Skipping Non-essential Instructions Makes Data-Dependence Profiling Faster

Nicolas Morew[1], Mohammad Norouzi[1], Ali Jannesari[2], and Felix Wolf[1]

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

Abstract. Data-dependence profiling is a dynamic program-analysis technique to discover potential parallelism in sequential programs. Unlike purely static analysis, which may overestimate the number of dependences because it does not know many pointers values and array indices at compile time, profiling has the advantage of recording data dependences that actually occur at runtime. But it has the disadvantage of significantly slowing down program execution, often by a factor of 100. In our earlier work, we lowered the overhead of data-dependence profiling by excluding polyhedral loops, which can be handled statically using certain compilers. However, neither does every program contain polyhedral loops, nor are statically identifiable dependences restricted to such loops. In this paper, we introduce an orthogonal approach, focusing on data dependences between accesses to scalar variables - across the entire program, inside and outside loops. We first analyze the program statically and identify memory-access instructions that create data dependences that would appear in any execution of these instructions. Then, we exclude these instructions from instrumentation, allowing the profiler to skip them at runtime and avoid the associated overhead. We evaluate our approach with 49 benchmarks from three benchmark suites. We improved the profiling time of all programs by at least 38%, with a median reduction of 61% across all the benchmarks.

1 Introduction

Data-dependence analysis is an essential step in the parallelization of sequential programs. Auto-parallelizing compilers [1–3] perform the analysis purely statically. They may overestimate the amount of data dependences because critical information such as the value of pointers and array indices are unknown at compile time. This is why auto-parallelization based on purely static analysis has not gained much success beyond the parallelization of loops that satisfy certain constraints.

Another group of tools [4–8] avoid the limitations of purely static analysis using a dynamic method. They detect parallelization opportunities based on data dependences captured at runtime. Running the program with several representative inputs, they counter the inherent input sensitivity of dynamic data-dependence analysis, also exploiting that data dependences in frequently executed code regions that are subject to parallelization do not change significantly with respect to different inputs [5–7]. These tools provide weaker correctness guarantees, although their suggestions more than often reproduce manual parallelization strategies.

Nonetheless, the tools have a high runtime overhead which is caused by profiling every memory access in the program. Many optimizations such as parallelizing the data-dependence profiler itself [6, 9] and skipping repeatedly executed memory operations [10] have been proposed to lower the overhead. In addition, taking a fundamentally different route, we recently introduced a hybrid approach [11] to data-dependence analysis. The approach exploited static analysis tools to extract data dependences in loops that follow the constraints of the polyhedral model [12] and profiled only memory accesses outside those loops. This reduced the profiling overhead significantly, but only for programs containing such loops.

However, only few loops are polyhedral. The strict conditions they have to satisfy make it hard for programmers to write a loop in the polyhedral form. More importantly, many data dependences that can be identified statically do not belong to such loops. In this paper, we introduce a method that is orthogonal to our earlier work. Now, we concentrate on static data dependences between accesses to scalar variables—across the entire source code, inside and outside loops. We first identify the memory instructions that belong to these dependences. Then, we run our dependence profiler, but without instrumenting these instructions, allowing the profiler to skip them at runtime and avoid their associated overhead. Eliminating instructions that can belong to all types of loops (e.g., polyhedral, canonical, or non-canonical) or functions (e.g., recursive or non-recursive), our approach is able to reduce the profiling overhead for a wide range of programs. Finally, we merge data dependences extracted statically or dynamically into one output. Here, our goal is to decrease the profiling overhead. Finding parallelization opportunities based on the identified data dependences is described in related work [5–8] and outside the scope of this paper. In summary, we make the following specific contributions:

– A hybrid technique to data-dependence analysis that combines the advantages of static and dynamic techniques. Contrary to our earlier work that excluded polyhedral loops from profiling, we now skip instructions that create statically-identifiable data dependences for scalar variables in all types of loops and functions, reducing the profiling overhead for a wider range of programs.
– An implementation as an extension of the data-dependence profiler of DiscoPoP [8], although our approach is generic enough to be implemented in any data-dependence profiler.
– An evaluation with 49 programs from three benchmark suites, reducing the profiling time by at least 38%, with a median improvement of 61%.

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

2 Related Work

A great deal of research has been made in the field of data-dependence analysis [1–8]. Most approaches focus on either static or dynamic analysis techniques, with only a few attempting to combine them.

autoPar [2] is a static analysis tool which can parallelize array-based loops [13]. Applying a set of loop transformations such as fusion, fission, interchange, unrolling, and blocking, autoPar checks whether or not a data dependence in a loop can be eliminated. If all dependences in the loop are eliminated, it suggests parallelizing the loop. Contrary to autoPar, which finds data dependences only in specific loops, our method identifies data dependences in all types of loops and functions. PLUTO [1], is another auto-parallelizing compiler which detects data dependences in polyhedral loops [12]. TaskMiner [3] is a static analysis tool which translates programs containing recursive functions into their parallel versions. It exploits LLVM data-dependence analysis to identify dependences. Like TaskMiner, our approach uses LLVM and its features to identify data dependences involving scalar variables. Contrary to TaskMiner, which extracts data dependences only in recursive functions, we identify data dependences in any functions and in loops. In general, static analysis techniques may overestimate the number of dependences because they lack critical runtime information at compile time such as the values of pointers and array indices.

Avoiding the limitations of purely static analysis, many tools [5–8] capture data dependences during program execution. They profile memory accesses, which imposes huge runtime overhead. SD3 [6] is a data-dependence profiler which decreases the overhead by parallelizing the profiler itself. DiscoPoP [8] is a parallelism discovery tool that contains a data-dependence profiler [9]. The profiler is based on LLVM and transforms the program into its LLVM-IR representation. It instruments all memory-access instructions with runtime library calls that track memory accesses at runtime. It skips repeatedly executed memory operations and, like SD3, runs multiple threads to reduce the overhead. Nonetheless, dependence profiling significantly slows down program execution, sometimes by more than a factor of 100.

Recently, we introduced a hybrid technique [11] for data-dependence analysis. The technique is called DiscoPoP+ and uses the profiler of DiscoPoP as the basis of its implementation. It first runs PLUTO to statically identify data dependences in polyhedral loops. Then, it excludes the loops from instrumentation, profiling only data dependences outside the loops. At the end, it merges static and dynamic dependences. It reduces the profiling overhead significantly, but only for programs containing polyhedral loops. Our approach, however, accelerates the profiling of all types of loops and functions. Based on the control flow graph of the program, it statically identifies data dependences of scalar

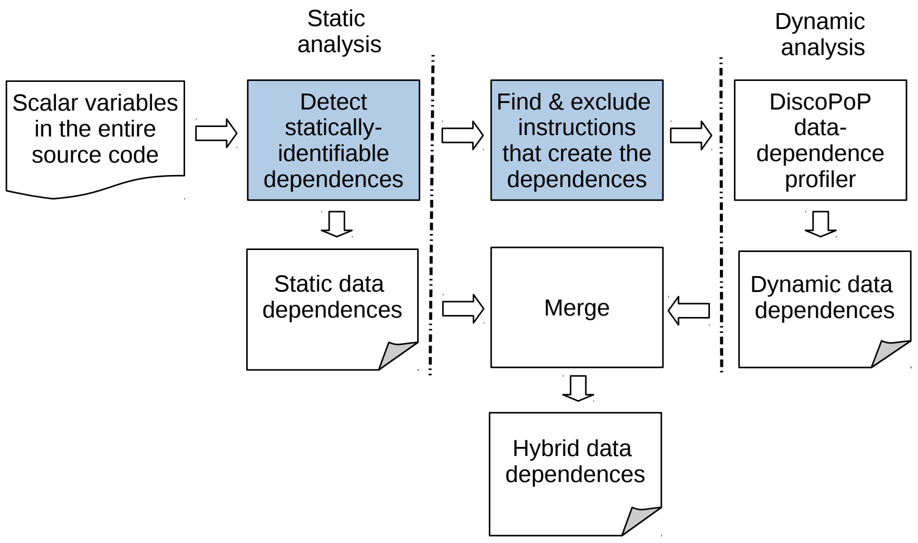

Static analysis
Dynamic analysis
Scalar variables in the entire source code
Detect statically-identifiable dependences
Find & exclude instructions that create the dependences
DiscoPoP data-dependence profiler
Static data dependences
Merge
Dynamic data dependences
Hybrid data dependences

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

variables, not including passed-by-reference parameters and pointers. It then identifies memory instructions that create the dependences and excludes them from instrumentation. Skipping such instructions, which may appear inside and outside loops, our method allows the reduction of the profiling overhead for a wide range of programs.

Another hybrid-analysis framework was proposed by Sampaio et al. [14]. Their goal is providing theoretical and practical foundations to apply aggressive loop transformations. They apply static alias and dependence analysis and provide their results to an optimizer. The optimizer, instead of filtering out invalid transformations, performs transformations believed to reduce the execution time. It then generates fast and precise tests to validate at runtime whether the transformations can be taken. In contrast to their work, our contribution happens at a lower level, where we obtain dependences with the aim to accelerate data-dependence profiling.

3 Approach

Below, we explain our hybrid method to the identification of data dependences. Figure 1 shows the basic workflow. Dark boxes highlight our contribution in relation to DiscoPoP+, our earlier hybrid approach, PLUTO, a static analyzer, and DiscoPoP, a dynamic data-dependence profiler.

DiscoPoP+ relies on PLUTO to extract data dependences statically. Unlike DiscoPoP+, which statically identifies data dependences only in polyhedral loops, we detect the dependences for scalar variables, excluding aliases, in the

```
1 void foo(int x){
2  int y = 0;
3  int *p = &y;
4  *p = 4;
5  bar(&x);
6 }
```

Fig. 2. A program containing only aliased variables.

entire source code. In addition, we find memory-access instructions that create the dependences and exclude them from instrumentation. Below, in Section 3.1, we present the details of our method. The dynamic data-dependence analysis will then skip these instructions during the profiling process. Finally, we merge all dependences we have found—whether of static or dynamic origin—into a single output file. Before we proceed to the evaluation in Section 4, we also discuss the relation between the set of dependences extracted by our approach and the purely dynamic technique in Section 3.2.

3.1 Data-dependence detection and instruction identification

We eliminate a memory-access instruction from profiling under certain conditions. They guarantee that the instruction creates only statically-identifiable data dependences and thus, we can safely omit it, without missing any data dependences that a purely dynamic analysis may capture at runtime.

The first condition is that the target address of a memory instruction must be predictable statically. We use Algorithm 1 to detect memory addresses that comply with the condition. Figure 2 serves as an illustrating example.

The static analysis we conduct in this paper does not cross function boundaries. This is why we continue profiling memory instructions of variables that create data dependences whose sink and source appear in different functions. Nevertheless, we will investigate the analysis of dependences between functions in the future. According to our algorithm, we first look for memory allocation instructions in a function. We retrieve the symbolic address from an allocation instruction and add it to the set of statically-predictable addresses. In Figure 2, the set includes initially the address of variables x, y, and p. Then, we look for call and store instructions. We exclude the addresses that are passed by reference to functions; they may create data dependences that cannot be identified statically. In the figure, a reference to variable x is passed to function bar at line 5. It means that we cannot exclude memory-access instructions of variable x from profiling and, thus, we remove the symbolic address of x from the set of static addresses. In addition, pointer variables create data dependences which may not be identified statically. According to Algorithm 1, we detect a pointer variable if a store instruction assigns the address of a variable to another variable. We remove the symbolic address of a pointee from the set of static addresses. In the figure, the address of variable y is assigned to variable p by the implicit store

Algorithm 1: Finding memory addresses that are statically predictable.

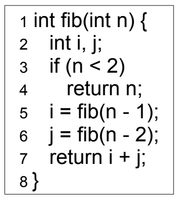
```
$staticAddrs = \{\}$
**for** ***each*** $instruction\ I \in function\ F$ **do**
    **if** $I.isAlloca()$ **then**
        $addr = I.getMemAddr()$
        $staticAddrs.insert(addr)$
**for** ***each*** $instruction\ I \in function\ F$ **do**
    **if** $I.isCall()$ **then**
        $params = I.getParams()$
        **for** ***each*** $param\ p \in params$ **do**
            **if** $p.isPassedByReference()$ **then**
                $addr = p.getMemAddr()$
                $staticAddrs.remove(addr)$
    **else if** $I.isStore()$ **then**
        $var = I.storedVar()$
        **if** $var.isMemAddr()$ **then**
            $pointeeVar = I.getPointee()$
            $staticAddrs.remove(var)$
            $staticAddrs.remove(pointeeVar)$
```

instruction at line 3. All memory instructions of variable y should be profiled and, therefore, we discard them from further analysis.

In Figure 2, most variables are aliased via pointers or references. In practice, we rarely find programs that contain only aliased variables. Figure 3a shows function fib from BOTS [15]. There, we can skip profiling memory instructions of all variables, namely, i, j, n, and an implicit variable retval, which saves the return value because we can identify data dependences between their accesses statically. Figures 3b to 3d demonstrate the analyses that we perform to extract data dependences statically, using function fib as an example.

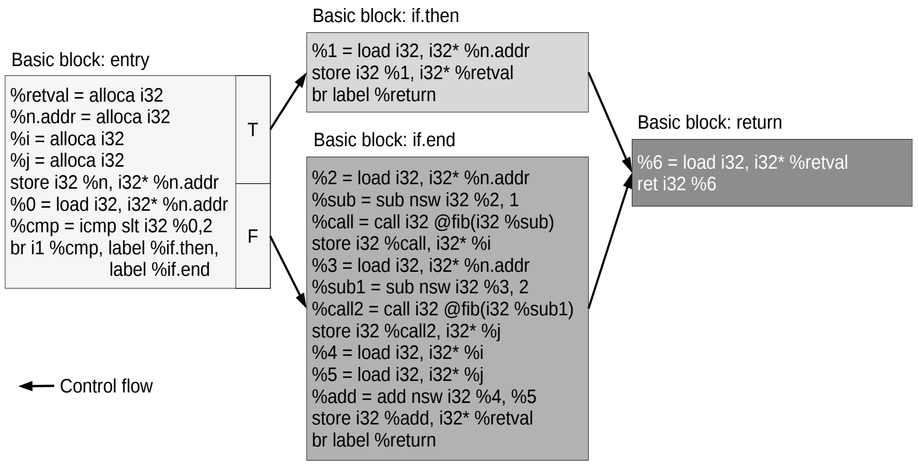
First, we convert the program into its LLVM-IR representation and generate the control flow graph (CFG) of the program. The CFG of function fib is shown in Figure 3b. The CFG contains many instructions that are irrelevant to the data-dependence analysis. We generate a memory-access CFG (MCFG) which has the same structure as the CFG but contains only memory-access instructions. Henceforth, we briefly refer to MCFG as memory-access graph or simply as graph if the context allows it. Figure 3c shows the memory-access graph of function fib.

We traverse the graph to extract data dependences statically. Algorithm 2 shows how. Figure 3d illustrates the dependences that we extract from the memory-access graph of fib. According to the algorithm, we use two recursive functions to traverse the graph of each function in the source code. First, we pass the return node in the graph to function findDepsFor. The function recursively



```
int fib(int n) {
  int i, j;
  if (n < 2)
    return n;
  i = fib(n - 1);
  j = fib(n - 2);
  return i + j;
}
```

(a) Function fib from BOTS. The memory address of all variables are statically predictable



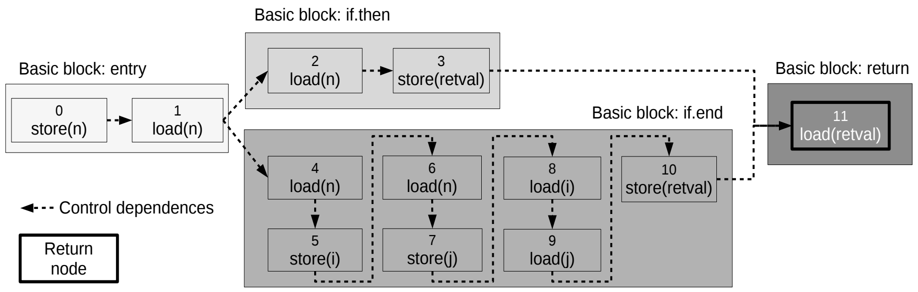
```
%retval = alloca i32
%n.addr = alloca i32
%i = alloca i32
%j = alloca i32
store i32 %n, i32* %n.addr
%0 = load i32, i32* %n.addr
%cmp = icmp slt i32 %0,2
br i1 %cmp, label %if.then,
                  label %if.end
```

```
%1 = load i32, i32* %n.addr
store i32 %1, i32* %retval
br label %return
```

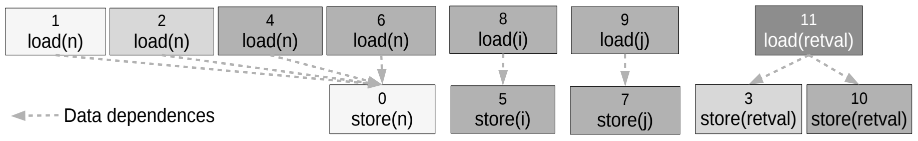
```
%2 = load i32, i32* %n.addr
%sub = sub nsw i32 %2, 1
%call = call i32 @fib(i32 %sub)
store i32 %call, i32* %i
%3 = load i32, i32* %n.addr
%sub1 = sub nsw i32 %3, 2
%call2 = call i32 @fib(i32 %sub1)
store i32 %call2, i32* %j
%4 = load i32, i32* %i
%5 = load i32, i32* %j
%add = add nsw i32 %4, %5
store i32 %add, i32* %retval
br label %return
```

```
%6 = load i32, i32* %retval
ret i32 %6
```

(b) Control-flow graph of fib



(c) Memory-access graph of fib



(d) Data dependences that our method extracts from fib

Fig. 3. How we obtain data dependences statically.

Algorithm 2: Traversing the graph of a function to extract data dependences.

```
Input: I: Return node in the memory-access graph of a function
Function findDepsFor(node I):
    if $I.isEntry()||I.isVisited()$ then
        return;
    for each node J directly preceding I do
        checkDepsBetween(I,J);
        findDepsFor(J);
Function checkDepsBetween(node I, node J):
    if $J.isEntry()$ then
        return;
    if $J.getMemAddr() == I.getMemAddr()$ then
        if $J.isStore()||I.isStore()$ then
            $addDataDeps(I, J)$;
            return;
        else
            $checkForRARDep()$;
    for each node K directly preceding J do
        if $!K.isVisited()$ then
            $K.isVisited = true$
            checkDepsBetween(I,K);
```

iterates over all nodes preceding the return node and calls function checkDepsBetween to look for dependences between the return node and its preceding nodes. It performs the same process for all other nodes until it has found dependences for all nodes. Function checkDepsBetween checks the memory addresses of the two nodes that it receives and, if they are equal and one of them is a store operation, creates a data dependence edge between the nodes. Considering the control flow, we determine the type of an identified data dependence, that is, whether it must be classified as read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW). In Figure 3c, the value of variable i is read in node 8. The value was previously stored in node 5. Figure 3d shows the data dependence that our approach adds between the nodes. The type of the dependence is RAW because the value of i is read after it is written.

We do not report read-after-read (RAR) dependences, although we identify them. This dependence type is irrelevant to the parallelization and, strictly speaking, does not even constitute a dependence. Most data-dependence profilers do not report them either. However, instrumenting memory-access instructions relevant to RAR dependences adds to the profiling overhead. If we prove during the static analysis that an instruction is only involved in RAR dependences, we can safely omit the instruction from profiling, without violating the complete-

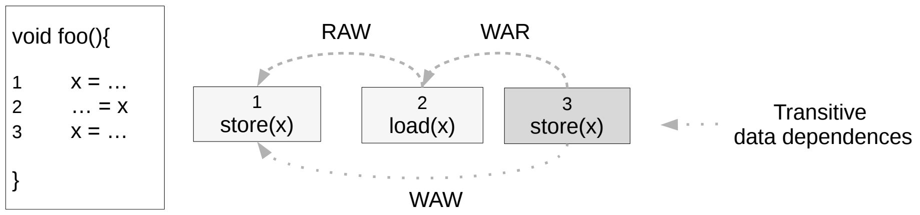


Fig. 4. A transitive data dependence.

ness of data dependences captured by purely dynamic analysis. In Algorithm 2, function checkForRARDep determines whether a memory address is only read in a function. In function fib in Figure 3a, variable n creates only RAR dependences after its memory initialization. We skip profiling all of its memory-access instructions and do not report its RAR data dependences.

We check the dependences between a node and all other nodes preceding it in the memory-access graph of a function. We repeat the process for all functions in a program. The worst-case complexity of our analysis $O(f \cdot n^2)$, where f is the number of functions and n is the maximum number of memory instructions in a function. However, given that during execution many instructions are executed many times, the overhead of the static pre-analysis, which usually takes in the order of minutes, is small in comparison to the profiling overhead the affected instructions would cause. Moreover, our analysis excludes memory-access instructions that can be safely removed during the static analysis. In the worst case, if there are no such instructions in a program, all instructions are instrumented and our approach falls back to the purely dynamic technique. In this case, we cannot reduce the profiling overhead.

In the end, we merge all the data dependences that we have identified using our portfolio of static and dynamic methods into a joint ASCII file. Furthermore, we compact the dependence data, combining all dependences with the same sink into a single line. The result can be used by parallelism discovery tools to find parallelization opportunities.

3.2 Transitive data dependences

Transitive data dependences are the only difference that we came across while comparing the sets of dependences extracted by a purely dynamic profiler and our approach. Consider two memory-access instructions S1 and S2 in a program. If S1 precedes S2 in execution and both either read from or write to the same memory location M, we say that S2 is data dependent on S1. Now consider an additional statement S3 that accesses M, too. We say that there is a transitive data dependence between S1 and S3 if S1 depends on S2 and S2 depends on S3. Transitive data dependences can be derived based on other data dependences that we identify. In Figure 4, the value of variable x is read in node 2. Nodes 1 and 3 store values in variable x. Our approach identifies a RAW dependence between nodes 1 and 2, and a WAR dependence between nodes 3 and 2. There is a transitive data dependence between nodes 3 and 1. The type of the dependence is WAW. We can identify the transitive data dependence and its type by following the chain of the identified dependences, starting from node 3 to node 2 and further to node 1. Note that transitive data dependences only provide additional information and are not important for parallelization, as long as the chain of dependences that create a transitive data dependence are extracted. Since our method identifies the dependences that constitute transitive dependences, we do not generate and report transitive dependences to keep the set of data dependences concise.

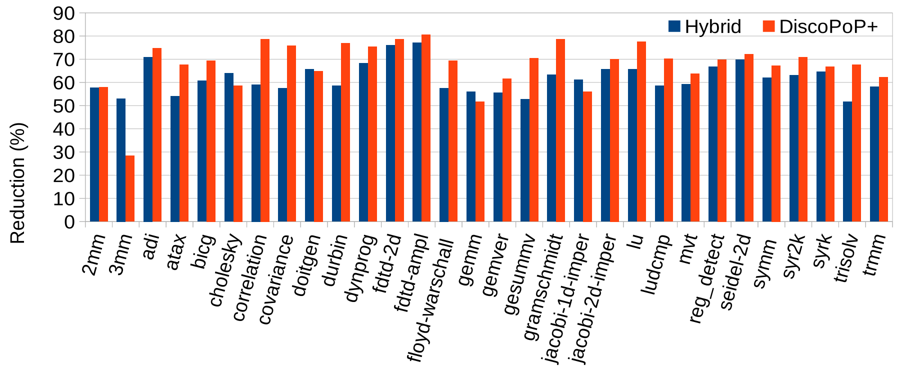
4 Evaluation

We performed a range of experiments to evaluate the effectiveness of our approach. We used the following benchmarks: NAS Parallel Benchmarks 3.3.1 [16] (NPB), a collection of programs derived from real-world computational fluid-dynamics applications, Polybench 3.2 [17], a set of benchmarks including polyhedral loops mainly, and the Barcelona OpenMP Task Suite (BOTS) 1.1.2 [15], a suite that all the benchmarks contain recursive functions. Since Polybench has been designed as a test suite for polyhedral compilers, it is well suited for comparison with DiscoPoP+ [11]. Also, the NBP benchmarks contain many polyhedral loops. In addition, we used BOTS to measure the usefulness of our method for recursive functions.

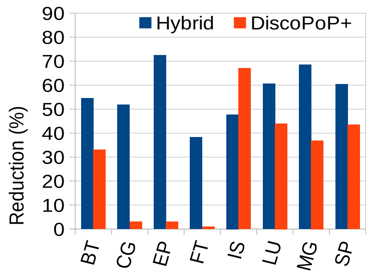
We compiled the benchmarks using clang 8.0.1, which is also used by the data-dependence profiler of DiscoPoP. We ran the benchmarks on an Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz with 64Gb of main memory, running Ubuntu 14.04 (64-bit edition). We profiled the benchmarks using the inputs packaged with 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 sets of data dependences extracted by the DiscoPoP profiler with and without our technique. Transitive data dependences were the only difference between the two sets. We identified all the dependences that created the transitive data dependences and thus, the set of dependences detected by our method can be used further to parallelize the programs.

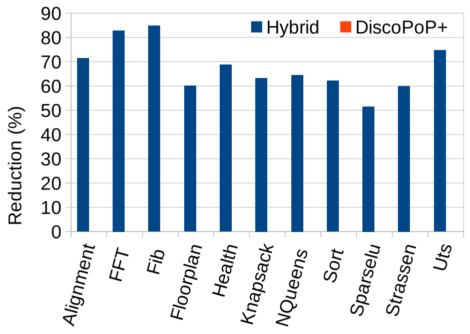
To measure the improvements in the profiling time, we executed the benchmarks with the vanilla version of the DiscoPoP profiler. We executed each benchmark five times in isolation, calculated the median of the execution times, and used it as our baseline. Then, we profiled the benchmarks using our method. Again, we ran each benchmark fives times in isolation and recorded its median execution time, which we then compared with the baseline. We repeated the process to obtain the median execution times for DiscoPoP+. We used the same input to execute the benchmarks with each approach. Table 1 shows the relative slowdown of each approach for the three benchmark suites. Figure 5 presents the relative reduction of the profiling overhead for each benchmark.

Whether we can reduce the profiling time of a benchmark depends on its memory access pattern. In theory, the more memory accesses that occur without using pointers and aliases, the more effective our method will be. If the variables in a program are mostly pointers or passed by reference to functions, we fail to reduce the profiling overhead significantly. Notably, our method lowered the profiling time in all test cases.



(a) Polybench





(b) NPB

(c) BOTS

Fig. 5. Profiling-time reduction relative to the standard DiscoPoP profiler.

For Polybench, our hybrid technique reduced the profiling overhead to a lesser degree than DiscoPoP+ because these benchmarks contain polyhedral loops. DiscoPoP+ eliminates these loops from profiling, whereas our approach skips only a subset of the memory instructions within those loops. This is why the median improvement of the profiling time is 70% with DiscoPoP+, but only 61% with our hybrid method.

BOTS does not contain any polyhedral loops, which is why DiscoPoP+ did not improve the profiling time at all. In contrast, the median improvement of

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

Benchmark	Standard DiscoPoP			DiscoPoP+			Hybrid approach		
suites	Min	Max	Median	Min	Max	Median	Min	Max	Median
BOTS	29	124	80	29	124	80	6	55	28
Polybench	70	200	121	17	70	37	24	85	43
NPB	25	116	88	22	113	53	7	71	36

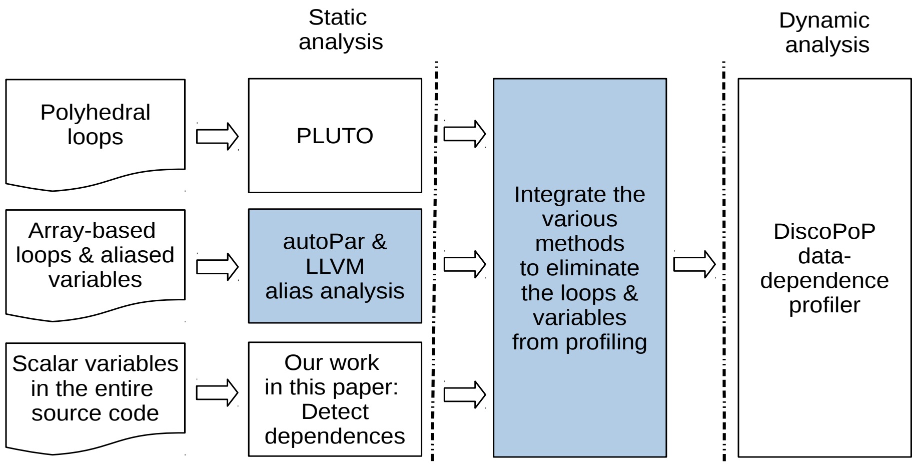

Static analysis
Dynamic analysis
Polyhedral loops
PLUTO
Array-based loops & aliased variables
autoPar & LLVM alias analysis
Scalar variables in the entire source code
Our work in this paper: Detect dependences
Integrate the various methods to eliminate the loops & variables from profiling
DiscoPoP data-dependence profiler

Fig. 6. The workflow of our future hybrid data-dependence analysis. Dark boxes show the contributions of our future work.

the profiling time by our method across all BOTS benchmarks was 64%. In Fib, we reduced the profiling time even by 84%.

In NPB, we found polyhedral loops in all benchmarks. Nevertheless, because these loops did not consume a major fraction of the execution time, excluding them did not make the profiling significantly better. DiscoPoP+ obtained only a median reduction of 35% for these benchmarks. Our approach, on the other hand, identified many variables in time-consuming loops. It skipped profiling the memory-access instructions related to those variables and improved the profiling time by a median percentage of 57% across all benchmarks in the suite.

Overall, compared to the vanilla version of DiscoPoP, we reduced the profiling time of all programs by at least 38%, with a median reduction of 61% across all the three benchmark suites.

5 Conclusion

Our hybrid approach to data-dependence analysis allows the profiler to skip the memory instructions of scalar variables whose dependences can be extracted statically. However, we still instrument memory operations of aliased variables to capture their data dependences at runtime, avoiding the loss of any data dependence that a purely dynamic method would extract. We implemented our approach as an extension of an advanced data-dependence profiler and decreased the profiling time by at least 38%, with a median reduction of 61% across 49 programs from three benchmark suites, making it far more practical than before. Having a faster profiler, DiscoPoP tool is able to identify parallelism opportunities in larger and longer-running programs. However, our method is generic enough to be implemented in any data-dependence profiler.

Our objective for the future work is to reduce the profiling overhead further and for a wider range of programs. Figure 6 shows the workflow of our future hybrid data-dependence analysis. First, we will aim to exploit LLVM alias analysis to statically detect data dependences for aliased scalar variables and eliminate their memory accesses from profiling. Then, we will investigate the inclusion of other promising tools such as autoPar, which statically identifies data dependences for array variables. Finally, we will combine them with DiscoPoP+ and our approach from this paper, creating a superior tool for hybrid data-dependence analysis.

6 Acknowledgement

This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

References

1. Bondhugula, U.: Pluto - an automatic parallelizer and locality optimizer for affine loop nests. http://pluto-compiler.sourceforge.net/ (2015) Accessed: 2019-06-13.
2. Liao, C., Quinlan, D.J., Willcock, J.J., Panas, T.: Semantic-aware automatic parallelization of modern applications using high-level abstractions. International Journal of Parallel Programming **38**(5) (Oct. 2010) 361–378
3. Ramos, P., Mendonca, G., Soares, D., Araujo, G., Pereira, F.M.Q.: Automatic annotation of tasks in structured code. In: Proc. of the International Conference on Parallel Architectures and Compilation Techniques (PACT), Limassol, Cyprus (May 2018) 20–33
4. Wilhelm, A., Cakaric, F., Gerndt, M., Schuele, T.: Tool-based interactive software parallelization: A case study. In: Proc. of the International Conference on Software Engineering (ICSE), Gothenburg, Sweden (June 2018) 115–123
5. Ketterlin, A., Clauss, P.: Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In: Proc. of the International Symposium on Microarchitecture (MICRO), Vancouver, B.C., Canada (Dec. 2012) 437–448
6. Kim, M., Kim, H., Luk, C.K.: SD3: A scalable approach to dynamic data-dependence profiling. In: Proc. of the International Symposium on Microarchitecture (MICRO), Atlanta, GA, USA (Dec. 2010) 535–546
7. Norouzi, M., Wolf, F., Jannesari, A.: Automatic construct selection and variable classification in OpenMP. In: Proc. of the International Conference on Supercomputing (ICS), Phoenix, AZ, USA (June 2019) 330–342

8. Li, Z., Atre, R., Huda, Z.U., Jannesari, A., Wolf, F.: Unveiling parallelization opportunities in sequential programs. Journal of Systems and Software **117**(C) (July 2016) 282–295
9. Li, Z., Jannesari, A., Wolf, F.: An efficient data-dependence profiler for sequential and parallel programs. In: Proc. of the International Parallel and Distributed Processing Symposium (IPDPS), Hyderabad, India (May 2015) 484–493
10. Li, Z., Beaumont, M., Jannesari, A., Wolf, F.: Fast data-dependence profiling by skipping repeatedly executed memory operations. In: Proc. of the International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP), Zhangjiajie, China (Nov. 2015) 583–596
11. Norouzi, M., Ilias, Q., Jannesari, A., Wolf, F.: Accelerating data-dependence profiling with static hints. In: Proc. of the European Conference on Parallel Processing (Euro-Par), Göttingen, Germany (Aug. 2019) 17–28
12. Benabderrahmane, M.W., Pouchet, L.N., Cohen, A., Bastoul, C.: The polyhedral model is more widely applicable than you think. In: Proc. of the Conference on Compiler Construction (CC), Paphos, Cyprus (Mar. 2010) 283–303
13. Liao, C., Quinlan, D.J., Willcock, J.J., Panas, T.: Extending automatic parallelization to optimize high-level abstractions for multicore. In: International Workshop on OpenMP (IWOMP), Dresden, Germany (June 2009) 28–41
14. Sampaio, D., Ketterlin, A., Pouchet, L., Rastello, F.: Hybrid data dependence analysis for loop transformations. In: Proc. of the International Conference on Parallel Architecture and Compilation Techniques (PACT), Los Alamitos, CA, USA (Sep. 2016) 439–440
15. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: Proc. of the International Conference on Parallel Processing (ICPP), Vienna, Austria (Sep. 2009) 124–131
16. 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. International Journal of Supercomputer Applications **5**(3) (Sep. 1991) 63–73
17. Pouchet, L.N.: Polyhedral suite. http://www.cs.ucla.edu/ pouchet/software/polybench/ (2011) Accessed: 31.01.2020.