Fast data-dependence profiling through prior static analysis

Mohammad Norouzi\textsuperscript{a,}\textsuperscript{∗}, Nicolas Morew\textsuperscript{a}, Qamar Ilias\textsuperscript{a}, Lukas Rothenberger\textsuperscript{a}, Ali Jannesari\textsuperscript{b}, Felix Wolf\textsuperscript{a}

\textsuperscript{a} Technical University of Darmstadt, Department of Computer Science, Germany
\textsuperscript{b} Iowa State University, Department of Computer Science, Ames, IA, United States of America

\textbf{A B S T R A C T}

Data-dependence profiling is a program-analysis technique for detecting parallelism opportunities in sequential programs. It captures data dependences that actually occur during program execution, filtering parallelism-preventing dependences that purely static methods assume only because they lack critical runtime information, such as the values of pointers and array indices. Profiling, however, suffers from high runtime overhead. In our earlier work, we accelerated data-dependence profiling by excluding polyhedral loops that can be handled statically using certain compilers and eliminating scalar variables that create statically-identifiable data dependences. In this paper, we combine the two methods and integrate them into DiscoPoP, a data-dependence profiler and parallelism discovery tool. Additionally, we detect reduction patterns statically and unify the three static analyses with the DiscoPoP framework to significantly diminish the profiling overhead and for a wider range of programs. We have evaluated our unified approaches with 49 benchmarks from three benchmark suites and two computer simulation applications. The evaluation results show that our approach reduces fewer false positive and negative data dependences than the original data-dependence profiler and reduces the profiling time by at least 43%, with a median reduction of 76% across all programs. Also, we identify 40% of reduction cases statically and eliminate the associated profiling overhead for these cases.

\textbf{1. Introduction}

Parallelism is a technique that leverages the performance of programs beyond software optimizations. Identifying parallelism opportunities in a program requires a deep knowledge of the program and its algorithm. Acquiring this knowledge is hard and, in many cases, impossible because the developers of the program are unavailable. In such situations, parallelism discovery tools can help programmers transform a sequential program into its parallel version. The transformations, however, must preserve the semantics of the sequential program. An important pillar that holds the semantics of the program is the set of data dependences. Therefore, all parallelism discovery tools must identify the set of data dependences before they can apply any transformations.

There are two major approaches for data-dependence analysis: Static and dynamic. Static methods are fast, but they assume that a data dependence exists when the detection process requires runtime information, making the parallelism discovery more conservative than it needs to be. In general, methods based on static analysis do not discover parallelism opportunities beyond trivial cases. Dynamic methods profile every memory access in the program to find data dependences that actually occur during the program execution. There are some concerns regarding parallelism discovery tools that rely on dynamic analysis. One of them is input sensitivity, which is the probability of obtaining inconsistent sets of data dependences when running the program with different inputs. Related work\textsuperscript{[1,2]} suggests that data dependences in code sections that are subject to parallelization do not change substantially with different inputs. Loops and recursive functions are two major programming constructs in which most of the execution time of a program is spent. These constructs often iterate over the elements of sequence containers (i.e., arrays, vectors, matrices, etc.), and the inputs to many scientific programs often determine the dimension/size of the containers and the values of the elements. Hence, the inputs change neither the instructions that access the memory nor their execution order. Therefore, a data-dependence profiler records the same set of dependences from the hotspots when it runs the program with different inputs. Nonetheless, tools that use dynamic analysis provide weaker correctness guarantees, although their suggestions more than often reproduce manual parallelization strategies\textsuperscript{[1,3,4]}.

In this paper, we propose a unified hybrid approach that coordinates the two methods into a single framework. Our approach employs profiling only when the detection of data dependences requires runtime.
information. First, we identify memory-access instructions that are inside polyhedral loops or the instructions that write to or read from memory locations that belong to scalar variables. Then, we omit both sets of instructions from the instrumentation, relieving the profiler from their associated overhead. With our unified approach, we are able to reduce the profiling overhead more significantly and for a wider range of programs. Now, we skip profiling polyhedral loops completely in addition to instructions that can belong to all types of loops (e.g., canonical or non-canonical) or functions (e.g., recursive or non-recursive) in a program. Additionally, our hybrid approach soothes the input sensitivity problem: the static phase of our approach is able to detect data dependences in code sections that might not be visited at runtime with the given inputs.

Moreover, our approach is able to detect reduction patterns in many cases statically. The patterns exist in loops with a specific type of inter-iteration dependence; each iteration of the loop reads the value of the reduction variable from the previous iteration, performs a mathematical operation (e.g., addition) on the variable, and passes the updated value to the next iteration. This happens, for example, when a loop adds all the elements of an array. Many parallel programming paradigms, including OpenMP, provide particular constructs to parallelize reduction loops.

In this paper, we focus on our approach for reducing the overhead of data-dependence profiling and refer the readers interested in discovering parallelism based on the extracted data dependences to related work [1,3–5]. In summary, we make the following specific contributions:

- Presenting a unified framework for hybrid data dependence analysis that combines the advantages of static and dynamic techniques. Running the dynamic analysis only for code sections where the data-dependence detection requires runtime information, the framework reduces the profiling overhead significantly for a wide range of applications.
- An implementation as an extension of the data-dependence profiler of DiscoPoP [5], which is a parallelism discovery tool. However, our approach is generic enough to be implemented in any data-dependence profiler.
- An evaluation with 49 programs from three benchmark suites and two computer simulation programs, reducing the profiling time by at least 43%, with a median reduction of 76%.
- Presenting a method for detecting reduction patterns statically using the DiscoPoP framework. Our technique found more than half of the reduction opportunities in the benchmarks and the simulation programs statically.

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 [1,3–9]. Most approaches focus on either static or dynamic analysis techniques, with only a few attempting to combine them.

autoPar [8] is a static analysis tool implemented on top of the ROSE [10] source-to-source compile infrastructure, which can parallelize array-based loops [11]. 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. Similar to autoPar, autoPar-clava [12] is a static automatic parallelization approach for the Clava source-to-source compiler [13] with a focus on identifying parallel loops and reduction operations. Similar to the DiscoPoP framework, it is able to propose OpenMP pragmas for parallelization and uses similar schemes for identifying reduction operations and parallelizable loops from a set of data dependences. In contrast to DiscoPoP's profiling-based approach, however, only dependences within loops are considered, which limits the scope to parallelizing individual loops compared to the theoretically possible (e.g., task-based) parallel execution of different loops side by side, which could be identified on the basis of the DiscoPoP framework.

PLUTO [7] is an auto-parallelizing compiler that detects data dependences statically in 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 [14]. 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. The overhead of PLUTO for identifying data dependences statically depends on the code section containing the polyhedral loop. In general, this overhead is in the order of minutes at most, and, thus, is negligible compared to the overhead of profiling. TaskMiner [9] is a static analysis tool that 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 such as the values of pointers and array indices. As a consequence, parallelism discovery tools that rely on these techniques may miss certain parallelism opportunities because they assume data dependences that may never occur.

Avoiding the limitations of purely static analysis, many tools [1,3–5] capture data dependences during program execution. They profile memory accesses, which imposes huge runtime overhead. There are many optimizations available to lower the profiling overhead. For example, Parwiz [3], a parallelism discovery tool, coalesces contiguous memory accesses. This lowers the profiling overhead, but only for a subset of the memory accesses. Kremlin [15], another parallelization recommender system, profiles data dependences only within specific code regions. To save memory overhead, SD3 [2,4], a dependence profiler, compresses memory accesses with stride patterns. Moreover, it reduces the runtime overhead by parallelizing the profiler itself.

Dyninst [16], like DiscoPoP's scalar variable elimination, detects parts of the code that can be eliminated from profiling. Unlike Dyninst, which works on the binary version of the program, DiscoPoP operates on the source-code level. The profiling part of DiscoPoP considers the line number of memory-access instructions to trace data dependences at runtime. As a result, the static analysis part of DiscoPoP must also work on the source-code level, determine code sections that can be eliminated from profiling, and feed the information to the dynamic analysis component.

Another method to reduce runtime overhead is sampling [17], 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. DiscoPoP [5] is a parallelism discovery tool that contains a data-dependence profiler [18]. 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 by factors typically in the order of magnitude of 100x (e.g., 86x on average for the approach presented in [18], and 70x for profiling only the 20 hottest loops using SD3 with eight threads [4]). In [18], DiscoPoP has been compared to a number of state-of-the-art data-dependence profilers and parallelism discovery tools over a range of benchmarks. In the study, DiscoPoP shows comparably better performance. Additionally, the overhead of data-dependence profiling has been addressed in a recent study [19]. The proposed profiler is able to achieve significantly better runtime overheads of merely 1.1x on average for Polybench compared to 24.4x resulting from DiscoPoP. However, due to its structure, Polybench acts as a perfect example for the approach presented in [19], while other benchmarks do not depict such a clear advantage (134.4x for NPB, 55.8x for SPEC 2006). In addition, and in contrast to DiscoPoP’s profiler, the profiling approach proposed in [19] has the potential to fall back to conservative assumptions in the case of indirect memory accesses and thus report unnecessarily restrictive data dependencies.

Recently, we introduced two techniques [20,21] for the data-dependence analysis. The techniques use the data-dependence profiler of DiscoPoP as the basis of their implementation. They both use static methods to extract data dependences and accelerate the profiling by excluding memory-access instructions that create the data dependences. The first method [20] runs PLUTO to statically identify data dependences in polyhedral loops. Then, it excludes the loops from instrumentation, profiling only data dependences outside the loops. In the end, it merges static and dynamic dependences. It reduces the profiling overhead significantly, but only for programs containing polyhedral loops. However, neither does every program contain polyhedral loops, nor are statically identifiable dependences restricted to such loops. The second technique [21], which is orthogonal to the first, 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 variables, not including passing-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. This method, however, does not skip profiling polyhedral loops. Our unified approach, which we present in this paper, combines both methods into a unified framework to reduce the profiling overhead further and for a wider range of programs.

Apollo [22] is a tool that parallelizes programs speculatively. It relies on the polyhedral model to find data dependences in polyhedral loops and suggests parallelizing them at runtime. Unlike Apollo, which is confined to loops, our approach detects data dependences in the whole program, including loops and functions. This allows more parallelization opportunities to be exploited in a broader spectrum of programs. Moreover, Apollo excludes only polyhedral loops from profiling. In addition to these loops, we eliminate profiling scalar variables that create statically identifiable data dependences, and, thus, accelerate the profiler further. Another hybrid-analysis framework was proposed by Sampaio et al. [23]. Their goal is to provide 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. Moreover, Rus et al. [24] presented a framework for hybrid data-dependence analysis. 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 the works [23,24], our contribution happens at a lower level, where we just collect data dependences, with the goal of increasing the profiling speed.

3. Hybrid data-dependence analysis

In this section, we first introduce the representation format that we use to report data dependences. Then, we explain our hybrid approach for the detection of data dependences. Fig. 1 shows the basic workflow of our approach. Dark boxes highlight our contribution in relation to our previous work and the isolated static and dynamic dependence analyses.

Our approach first eliminates the memory-access instruction that create statically-identifiable data dependences from instrumentation. Section 3.1 presents the details of our method for the detection of such instructions. In a complementary approach, our hybrid excludes polyhedral loops from instrumentation. We use PLUTO to draw out the data dependences that exist within the boundaries of polyhedral loops. Section 3.2 explains the details for identifying the loops and skipping them from profiling. We also discuss the relation between the set of data dependences extracted by our hybrid and the purely dynamic approach in Section 3.4. Finally, we present our approach for the detection of reduction patterns in Section 3.5.

3.1. Scalar variable elimination

We eliminate a memory-access instruction from profiling under certain conditions. For this to be applicable, it needs to be guaranteed that the instruction creates only statically-identifiable data dependences and, thus, we can safely omit it without missing any 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. Fig. 2 serves as an illustrating example.

**Algorithm 1**: Finding memory addresses that are statically predictable.

```plaintext
for each function F ∈ program P do
  staticAddrs = {}
  for each instruction I ∈ F do
    if I.isAlloc() then
      addr = I.getMemAddr()
      staticAddrs.insert(addr)
    for each instruction I ∈ F do
      if I.isCall() then
        params = I.getParams()
        for each param p ∈ params do
          if p.isPassedByReference() then
            addr = p.getMemAddr()
            staticAddrs.remove(addr)
      else if I.isStore() then
        var = I.storedVar()
        if var.isMemAddr() then
          pointerVar = I.getPointer()
          staticAddrs.remove(pointerVar)
```

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 Fig. 2, the set initially includes the address of variables.

---

M. Norouzi et al.
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 \( \text{bar} \) 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 instruction at line 3. All memory instructions of variable \( y \) should be profiled, and therefore, we discard them from further analysis.

In Fig. 2, most variables are aliased via pointers or references. In practice, we rarely find programs that contain only aliased variables. Fig. 3 shows function \( \text{fib} \) from BOTS [25]. There, we can skip profiling memory instructions of all variables, i.e., \( i, j, n \), and an implicit variable \( \text{retval} \), which saves the return value because we identify data dependences between their accesses statically.

Figs. 4 to 6 demonstrate the analyses that we perform to extract data dependences statically, using function \( \text{fib} \) in Fig. 3 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 \( \text{fib} \) is shown in Fig. 4. 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. Hence, we briefly refer to MCFG as the memory-access graph or simply as graph if the context allows it. Fig. 5 shows the memory-access graph of function \( \text{fib} \).

We traverse the graph to extract data dependences statically. Algorithm 2 shows how. Fig. 6 illustrates the dependences that we extract from the memory-access graph of \( \text{fib} \).

---

**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 \( I \)):**

\[
\begin{align*}
\text{if } & I.\text{isEntry}() \text{ or } I.\text{isVisited}() \text{ then } \text{ return; }
\end{align*}
\]

for each node \( J \) directly preceding \( I \) do

checkDepsBetween(I,J);

findDepsFor(J);

**Function checkDepsBetween(node \( I \), node \( J \)):**

\[
\begin{align*}
\text{if } & J.\text{isEntry}() \text{ then } \text{ return; }
\end{align*}
\]

if \( J.\text{getMemAddr}() == I.\text{getMemAddr}() \) then

\[
\begin{align*}
\text{if } & J.\text{isStore}() \text{ then } \text{ addDataDeps(I, J); return; }
\end{align*}
\]

else

checkForRARDep();

for each node \( K \) directly preceding \( J \) do

if \( K.\text{isVisited}() \) then

\[
\begin{align*}
K.\text{isVisited} = \text{true}
\end{align*}
\]

checkDepsBetween(I,K);

---

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

**Fig. 2.** A program containing only aliased variables.

**Fig. 3.** Function \( \text{fib} \) from BOTS. The memory addresses of all variables are statically predictable.
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 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 Fig. 5, the value of variable i is read in node 8. The value was previously stored in node 5. Fig. 6 shows the data dependence that our approach adds between the nodes. The type of the dependence is RAW because the value of variable i is read after it is written.

Read-after-read (RAR) dependences are helpful in identifying data reuse opportunities. Although we identify them, we do not report them because 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 completeness of data dependences captured by purely dynamic analysis. In Algorithm 2, function checkForRARDep determines whether a memory address is read in a function. In function fib in Fig. 3, 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 is $O(f \cdot n^2)$, 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 compared 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.
Algorithm 3: Exclusion of memory-access instructions from instrumentation

```plaintext
for each function \( f \in \text{program} \) do
  \( \text{SCoPSet} = \text{PLUTO.getSCoPs}(f) \)
  for each \( \text{SCoP} s \in \text{SCoPSet} \) do
    \( \text{varSet} = \text{getVariables}(s) \)
    for each \( \text{variable} \ var \in \text{varSet} \) do
      instrument(firstLoadInst(var, s))
      instrument(lastLoadInst(var, s))
      instrument(firstStoreInst(var, s))
      instrument(lastStoreInst(var, s))

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 to be used by parallelism discovery tools.

3.1.1. 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 \( S_1 \) and \( S_2 \) in a program. If \( S_1 \) precedes \( S_2 \) in execution and both either read from or write to the same memory location \( M \), we say that \( S_2 \) is data dependent on \( S_1 \). Now consider an additional statement \( S_3 \) that accesses \( M \) too. We say that there is a transitive data dependence between \( S_1 \) and \( S_3 \) if \( S_1 \) depends on \( S_2 \) and \( S_2 \) depends on \( S_3 \). Transitive data dependences can be derived based on other data dependences that we identify. In Fig. 7, 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.

3.2. Polyhedral loop exclusion

We exclude specific memory-access instructions from instrumentation that appear inside source code locations from which PLUTO can extract data dependences statically. Algorithm 3 shows the details and can be best understood when following the examples in Fig. 8.

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 \( a \).

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 to avoid missing 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. Figs. 9(a) and 9(b) show situations that create false negatives. If we exclude the first read in Fig. 9(a), 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-write (WAW) between the first write and the write before the SCoP (Fig. 9(b)), 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. 9(a)), 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. 9(b)) 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.

Figs. 9(c) and 9(d) show situations that create false positives. Three types of false positives are reported if the boundary instructions are not instrumented. Fig. 9(c) shows a false positive RAW between the last write preceding the SCoP and the first read succeeding it. Fig. 9(d) 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.

Algorithm 4: Transformation of data dependences identified by PLUTO into the unified representation.

1. for each SCoP scop ∈ SCoPSet do
2.   fileList = findFileID(scop)
3.   depSet = PLUTO.getDepSet(scop)
4.   for each dependence dep ∈ depSet do
5.     varName = getVarName(dep)
6.     sourceLine = findSourceLine(dep)
7.     sinkLine = findSinkLine(dep)
8.     depType = getDataType(dep)
9.     print(fileID : sinkLine depType)
10.   print(fileID : sourceLine varName)

3.3. 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. 10. 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:variablesName>. We assign a unique fileId to each file in a program. Existing profilers, including Parviz, DiscoPoP, SD3, and Intel Pin [26], 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 4 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. In 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.

### 3.4. 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. 11. To better understand this relation, let us consider the listings in the figure. In Fig. 11(b), the loop in the if part meets the constraints of the polyhedral model. PLUTO finds data dependences in the loop. The scalar variable elimination approach detects data dependences in the else part, and thus, our unified hybrid approach excludes the whole conditional block from profiling. Profilers might execute either the if or the else branch, depending on the condition \( k < \text{average} \), 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 therefore a superset of the set of purely dynamic data dependences (i.e., \( D \subseteq H \)). Fig. 11(c) 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. 11(d), 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., \( H = D \)).

### 3.5. Reduction detection

Reduction patterns are suitable for a loop with a specific type of inter-iteration dependence, that is, the loop uses an associative binary operator to reduce all elements of a container to a single scalar value. This happens, for example, when a loop adds all the elements of an array. We developed an LLVM pass [1], as part of the DiscoPoP tool, that instruments all LLVM-IR instructions that create inter-iteration dependences in a loop. It records the source-line numbers for each read and write operation on every variable. The variables can be scalar or arrays with any number of dimensions. If a memory address is written only once after it is read, we mark the loop as a possible candidate for a reduction. However, many cases of the reduction pattern can be detected statically. We have modified the LLVM pass to skip profiling loops in which we can detect reduction opportunities statically. Now, the LLVM pass marks a variable as a reduction candidate if the read and write operations on the variable happen at the same source code line. The method used to analyze usage patterns and thereby detect reduction operations is fundamentally similar to the approach described by Arabnejad et al. in [12]. However, our approach does not identify all cases of reduction statically. The cases that are beyond the scope of our analysis are the reductions related to multi-dimensional arrays (i.e., more than two dimensions), arrays with complex subscripts, or variables with non-standard (i.e., user-defined) data types. This is acceptable since we can rely on the profiling results to identify the missing reduction opportunities. Nevertheless, adding, for example, a more sophisticated static analysis of array subscripts and especially their relation to the loop indices, as shown in [12], could be a reasonable approach to increasing the amount of statically identifiable reduction loops, thus decreasing the profiling overhead.

### 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 [27] (NPB), a collection of programs derived from real-world computational fluid-dynamics applications, Polybench 3.2 [28], a set of benchmarks including polyhedral loops mainly, and the Barcelona OpenMP Task Suite (BOTS) 1.1.2 [25], a suite of benchmarks containing recursive functions. In addition to the benchmarks, we have analyzed our approach with LULESH, a C++ application that is used in the field of dynamic fluid simulation, and EOS-MBPT code, a C++ astro-physics simulation code. Since Polybench has been designed as a test suite for polyhedral compilers, it is well suited for comparison with our earlier work [20]. Also, the NPB 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 @ 2.60 GHz with 64Gb of main memory, running Ubuntu 14.04 (64-bit edition). We profiled the benchmarks using the inputs packaged with the programs.

EOS-MBPT code has many software dependencies. It uses the Cuba library [29] and relies on the GNU scientific library (GSL) [30] and the OpenBLAS [31] package. Because of the software dependencies, we ran the program on Lichtenberg Cluster of Technical University of Darmstadt, Germany. The cluster provides the software dependencies and 2.5 GHz Intel Xeon E5-2670v3 processors. Moreover, we used clang 8.0.1 to compile the code and profile it with the data-dependence profiler of DiscoPoP and our hybrid approach.

Our evaluation criteria are the completeness of the data dependences in relation to purely dynamic profiling and the profiling time. In Section 4.1, we discuss the accuracy of the identified data dependences and explain the performance of our approach in Section 4.2.

#### 4.1. Accuracy of the extracted data dependences

##### 4.1.1. Input sensitivity

We compared the sets of data dependences extracted by the DiscoPoP profiler with and without our technique. Because the entire source code of the benchmarks was visited during the execution with the given inputs, we observed that, excluding the transitive data dependences, there is no difference in the reported data dependences. 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. Moreover, following the arguments of Section 3.4, we believe that higher code-coverage [32] potential makes our approach generally less input-sensitive than purely dynamic methods, a claim we want to substantiate in a follow-up study.
million slots in the signature, the highest rate of false positive data to recorded memory accesses and identify data dependences. With one profiler assigns a slot in the hash table to every element of the matrices large matrices, which increase the chance of hash conflicts. DiscoPoP negative rates in the Polybench programs. These programs contain DiscoPoP. We observe that our approach has smaller false positive negative data dependences reported by our approach and the original shadow signatures. Table 1 shows the rates of false positive and false can be used further to parallelize the programs. We compared the sets of dependences extracted by the DiscoPoP profiler with and without our unified hybrid approach. Whenever we used perfect signatures, 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 sets. We identified all the dependences that created the transitive data dependences extracted by the DiscoPoP profiler with and without the variable. However, DiscoPoP provides users with two options: perfect and shadow signatures. When the perfect signature option is selected, every memory location is recorded in a unique slot in the hash, and thus, there will be no conflicts. The shadow signature, on the other hand, provides users with the option to set the number of slots in the hash table. This option enables users to maintain the memory consumption, but it also introduces the chance of false positive and false negative data dependences in case of a hash conflict, i.e., two memory addresses are assigned to one slot in the table. Li et al. [5] presented the rates of the false positive and false negative data dependences in relation to the number of slots in the signature data structure. To compute the rate, they run programs with the perfect signature and record the data dependences. Then, they run programs under the same conditions with the shadow signature. They used different signature sizes and concluded that the false positive and false negative rates reduce significantly when the number of slots in the signature increases.

We used the same approach to evaluate the accuracy of the data dependences extracted by our approach. We compared the sets of data dependences extracted by the DiscoPoP profiler with and without our unified hybrid approach. Whenever we used perfect signatures, 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.

We also evaluated the effectiveness of our approach with the shadow signatures. Table 1 shows the rates of false positive and false negative data dependences reported by our approach and the original DiscoPoP. We observe that our approach has smaller false positive and false negative rates. We observed the highest false positive and negative rates in the Polybench programs. These programs contain large matrices, which increase the chance of hash conflicts. DiscoPoP profiler assigns a slot in the hash table to every element of the matrices to recorded memory accesses and identify data dependences. With one million slots in the signature, the highest rate of false positive data dependences reported with the original profiler of DiscoPoP occurs for Polybench and is 16%. Under the same settings, our approach reported only 4.3% false positives. Excluding polyhedral loops from profiling, our approach reduced the number of memory locations that the profiler should allocate in the signature. In general, our approach excludes certain variables from the instrumentation, letting the profiler allocate fewer slots in the signature, thus creating fewer hash conflicts.

Moreover, the profiler monitors all accesses to a memory location at runtime. Removing a variable (and all its accesses from instrumentation) relieves the profiler from dedicating a slot in the signature for the variable. So, the reduction in memory overhead is proportional to the number of variables that are eliminated from profiling. If the variable that can be eliminated from profiling is defined inside a loop or a recursive function, the memory overhead reduction can be significant.

4.2. Performance

To measure the improvements in the profiling time, we executed the programs 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 profied the benchmarks using our methods: polyhedral exclusion, scalar variable elimination, and the unified hybrid approach. We ran each benchmark five 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 each of the methods. We used the same input and thread count of four to execute the benchmarks with each approach. Table 2 shows the relative slowdown of each approach for the three benchmark suites and the simulation programs. Fig. 12 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 and computational patterns. 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, it is unlikely that our method reduces the profiling overhead substantially. Additionally, we observed that our approach reduces the profiling overhead if a program contains polyhedral loops that take up a substantial amount
of the execution time of the program. If a program does not contain such loops, we fail to reduce the profiling overhead significantly.

BOTS, LULESH, and the EOS-MBPT do not contain polyhedral loops, and thus, the polyhedral exclusion method does not help with the profiling. Skipping memory-access instructions of the scalar variables from profiling, however, we reduce the overhead significantly. The median improvement of the profiling time by our method across all BOTS benchmarks was 64%. For four benchmarks in NPB, namely EP, IS, CG, and MG, we observed only small improvements when eliminating polyhedral loops from profiling because the benchmarks do not contain long-running polyhedral loops, and thus, we could not exclude many instructions from instrumentation. This method only reduced the profiling time by a median of 35% across all the NPB benchmarks. In all benchmarks in the NPB suite, however, we find many instructions that create statically-detectable data dependences. Removing them from instrumentation, we improved the profiling time across all the NPB programs by a median of 57%. Our hybrid approach, however, reduced the profiling overhead further by a median of 66% over the benchmarks in the suite.

For Polybench, the scalar variable elimination technique reduced the profiling overhead to a lesser degree than the polyhedral loop exclusion method because these benchmarks contain polyhedral loops. Excluding the loops from profiling, we reduced the overhead by a median of 70% across the benchmarks in this suite while eliminating memory-access instructions of scalar variables improved the profiling time only by a median of 61%. Combining the strengths of both methods in a single framework, however, our unified approach decreased the overhead by a median of 81% across all the benchmarks in the suite.

Overall, compared to the vanilla version of DiscoPoP, our unified hybrid approach reduced the profiling time of all programs by at least 43%, with a median reduction of 76% across all the benchmark suites and the simulation programs.

Moreover, Table 3 shows the number of reduction operations that DiscoPoP and our hybrid approach identify. DiscoPoP always profiles loops to detect whether or not they contain the reduction pattern. Our approach, on the other hand, first attempts to find reduction loops statically and eliminates them from profiling. For loops where static analysis is not enough to identify reduction, we employ profiling, like DiscoPoP. According to Table 3, DiscoPoP and our approach detect the same number of reduction cases. However, we require to profile only 156 loops and find 277 cases statically, thus accelerating the reduction detection by skipping almost 60% of the cases from profiling.

Table 1
False positive (FP) and false negative (FN) rates of the detected data dependences by the original DiscoPoP vs. our unified hybrid approach for the three benchmark suites.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Hybrid</th>
<th>DiscoPoP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slots</td>
<td>1.0E+6</td>
<td>1.0E+7</td>
</tr>
<tr>
<td>Rates</td>
<td>FP</td>
<td>FN</td>
</tr>
<tr>
<td>BOTS</td>
<td>0.9</td>
<td>0.1</td>
</tr>
<tr>
<td>Polybench</td>
<td>4.3</td>
<td>0.0</td>
</tr>
<tr>
<td>NPB</td>
<td>0.5</td>
<td>0.0</td>
</tr>
<tr>
<td>Median</td>
<td>0.9</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Table 2
Relative slowdown caused by standard DiscoPoP vs. polyhedral loop exclusion vs. scalar variable elimination vs. our unified hybrid approach. The presented relative slowdowns for the individual benchmark suites were determined by calculating the median across the observed slowdowns of all included benchmarks. The reported values are rounded for convenience. The median is calculated across all benchmarks from all five benchmark suites.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DiscoPoP</th>
<th>Polyhedral loop exclusion</th>
<th>Scalar variable elimination</th>
<th>Unified hybrid approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOTS</td>
<td>80</td>
<td>0</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Polybench</td>
<td>121</td>
<td>34</td>
<td>43</td>
<td>20</td>
</tr>
<tr>
<td>NPB</td>
<td>88</td>
<td>53</td>
<td>36</td>
<td>30</td>
</tr>
<tr>
<td>LULESH</td>
<td>50</td>
<td>0</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>EOS-MBPT</td>
<td>290</td>
<td>0</td>
<td>95</td>
<td>95</td>
</tr>
<tr>
<td>Median</td>
<td>110</td>
<td>36</td>
<td>38</td>
<td>22</td>
</tr>
</tbody>
</table>

Table 3
The number of reduction patterns identified by DiscoPoP vs. our hybrid approach.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DiscoPoP</th>
<th>Unified hybrid approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOTS</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>Polybench</td>
<td>94</td>
<td>94</td>
</tr>
<tr>
<td>NPB</td>
<td>217</td>
<td>146</td>
</tr>
<tr>
<td>LULESH</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>EOS-MBPT</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>Total</td>
<td>383</td>
<td>227</td>
</tr>
</tbody>
</table>

Our approach finds almost all the reduction cases in Polybench suite and the EOS-MBPT code. Unlike our approach, DiscoPoP identifies reduction opportunities for variables that have a user-defined data type in the EOS-MBPT code. Also, our hybrid method could not identify reductions related to arrays with more than two dimensions in Polybench and most NPB programs. Moreover, we could not identify the reduction pattern in arrays whose indices are determined at runtime. However, we identified all the reductions in BOTS and LULESH. Notably, our method found more than half of the reduction opportunities in all test cases statically.

5. Conclusion

In this paper, we have presented a hybrid approach to data dependence analysis. Our approach identifies scalar variables that create statically-identifiable data dependences and eliminates from profiling the memory-access instructions that generate the dependences. Additionally, it uses PLUTO to extract data dependences within polyhedral loops and skips profiling them. We extended the data-dependence profiler of DiscoPoP with our approach and reduced the profiling time by at least 43%, with a median reduction of 76%, resulting in a median slowdown of 21.5x across 49 programs from three benchmark suites and two computer simulations. The evaluation results show that we are able to reduce the profiling overhead significantly and for a wider range of applications. Our method is generic enough to be implemented in any data-dependence profiler. Using our hybrid approach, parallelism discovery tools such as DiscoPoP can detect parallelization opportunities in a realistic time.

We still believe that we can exploit existing compiler methods to improve the static phase of our approach and, thus, reduce the profiling...
overhead even further. Two major options that we would like to pursue are the autoPar and LLVM alias analysis. We aim to use autoPar to extract data dependences in non-polyhedral loops and exploit LLVM analysis to detect data dependences for aliased scalar variables.

Declaration of competing interest

No conflicts.

Acknowledgments

This work was funded by the Federal Ministry of Education and Research (BMBF), funding no. NHR2021HE, and the state of Hesse (HMKW), funding no. Kapitel 1502, Förderprodukt 19 NHR-CE5 as part of the NHR Program, as well as by the Hessian LOEWE initiative within the Software-Factory 4.0 project. The authors gratefully acknowledge having conducted a part of this study on the Lichtenberg high-performance computer of TU Darmstadt.

References

M. Norouzi et al.

Parallel Computing 119 (2024) 103063


Mohammad Norouzi is a research associate at the Laboratory for Parallel Programming at TU Darmstadt in Germany. He received his Ph.D. degree in computer science from TU Darmstadt in 2021. His research interests revolve around parallelization and program analysis.

Nicolas Morew is a software developer at CGI company in Darmstadt. He graduated from TU Darmstadt with a Master’s degree in 2020.

Qamar Ilias is a software engineer at COMPRION, USA. He graduated from TU Darmstadt with a Master’s degree in 2018.

Lukas Rothenberger is a research associate at the Laboratory for Parallel Programming at TU Darmstadt in Germany. His research interests revolve around parallelization and program analysis.

Ali Jannesari is an assistant professor and the director of the Laboratory for Software Analytics and Pervasive Parallelism (SwAPP) at the Department of Computer Science of Iowa State University. His research focuses on HPC, parallel computing and data science. Before joining the faculty at ISU, he was a Senior Research Fellow at the University of California, Berkeley. He was a junior research group leader at the Technical University of Darmstadt and RWTH Aachen University. He holds a Habilitation Degree from the Technical University of Darmstadt and received his Ph.D. from the Karlsruhe Institute of Technology.

Felix Wolf is full professor at the Department of Computer Science of TU Darmstadt in Germany. He specializes in software and tools for parallel computing. After receiving his Ph.D. degree from RWTH Aachen University in 2003, he worked for more than two years as a postdoc at the Innovative Computing Laboratory of the University of Tennessee. In 2005, he was appointed research group leader at the Jülich Supercomputing Center. From 2009 until recently, he was head of Parallel Programming at the German Research School for Simulation Sciences. Prof. Wolf has published over a hundred refereed articles on parallel computing, several of which have received awards.

Nicolas Morew is a software developer at CGI company in Darmstadt. He graduated from TU Darmstadt with a Master’s degree in 2020.

Qamar Ilias is a software engineer at COMPRION, USA. He graduated from TU Darmstadt with a Master’s degree in 2018.

Lukas Rothenberger is a research associate at the Laboratory for Parallel Programming at TU Darmstadt in Germany. His research interests revolve around parallelization and program analysis.