variables. We apply use-def analysis to statically identify the types of data dependences for the indices appearing in SCoPs. We cannot run this analysis for an entire program because the code beyond the SCoPs may contain pointers that cannot be tracked with use-def analysis. At the end, we transform the dependences for the loop indices into the unified representation.

Once we have collected all data dependences using our portfolio of static and dynamic methods, we merge them into a joint ASCII file. To reduce the size of the output, we compress the dependence data, merging all dependences with the same sink into a single line. Finally, we sort the dependences based on the sink. The result can be used by parallelism discovery tools to find parallelization opportunities.

3.3 Hybrid vs. Dynamic Data Dependences

Now, we take a deeper look into the relationship between the set of data dependences extracted by our hybrid approach in comparison to the one produced by purely dynamic analysis, which is illustrated in Fig. 5. To better understand this relation, let us consider the listings in the figure. In Fig. 5b, both loops meet the constraints of the polyhedral model. PLUTO finds data dependences in those loops and, thus, our hybrid approach excludes the whole conditional block from profiling. Profilers might execute either the if or the else branch, depending on the condition $k < average$, and extract dependences only in the executed part. Only running the program with two different inputs, each of them causing the program to take a different branch, however, would allow a profiler to identify dependences in both parts. In general, the set of hybrid data dependences is



(a)

```
if ( k < average ){
    for ( i = 0; i < n; i++ ) {
        q[i] = q[i] + A[i] * p[i];
    }
} else {
    for ( j = 0; j < n; j++ ) {
        s[j] = s[j] + r[j] * A[j];
    }
}
```

(b) Both loops are polyhedral

```
if ( k < average ){
    for ( i = 0; i < n; i++ ) {
        w = a[f(i)];
        a[g(i)] = v;
    }
} else {
    for ( j = 0; j < n; j++ ) {
        s[j] = s[j] + r[j];
    }
}
```

(c) Only the loop in the else part is polyhedral

```
if ( k < average ){
    for ( i = 0; i < n; i++ ) {
        w = b[f(i)];
        a[g(i)] = v;
    }
} else {
    for ( j = 0; j < n; j++ ) {
        d = d * z[colidx[j]];
    }
}
```

(d) Neither loops are polyhedral

Fig. 5. (a): The relation between dynamic and hybrid data dependences. H includes data dependences that are identified via hybrid analysis. D contains data dependences identified via dynamic analysis with a finite set of inputs. (b) and (c): Two examples where $D \subseteq H$. (d) One example where $H = D$.

therefore a superset of the set of purely dynamic data dependences (i.e., $D \subseteq H$). Figure 5c shows a similar case where the set of hybrid dependences contains the set of dynamic dependences (i.e., $D \subseteq H$). There are two loops, but only the one in the else branch is polyhedral. Again, profilers might miss the dependences in the polyhedral loop if none of the provided inputs makes the program go through the else branch. Finally, in Fig. 5d, neither loop is polyhedral. PLUTO does not extract dependences from either loop and, thus, our approach does not exclude any instructions from instrumentation. In this case, the set of dependences identified by our approach is equal to the set of dependences detected by purely dynamic analysis (i.e., $H = D$).

In theory, H and D would be different for a program only if a polyhedral loop recognized by PLUTO was never executed. However, this condition happens rarely in practice because polyhedral loops constitute hotspots, that is, they consume major portions of the execution time. As several authors have shown [4–7], such regions are usually always visited—regardless of the specific input. Exceptions include, for example, erroneous inputs that cause the program to terminate prematurely.

4 Evaluation

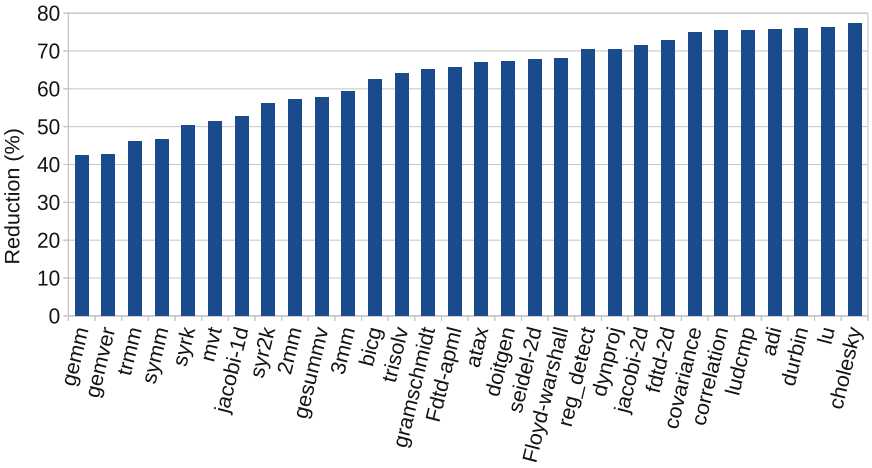
We conducted a range of experiments to evaluate the effectiveness of our approach. Our test cases are the NAS Parallel Benchmarks 3.3.1 [14] (NPB), a suite of programs derived from real-world computational fluid-dynamics applications, and Polybench 3.2 [15], a test suite originally designed for polyhedral compilers. We compiled the benchmarks using clang 3.6.8, which is also used by the DiscoPoP profiler for program instrumentation. We ran the benchmarks on an Intel(R) Xeon(R) CPU E5-2650 2.00 GHz with 32Gb of main memory, running Ubuntu 14.04 (64-bit edition). To profile the benchmarks, we used the inputs the benchmark designers provided alongside the programs. Our evaluation criteria are the completeness of the data dependences in relation to purely dynamic profiling and the profiling time. We compared the set of data dependences identified by the profiler with and without prior static analysis. Because the entire source code of the benchmarks was visited during the execution with the given inputs, we observed no difference in the reported data dependences. Following the arguments of Sect. 3.3, however, we believe that higher code-coverage potential makes our approach generally less input sensitive than purely dynamic methods, a claim we want to substantiate in a follow-up study.

To measure how much our hybrid method speeds up the profiler, we ran the benchmarks first with the vanilla version of the DiscoPoP profiler. We ran each benchmark five times in isolation, recorded the median of the execution times, and declared it as our baseline. Then, we profiled the benchmarks with the enhanced version of the profiler, taking advantage of prior static analysis and reduced instrumentation. Again, we ran each benchmark fives times in isolation and calculated the median of the execution times, which we then compared with

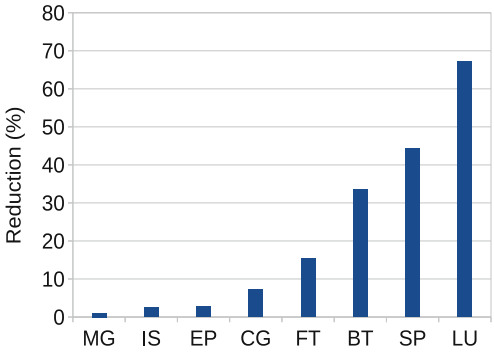
our baseline. Table 1 summarizes the relative slowdown caused by the purely dynamic vs. the hybrid approach for the two benchmark suites. Finally, Fig. 6 presents the relative overhead reduction for each benchmark.

Table 1. Relative slowdown caused by standard DiscoPoP vs. the hybrid approach.

Benchmark suites	Standard DiscoPoP			Hybrid approach		
	Min	Max	Median	Min	Max	Median
Polybench	37.57	144.98	71.67	14.26	47.84	24.42
NPB	18.60	130.50	82.67	18.11	121.09	63.18
All	18.60	144.98	72.28	14.26	121.09	27.32



(a) Polybench



(b) NPB

Fig. 6. Profiling-time reduction relative to the standard DiscoPoP profiler.

Whether we can reduce the profiling time of a benchmark depends on its computational pattern. In theory, the more work is done in polyhedral loops, the more effective our method will be. If a program does not contain such loops, we fail to reduce the profiling overhead significantly. Notably, our method lowered the profiling time in all test cases. For four benchmarks in NPB, namely EP, IS, CG, and MG, we observed only small improvements because there we could not exclude many instructions from profiling. For all other benchmarks, our approach was highly effective. We noticed that removing write operations influences the profiling time more than removing reads; when profiling a write operation we need to look for both WAW and WAR dependences, whereas we only need to look for RAW dependence when profiling a read operation. In general, however, the number of excluded write instructions is less than the number of reads. Overall, we achieved a median profiling-time reduction by 62%. The size of the dependence files generated by the hybrid approach for these benchmarks is in the order of kBs.

5 Conclusion

Our hybrid approach to data-dependence analysis allows the profiler to skip code locations whose dependences can be extracted statically. Nevertheless, not to miss any data dependence a purely dynamic method would obtain, we still profile memory operations at the boundaries of these locations, capturing data dependences that point into and out of them. We implemented our approach in a state-of-the-art data-dependence profiler and achieved a median reduction of the profiling time by 62% across a large set of benchmarks, making it far more practical than before. Faster profiling will enable the DiscoPoP framework to identify parallelism in larger and longer running programs. However, in principle, our method can serve as frontend to any data-dependence profiler. Our specific PLUTO-based implementation focuses on polyhedral loops, which opens up two possible avenues to future work. First, we could try to expand the coverage of the static analysis, exploring dependences outside polyhedral loops. Second, since polyhedral loops can be easily parallelized statically, we could make parallelism discovery tools, whose strength lies in more unstructured parallelism outside such loops, aware of them and make them cooperate with polyhedral tools also on the level of parallelism discovery and code transformation, exploiting their advantages while filling their gaps.

Acknowledgement. This work has been funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project. Additional support has been provided by the German Research Foundation (DFG) through the Program Performance Engineering for Scientific Software and the US Department of Energy under Grant No. DE-SC0015524.

References

1. Bondhugula, U.: Pluto - an automatic parallelizer and locality optimizer for affine loop nests (2015). http://pluto-compiler.sourceforge.net/. Accessed 13 June 2019

2. Benabderrahmane, M.W., Pouchet, L.N., Cohen, A., Bastoul, C.: The polyhedral model is more widely applicable than you think. In: Proceedings of the Conference on Compiler Construction. CC 2010, Paphos, Cyprus, pp. 283–303, March 2010
3. Wilhelm, A., Cakaric, F., Gerndt, M., Schuele, T.: Tool-based interactive software parallelization: a case study. In: Proceedings of the International Conference on Software Engineering. ICSE 2018, Gothenburg, Sweden, pp. 115–123, June 2018
4. Ketterlin, A., Clauss, P.: Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In: Proceedings of the International Symposium on Microarchitecture. MICRO 1945, Vancouver, B.C., Canada, pp. 437–448, December 2012
5. Kim, M., Kim, H., Luk, C.K.: SD3: a scalable approach to dynamic data-dependence profiling. In: Proceedings of the International Symposium on Microarchitecture. MICRO 1943, Atlanta, GA, USA, pp. 535–546, December 2010
6. Norouzi, M., Wolf, F., Jannesari, A.: Automatic construct selection and variable classification in OpenMP. In: Proceedings of the International Conference on Supercomputing. ICS 2019, Phoenix, AZ, USA, pp. 330–342, June 2019
7. Li, Z., Atre, R., Huda, Z.U., Jannesari, A., Wolf, F.: Unveiling parallelization opportunities in sequential programs. J. Syst. Softw. **117**(1), 282–295 (2016)
8. Jimborean, A., Clauss, P., Martinez, J.M., Sukumaran-Rajam, A.: Online dynamic dependence analysis for speculative polyhedral parallelization. In: Wolf, F., Mohr, B., an Mey, D. (eds.) Euro-Par 2013. LNCS, vol. 8097, pp. 191–202. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40047-6_21
9. Li, Z., Jannesari, A., Wolf, F.: An efficient data-dependence profiler for sequential and parallel programs. In: Proceedings of the International Parallel and Distributed Processing Symposium. IPDPS 2015, Hyderabad, India, pp. 484–493, May 2015
10. Li, Z., Beaumont, M., Jannesari, A., Wolf, F.: Fast data-dependence profiling by skipping repeatedly executed memory operations. In: Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing. ICA3PP 2015, Zhangjiajie, China, pp. 583–596, November 2015
11. Garcia, S., Jeon, D., Louie, C.M., Taylor, M.B.: Kremlin: rethinking and rebooting gprof for the multicore age. In: Proceedings of the Conference on Programming Language Design and Implementation. PLDI 2011, pp. 458–469, June 2011
12. Rus, S., Rauchwerger, L., Hoeflinger, J.: Hybrid analysis: static & dynamic memory reference analysis. Int. J. Parallel Prog. **31**(4), 251–283 (2003)
13. Intel: Pin - a dynamic binary instrumentation tool (2010). https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool. Accessed 13 June 2019
14. Bailey, D.H., et al.: The NAS parallel benchmarks. Int. J. Supercomput. Appl. **5**(3), 63–73 (1991)
15. Pouchet, L.N.: Polyhedral suite (2011). http://www.cs.ucla.edu/pouchet/software/polybench/. Accessed 13 June 2019