The First Workshop on Software Engineering for Parallel Systems (SEPS 2014)
Portland, Oregon, United States
October 21, 2014

Discovering Parallelization Opportunities in Sequential Programs — A Closer-to-Complete Solution

Zhen Li, Ali Jannesari, and Felix Wolf

German Research School for Simulation Sciences, Aachen, Germany
RWTH Aachen University, Aachen, Germany

Abstract

The stagnation of single-core performance leaves application developers with software parallelism as the only option to further benefit from Moore's Law. However, in view of the complexity of writing parallel programs, the parallelization of myriads of sequential legacy programs presents a serious economic challenge. A key task in this process is the identification of suitable parallelization targets in the source code. Reversing the idea underlying data-race detectors, we show how dependency profiling can be used to automatically identify potential parallelism in sequential programs of realistic size. In comparison to earlier approaches, our work combines a unique set of features that make it superior in terms of functionality: It not only (i) detects available parallelism with high accuracy but also (ii) identifies the parts of the code that can run in parallel—even if they are spread widely across the code, (iii) ranks paralleization opportunities according to the speedup expected for the entire program, while (iv) maintaining competitive overhead both in terms of time and memory.

Keywords: Program Analysis, Profiling, Data Dependency, Parallelization

1. Introduction

Although the component density of microprocessors is still rising according to Moores Law, single-core performance is stagnating for more than ten years now. As a consequence, extra transistors are invested into the replication of cores, resulting in the multi- and many-core architectures popular today. The only way for developers to take advantage of this trend if they want to speed up an individual application is to match the replicated hardware with thread-level parallelism. This, however, is often challenging especially if the sequential version was written by someone else. Unfortunately, in many organizations the latter is more the rule than the exception [1]. To find an entry point for the parallelization of an organization's application portfolio and lower the barrier to sustainable performance improvement, tools are needed that identify the most promising parallelization targets in the source code. These would not only reduce the required manual effort but also provide a psychological incentive for developers to get started and a structure for managers along which they can orchestrate parallelization workflows.

In this paper, we present an approach for the discovery of potential parallelism in sequential programs that—to the best of our knowledge—is the first one to combine the following elements in a single tool:

1. Detection of available parallelism with high accuracy
2. Identification of code sections that can run in parallel, supporting the definition of parallel tasks—even if they are scattered across the code

3. Ranking of parallelization opportunities to draw attention to the most promising parallelization targets
4. Time and memory overhead that is low enough to deal with input programs of realistic size

Our tool, which we call DiscoPoP (= Discovery of Potential Parallelism), reverses the idea of data-race detectors. It profiles dependencies, but instead of only reporting their violation it also watches out for their absence. The use of signatures [2] to track memory accesses, a concept borrowed from transactional memory, keeps the overhead at bay without significant loss of information, reconciling the first with the last requirement. We use the dependency information to represent program execution as a graph, from which parallelization opportunities can be easily derived or based on which their absence can be explained. Since we track dependencies across the entire program execution, we can find parallel tasks even if they are widely distributed across the program or not properly embedded in language constructs, fulfilling the second requirement. To meet the third requirement, our ranking method considers a combination of execution-time coverage, critical-path length, and available concurrency. Together, these four properties bring our approach closer to what a user needs than alternative methods [3, 4, 5] do. We expand on earlier work [6], which introduced the algorithm for building the dependency graph based on the notion of computational units—at that time a purely dynamic approach with significant time and memory overhead. This is why this paper concentrates mainly on the overall workflow, including a preceding static analysis, the minimization of runtime overhead, the ranking algorithm, and an evaluation using realistic examples along with a demonstration of identifying non-obvious tasks.

The remainder of the paper is structured as follows: In the next section, we review related work and highlight the most important differences to our own. In Section 3, we explain our approach in more detail. In the evaluation in Section 4, we run the NAS parallel benchmarks [7], a collection of programs derived from real CFD codes, to analyze the accuracy at which we identify and rank parallelism in spite of the optimizations we apply. Also, we show how we find tasks that are not trivial to spot. Finally, we quantify the overhead of our tool both in terms of time and memory. Section 5 summarizes our results and discusses further improvements.

2. Related Work

After purely static approaches including auto-parallelizing compilers had turned out to be too conservative for the parallelization of general-purpose programs, a range of predominantly dynamic approaches emerged. As a common characteristic, all of them capture dynamic dependencies to asses the degree of potential parallelism. Since this procedure is input sensitive, the analysis should be repeated with a range of representative inputs and the final validation is left to the user. Such dynamic approaches can be broadly divided into two categories. Tools in the first merely count dependencies, whereas tools in the second, including our own, exploit explicit dependency information to provide detailed feedback on parallelization opportunities or obstacles.

Kremlin [5] belongs to the first category. Using dependency information, it determines the length of the critical path in a given code region. Based on this knowledge, it calculates a metric called self-parallelism, which quantifies the parallelism of a code region. Kremlin ranks code regions according to this metric. Alchemist [4] follows a similar strategy. Built on top of Valgrind, it calculates the number of instructions and the number of violating read-after-write (RAW) dependencies across all program constructs. If the number of instructions of a construct is high while the number of RAW dependencies is low, it is considered to be a good candidate for parallelization. In comparison to our own approach, both Kremlin and Alchemist have two major disadvantages: First, they discover parallelism only at the level of language constructs, that is, between two predefined points in the code, potentially ignoring parallel tasks not well aligned with the source-code structure. Second, they merely quantify parallelism but do neither identify the tasks to run in parallel unless it is trivial as in loops nor do they point out parallelization obstacles.

Like DiscoPoP, Parwiz [3] belongs to the second category. It records data dependencies and attaches them to the nodes of the execution tree (i.e., a generalized call tree that also includes basic blocks) it maintains. In comparison to DiscoPoP, Parwiz lacks a ranking mechanism and does not explicitly identify tasks. They have to be manually derived from the dependency graph, which is demonstrated using small text-book examples.

Reducing the significant space overhead of trac-

ing memory accesses was also successfully pursued in SD3 [8]. An essential idea that arose form there is the dynamic compression of strided accesses using a finite state machine. Obviously, this approach trades time for space. In contrast to SD3, DiscoPoP leverages an acceptable approximation, sacrificing a negligible amount of accuracy instead of time. The work from Moseley et al [9] is a representative example of this approach. Sampling also falls into this category. Vanka and Tuck [10] profiled data dependencies based on signature and compared the accuracy under different sampling rates.

Prospector [11] is a parallelism-discovery tool based on SD3. It tells whether a loop can be parallelized, and provides a detailed dependency analysis of the loop body. It also tries to find pipeline parallelism in loops. However, no evaluation result or example is given for this feature.

3. Approach

Figure 1 shows our parallelism-discovery workflow. It is divided into three phases: In the first phase, we instrument the target program and execute it. Control flow information and data dependencies are obtained in this phase. In the second phase, we search for potential parallelism based on the information produced during the first phase. The output is a list of parallelization opportunities, consisting of several code sections that may run in parallel. Finally, we rank these opportunities and write the result to a file.

3.1. Phase 1: Instrumentation, Control Flow Analysis, and Data Dependency Analysis

The first phase includes both static and dynamic analyses. The static part includes:

- Instrumentation. DiscoPoP instruments every memory access, control region, and function in the target program after it has been converted into intermediate representation (IR) using LLVM [12].
- Static control-flow analysis, which determines the boundaries of control regions (loop, if-else, switch-case, etc.).

The instrumented code is then linked to libDiscoPoP, which implements the instrumentation functions, and executed. The dynamic part of this phase then includes:

- Dynamic control flow analysis. Runtime control information such as entry and exit points of functions and number of iterations of loops are obtained dynamically.
- Data dependency analysis. DiscoPoP profiles data dependencies using a signature algorithm.
- Variable lifetime analysis. DiscoPoP monitors the lifetime of variables to improve the accuracy of data-dependency detection.
- Data dependency merging. An optimization to decrease the memory overhead.

3.1.1. Hybrid Control-Flow Analysis

We perform the control-flow analysis in a hybrid fashion. During instrumentation, the boundaries of control structures (loop, if-else, and switch-case, etc.) are logged while traversing the IR of the program. The boundaries are indexed by source line number, which allows us later to provide detailed information to the user. At the same time, we instrument control structures and functions.

During execution, inserted instrumentation functions log runtime control-flow information dynamically. Instrumentation functions for loops count the number of iterations, while functions for branches remember which branch is being executed so that data dependency information can be correctly mapped onto control-flow information. Instrumentation functions for function calls log the function boundaries. This is done dynamically because a function may have multiple return points and can return from different positions during execution.

3.1.2. Signature-Based Data Dependency Analysis

To lower the memory requirements of the data dependency analysis, we record memory accesses based on signatures. This idea is originally introduced in [10]. A signature is a data structure that supports the approximate representation of an unbounded set of elements with a bounded amount of state [2]. It is widely used in transactional memory systems to uncover conflicts. A signature usually supports three operations:

- Insertion: A new element is inserted into the signature. The state of the signature is changed after the insertion.
- Membership check: Tests whether an element is already a member of the signature.

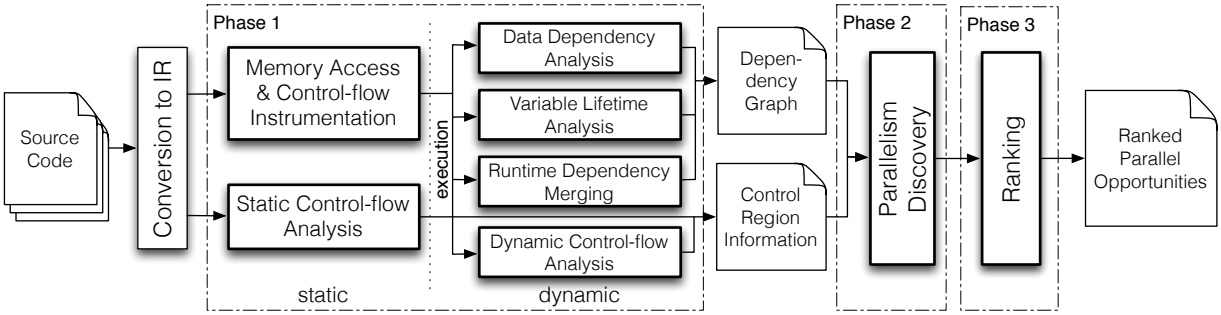


Figure 1: Parallelism discovery workflow.

- Disambiguation: Intersection operation between two signatures. If an element was inserted in both of them, the resulting element must be represented in the resulting intersection.

A data dependency can be regarded as a conflict because a data dependency exists only when two or more memory operations access the same memory location in some order. Therefore, a signature is also suitable for detecting data dependencies. In our approach, we adopt the idea of signatures to store memory accesses. A fixed-length array is combined with a hash function that maps memory addresses to array indices. In each slot of the array, we save the source line number where the memory access occurs. Because of the fixed length of the data structure, memory consumption can be adjusted as needed.

To detect data dependencies, we use two signatures. One for recording read operations, one for write operations. When a memory access c at address x is captured, we first obtain the access type (read or write). Then, we run the membership check to see if x exist in the signature of the correspondent type. If x already exist, we update the source line number to where c occurs and build a data dependency. Otherwise, we insert x into the signature. At the same time, we check whether x exist in the other signature. If yes, a data dependency has to be built as well. An alternative would be performing membership check and disambiguation whenever a write operation occurs, since read-after-read (RAR) dependencies do not prevent parallelization.

Figure 2 shows an example of how our algorithm works. The signature size in this example is four.

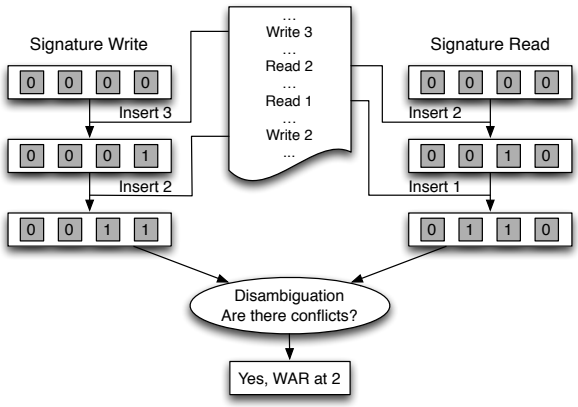


Figure 2: Signature algorithm example.

Four memory accesses are recorded, including two write and two read accesses. A disambiguation of the two signatures indicates a conflict at address 2. In this case, a write-after-read (WAR) dependency must be built.

We insert a function at the beginning of the target program to initialize the data structures. Every read and write operation of the target program is instrumented. Since disambiguation usually incurs a bigger overhead than the membership check does, we build data dependencies using membership check whenever possible.

3.1.3. Estimated False-Positive Rate

Representing an unbounded set of elements with a bounded amount of state means adding new elements can introduce errors. A signature monitors all memory accesses, which means it will not miss any data dependency (no false-negative). However,

dependencies which do not exist in the program may be built because of hash collisions (i.e., it may have false positives). For this reason, we estimate the false-positive rate as a function of the signature size and the number of elements. Assume that we use one hash function, which selects each array slot with equal probability. Let m be the number of slots in the array. Then, the probability that the hash function does not use a slot during insertion is:

$$1 - \frac{1}{m}$$

After inserting n elements, the probability that a certain slot is still *unused* is:

$$(1 - \frac{1}{m})^n$$

Now the estimated false-positive rate (EFPR), i.e., the probability that a certain slot is *used* is thus:

$$EFPR = 1 - (1 - \frac{1}{m})^n$$

Obviously, to control the false-positive rate, we need to adjust the size of the signature m to the number of variables n in the program. For example, using a fixed number of slots (800,000) for the NAS Parallel Benchmarks lets the EFPR vary between 0.01% and around 60%, wasting a lot of memory on small programs while not having enough for the big ones. To avoid such a scenario, the user can specify an maximum EFPR, and DiscoPoP will choose the size of signature accordingly. In this way, memory can be used more efficiently and the quality of the final suggestions can be assured.

To calculate the size of signature based on a given EFPR, we need to know the number of variables in the program. To avoid running the program more than once, we estimate the number of variables during instrumentation. The number is counted based on the intermediate representation (IR) of the program produced by the front-end of LLVM. Although the IR is in Static Single Assignment (SSA) form, it provides the possibility to distinguish constants, global variables, named and unnamed variables. Thus it is easy to define rules that filter out the variables that originated from the program. The rules are relaxed so that the counted number is always bigger than the real number of variables. This will result in a bigger size of the signature, leaving the actual false-positive rate usually below the specified EFPR threshold.

3.1.4. Variable Lifetime Analysis

Although false positives are a basic property of signatures and cannot be completely eliminated, we apply an optimization to lower the false-positive rate further. The main idea is to remove variables from the signature once it is clear that they will never be used again during the remainder of the execution. Thus, we need a way to monitor the lifetime of a variable. The lifetime of a variable is the time between its allocation and deallocation. The lifetime of variables has an impact on the correctness of the data dependency analysis because signature slots of dead variables might be reused for new variables. If this happens, a false dependency will be built between the last access of the dead variable and the first access of the new variable.

To resolve this problem, we perform variable lifetime analysis dynamically. This means we observe the allocation and deallocation of variables. In addition, we exploit dynamic control-flow information, which is helpful to determine the lifetime of local variables allocated inside a control region. Although there is no explicit deallocation of local variables, they die once the program leaves the control region where they have been allocated. In this way, signature slots for local variables can be reused without the danger of building false dependencies. With variable lifetime analysis, our signature algorithm can support more variables with the same amount of memory.

3.1.5. Runtime Data Dependency Merging

Recording every data dependency may consume an excessive amount of memory. DiscoPoP performs all the analyses on every instruction that is dynamically executed. Depending on the size of both the source code and the input data, the size of the file containing processed data dependencies can quickly grow to several gigabytes for some programs. However, we found that many data dependencies are redundant, especially for regions like loops and functions which will be executed many times. Therefore, we merge identical data dependencies. This approach significantly reduces the number of data dependencies written to disk.

A data dependency is expressed as a triple `<Dependent-Line, Dependency-Type, Depends-On-Line>`. Two data dependencies are identical if and only if each element of the triple is identical. When a data dependency is found, we check whether it already exists. If there is no

match, a new entry for the dependency is created. Otherwise the new dependency is discarded. For a code region that is executed more than once, we maintain only one set of dependencies, merging the dependencies that occur across multiple instances. When the parallelism-discovery module reads the dependency file, it still treats these multiple execution instances as one. For example, a loop will always be considered as a whole and its iterations will never be expanded.

The effect of merging data dependencies is significant. Previously, the size of the dependency file for the on NAS Parallel Benchmarks ranged from 330 MB to about 37 GB with input class W (6.1 GB on average). After introducing runtime data dependency merging, the file size decreased to between 3 KB and 146 KB (53 KB on average), corresponding to an average reduction by a factor of 120,685x. Since the parallelism-discovery module redirects the read pointer in the file when encountering function calls rather than processing the file linearly, data dependency merging drastically reduces the time needed for parallelism discovery. The time overhead of data dependency merging is evaluated in Section 4.

3.2. Phase 2: Parallelism Discovery

During the second phase, we search for potential parallelism based on the output of the first phase, which is essentially a graph of dependencies between source lines. This graph is then transformed into another graph, whose nodes are parts of the code without parallelism-preventing read-after-write (RAW) dependencies inside. We call these nodes *computational units* (CUs). Based on this CU graph, we can detect potential parallelism and already identify tasks that can run in parallel.

3.2.1. CU Graph and Execution Tree

A CU is built for a collection of instructions following the *read-compute-write* pattern: a set of variables (at least one) is read by a collection of instructions (i.e., source lines) and used to perform computation, then the result is written back to another set of variables (again at least one). The two sets do not have to be disjoint. The edges in the CU graph are true data dependencies (RAW). The initial concept of the CU graph is presented in [6].

The CU graph is mapped onto an execution tree, which represents the program in tree style. It is called execution tree because only executed code is included. Figure 3 shows an example execution tree. The root of the tree is the whole program. Internal nodes represent control structures, and leaves are basic blocks—plain code with no branch instructions inside. To ensure that leaf nodes represent always basic blocks, internal nodes representing empty control structures (e.g. loops without body) will have an empty child. CUs are attached to leaf nodes, and data dependencies can exist both inside and between leaf nodes.

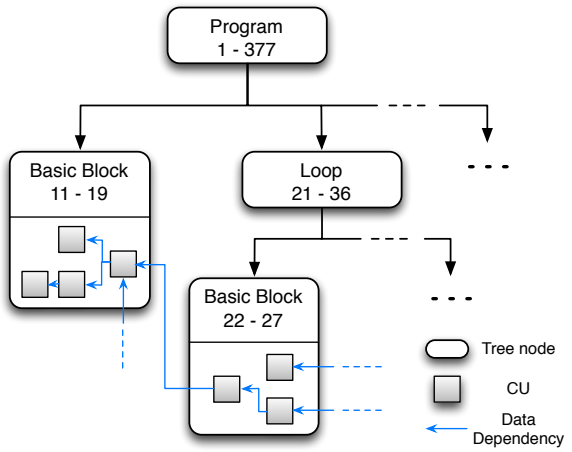


Figure 3: CU graph mapped onto execution tree.

3.2.2. Detecting Parallelism

DiscoPoP finds potential parallelism based on the CU graph. It is well known that among the four kinds of data dependencies, read-after-read (RAR) does not affect parallelization. Write-after-read (WAR) and write-after-write (WAW) are easy to resolve by privatizing the affected variables. Only read-after-write (RAW) seriously prevents parallelization.

To identify tasks, we need to find the absence of RAW dependencies. For this purpose, we concatenate nodes that are directly connected through RAW dependencies and form linear *chains* of CUs between fork and join nodes. We suggest task parallelism between independent chains of CUs, that is, without RAW dependencies between them. Recall that a RAW dependency can exist both inside a leaf node (basic block) and between two leaf nodes (between basic blocks). Thus, a chain of CUs may start and end anywhere in the program, without the limitation of predefined constructs, and the code in

a chain of CUs does not need to be continuous. A set of CU chains that can run in parallel is called a *parallelization opportunity*. To find loop parallelism, we check loop-carried data dependencies for each sub-tree of the execution tree whose root represents a loop. Again, we focus on RAW dependencies between iterations.

3.3. Phase 3: Ranking

Ranking parallelization opportunities of the target program helps users to focus on the most promising ones. Three metrics are involved: *instruction coverage*, *local speedup*, and *CU imbalance*. Currently, our ranking method still assumes that the parallelization opportunity is confined to a well-defined code section represented by a node of the execution tree.

3.3.1. Instruction Coverage

The instruction coverage (IC) provides an estimate of how much time will be spent in a code section. The estimation is based on the simplifying assumption that each kind of instruction costs about the same amount of time. Given a node i and its parent node p in the execution tree,

$$IC(i) = \frac{N_{inst}(i)}{N_{inst}(p)}$$

where $N_{inst}(i)$ and $N_{inst}(p)$ are the number of instructions node i and p contain, respectively. Note that the instruction coverage is only calculated between a node and its direct parent, and N_{inst} always represents the total number of instructions which are really executed at runtime. For a loop, N_{inst} is the sum of the number of instructions across all iterations.

3.3.2. Local Speedup

The local speedup (LS) reflects the potential speed up that would be achieved if the code section represented by a node was parallelized according to the suggestion. Since it refers only to a given code section and not necessarily to the whole program it is called local. The local speedup is based on the critical path and Amdahl's Law, which is why super linear effects are not considered. The formula used to calculate LS depends on the type of node:

Leaf nodes.. Leaf nodes are nodes that contain only plain code without any control region inside. In this way,

$$LS(i)_{leaf} = \frac{N_{inst}(i)}{length(CP(i))}$$

where N_{inst} is the total number of instructions and length(CP) is the length of the critical path the node i contains—again, based on the assumption that each kind of instruction costs the same amount of time. The formula is a direct application of the critical path.

Internal nodes.. Internal nodes represent control regions and may contain control regions nested inside. This means the local speedup of internal nodes should represent the speedup estimated for the whole control region. Thus,

$$LS(i)_{internal} = \frac{1}{\sum_{c=1}^{n} \frac{IC(c)}{LS(c)}}$$

where n is the number of children of the internal node. This is a direct application of Amdahl's Law. The local speedup is recursively calculated from the leaves towards the root of the execution tree.

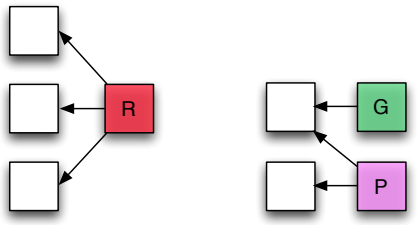
Loops.. A special case are loops. If a loop can be parallelized across iterations, a speedup of $min(N_{threads}, N_{iter})$ could be gained, where $N_{threads}$ is the number of threads, and N_{iter} is the number of iterations. In this case, the local speedup of the loop will be $min(N_{threads}, N_{iter})$. If a loop cannot be parallelized, the local speedup is calculated just as if the loop was a usual internal node.

3.3.3. CU Imbalance

The CU imbalance reflects how evenly CUs are distributed in each stage of the critical path, which means whether every thread has some work to do in each step of the computation. Otherwise, some of the threads have to wait because of data dependencies, which means the suggested parallelization may have a bottleneck. We define the CU imbalance for a node i as

$$CI(i) = \frac{\sigma(i)}{MP(i)}$$

where $\sigma(i)$ is the standard deviation of the number of CUs in each stage of the critical path, and $MP(i)$ is the number of CUs in the largest stage of the critical path of node i. The CU imbalance is a value



(a) (b)

Figure 4: Scenarios with different degrees of CU imbalance.

in $[0, +\infty)$. The more balanced the CU ensemble is, the smaller the value becomes.

Figure 4 provides an example. Under the assumption that each CU has the same number of instructions, both of situations have a local speedup of two and will complete all the tasks in two units of time. However, the arrangement in Figure 4(a) requires three threads while 4(b) requires only two. The red CU (R) in 4(a) needs the results from three CUs, constituting a bottleneck of the execution. Although the purple CU (P) in 4(b) is in the same situation, the other thread still has some work to do (green CU, G) so that it does not need to wait. The CU imbalance values of the two situations (4(a): $\sqrt{2}/3 = 0.47$, 4(b): $0/2 = 0$) reflect such a difference. Note that a code section containing no parallelism (CUs are sequentially dependent) will also get a CU imbalance of zero, which is consistent with our definition.

Our ranking method now works as follows: Parallelization opportunities are ranked by their estimated global speedup (GS) in descending order, with

$$GS = \frac{1}{\frac{IC}{LS} + (1 - IC)}.$$

Should two or more opportunities exhibit the same amount of global speedup, they will be ranked by their CU imbalance in ascending order.

4. Evaluation

We conducted a range of experiments to evaluate both the accuracy and the performance of our tool. All experiments ran on a server with 2 x 8-core Intel Xeon E5-2650 2 GHz processors with 32 GB memory, running Ubuntu 12.04 (64-bit server edition). We took our test cases from the NAS Parallel Benchmarks (NPB) 3.3.1, a suite of programs derived from real-world computational fluid dynamics applications. The suite includes both sequential and OpenMP-based parallel versions of each program, facilitating a quantitative assessment of our tool's ability to spot potential parallelism. All the benchmark programs were compiled with option `-g -O0` and the EFPR was always set to 1%. Whenever possible, we tried different inputs to compensate for the input sensitivity of the dynamic approach.

4.1. Accuracy of Parallelism Detection

The purpose of the first experiment was to see how the approximation in data dependency profiling affects the accuracy of the suggestions on parallelism. We searched for parallelizable loops in sequential NPB programs and compared the results with the parallel versions provided by NPB. Table 1 shows the results of the experiment. The data listed in the column set "Executed" are obtained dynamically. Column "# loops" gives the total number of loops which were actually executed. The number of loops that we identified as parallelizable are listed under "# parallelizable". At this stage, prior to the ranking, DiscoPoP considers only data dependencies, which is why still many loops carrying no dependency but bearing only a negligible amount of work are reported. The second set of columns shows the number of annotated loops in OpenMP versions of the programs (# OMP). Under "# identified" we list how many annotated loops were identified as parallelizable by DiscoPoP.

As shown in Table 1, DiscoPoP identified 92.5% (136/147) of the annotated loops, proving the effect of the signature approximation to be negligible. A comparison with other tools is challenging because none of them is available for download. A comparison based exclusively on the literature has to account for differences in evaluation benchmarks and methods. For Parwiz [3], the authors reported an average of 86.5% after applying their tool to SPEC OMP-2001. Kremlin [5], which was also evaluated with NPB, selects only loops whose expected speedup is high. While Kremlin reported 55.0% of the loops annotated in NPB, the top 30% of DiscoPoP's ranked result list cover 65.3% (96/147).

4.2. Identification of Tasks

Because parallelizing non-obvious tasks usually entails code re-factorization, finding such tasks is harder to evaluate than finding parallel loops that

Table 1: Detection of parallelizable loops in NAS Parallel Benchmark programs.

Program	Executed		OpenMP-annotated loops			
	# loops	# parallelizable	# OMP	# identified	# in top 30%	# in top 10
BT	184	176	30	30	22	9
SP	252	231	34	34	26	9
LU	173	164	33	33	23	7
IS	25	20	11	8	2	2
EP	10	8	1	1	1	1
CG	32	21	16	9	5	5
MG	74	66	14	14	11	7
FT	37	34	8	7	6	5
Overall	787	720	147	136	96	45

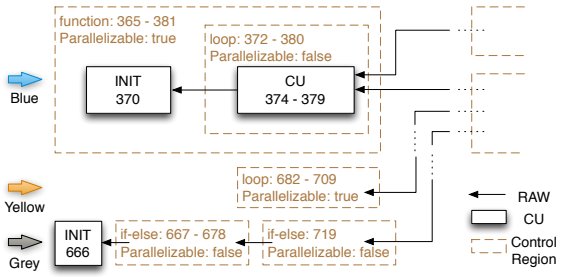


Figure 5: Identifying tasks in IS.

require at most minor modifications. For this reason, we demonstrate two real cases where task parallelism was found.

4.2.1. LibVorbis.

LibVorbis is a reference implementation of the Ogg Vorbis codec. It provides both a standard encoder and decoder for the Ogg Vorbis audio format. In this study, we analyzed the encoder part. The suggested pipeline resides in the body of the loop that starts at file encoder_example.c, line 212, which is inside the main function of the encoder. The pipeline contains only two stages: `vorbis_analysis()`, which applies some transformation on audio blocks according to the selected encoding mode (this process is called analysis), and the remaining part that actually encodes the audio block. After investigating the loop of the encoding part further, we found it to have two sub-stages: encoding and output.

We constructed a four-stage pipeline with one stage each for analysis, encoding, serialization, and output, respectively. We added a serialization stage, in which we reorder the audio blocks because we do not force audio blocks to be processed in order in the analysis and the encoding phase. We ran the test using a set of uncompressed wave files with different sizes, ranging from 4 MB to 47 MB. As a result, the parallel version achieved an average speedup of 3.62 with four threads.

4.2.2. IS.

Non-trivial tasks are also identified in a part of IS, one of the programs in NPB. Figure 5 shows the relevant part of the CU graph.

As visible in the figure, IS starts with three different paths: blue, yellow, and grey. Due to space limitations, some CUs on the yellow and the grey path are omitted. At some later point, data dependencies force the paths to converge. Before they converge, code on the different paths can be executed in parallel without violating any data dependencies. The code on the blue path belongs to a function without self-dependency, which means the function can be invoked multiple times in parallel (i.e., it is thread safe). Note that the grey path crosses a variety of constructs, including different functions. The IC values of the yellow and the grey path indicate that they consume only a small fraction of the overall runtime. The loop on the yellow path merely initializes the array based on the given input size. The code on the grey path monitors the time spent in each computational phase. Creating a task for such small amounts of work does not make sense. In addition, although no data dependency exists between them, the time measurements on the grey path depend logically on the progress of the computation on the yellow path, which is why the two paths would still need to be synchronized when being executed in parallel or why they would have to be merged, still leaving two parallel paths

in total.

Nevertheless, the function on the blue path contains a loop that produces a large set of independent random numbers, which means that parallelizing it would accelerate the process. Unfortunately, data dependencies prevent straight forward loop parallelization because every thread needs its own random seed. Yet, that the function as a whole is thread safe suggests that the work could be done in parallel provided that each thread creates a private seed first. The OpenMP version of IS confirms this conclusion. The function on the blue path was re-factored by introducing a parallel region with `#pragma omp parallel`. The parallelization made the function significantly bigger and more complex.

This example shows the power of DiscoPoP to identify parallel tasks and support the parallelization of non-trivial loops in spite of reported dependencies. Although in this specific example no practical value could be gained from knowing about all three paths, it could be demonstrated that parallel tasks can even be spread across different language constructs, including different functions, without escaping detection. We expect that this feature will ultimately turn out to be useful to find pipeline parallelism in streaming applications.

4.3. Ranking Method

We also evaluated the precision of our ranking method. The results are shown in Table 1. Column "# in top 30%" lists the number of suggestions matched by actual parallelization in the OpenMP version (# identified) that end up in the top thirty percent after ranking. We believe that only few programmers would examine all the suggestions one by one and that for most the first 30% would be the upper limit. As one can see, 70.6% (96/136) of the matched suggestions can be found in the top 30%. This means by examining only 30% of the suggestions, 70% of the actually implemented parallelism can be explored.

We also verified whether the top 10 suggestions for each program are really parallelized in the official OpenMP version. The results are listed in the column "# in top 10". For most of the programs, more than a half (for some of them even 90%) of the top 10 suggestions are parallelized, proving the effectiveness of our ranking method.

4.4. Overhead

In the last experiment, we measured the time and memory consumption of DiscoPoP, also comparing our results to other approaches. The results obtained with NPB 3.3.1 with input size W and EFPR = 1.0% are summarized in Table 2.

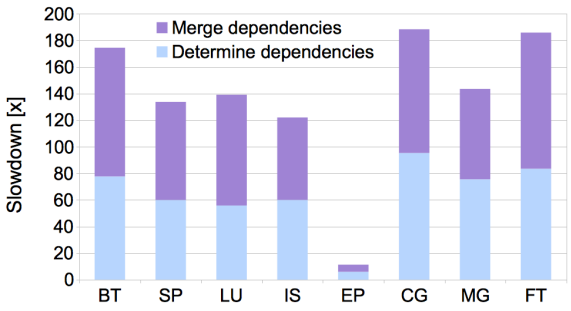


Figure 6: Runtime overhead of determining vs. merging data dependencies.

4.4.1. Time

In Table 2, the column "Original time" shows the original execution time of the programs from the benchmark suite. The column set "Slowdown" lists the slowdown attributable to the instrumentation phase (instrumentation), to the execution of the instrumented code (execution), and to the detection of parallelism (detection), always including the time spent in I/O. The memory consumption is listed in the column set "Memory".

The time needed to instrument a target program is relatively low. Given that a program needs to be instrumented only once, this overhead can be neglected. Dynamic data dependency profiling typically slows the program down by a factor ranging from 100x to 150x, where DiscoPoP has an average slowdown of 137x. Figure 6 presents a breakdown of the data-dependency analysis phase. About half (73x) of the time overhead is due to runtime dependency merging. However, it greatly reduces the time needed to detect parallelism later on. Without dependency merging, it would not be uncommon to spend several hours searching for parallelism since the dependency-file sizes easily reach an order of several GB. After applying dependency merging, the time overhead of detecting parallelism (detection) is only 2.14x on average.

We compare the overhead of DiscoPoP with overhead numbers reported for other parallelism-discovery tools in the literature. The numbers are summarized in Table 3. As claimed in [5], the code instrumented by Kremlin is about 50 times slower than `gprof`-instrumented code, whereas the code instrumented by DiscoPoP experiences an average

Table 2: Overhead when running NAS Parallel Benchmarks.

Program	Original time (sec)	Slowdown (x)			Memory (MB)		File (KB)
		instrumentation	execution	detection	original	DiscoPoP	
BT	8.21	2.89	174.48	2.52	12.91	2656.59	146
SP	24.79	3.00	133.72	1.27	37.46	161.06	105
LU	23.36	4.12	139.11	1.53	25.73	105.21	108
IS	0.14	7.00	122.00	2.00	36.34	580.23	5
EP	5.96	0.02	11.27	0.02	6.94	65.58	3
CG	0.90	0.79	188.36	0.14	33.65	106.33	13
MG	1.06	12.50	143.50	9.50	229.65	289.04	29
FT	0.87	0.43	185.86	0.15	86.14	355.13	15
Average	8.16	3.84	137.29	2.14	58.60	539.90	53

slowdown of 137x compared to the original code. Although Kremlin is faster, it lacks the ability of identifying parallel tasks because it maintains no dependency graph.

The slowdown reported for Alchemist when applied to gzip and bzip2 is 265x and 713x, respectively. The initial design of DiscoPoP [6] suffered a slowdown of 364x and 964x. However, the improved design presented here causes a slowdown of only 28x and 26x on gzip and bzip2, respectively, with the same size of input, which is about 10 to 27 times faster than Alchemist.

Parwiz [3] does not share any common benchmark programs with us. Evaluation results reported for SPEC OMP-2001 suggest that Parwiz causes an slowdown of 120x on average. Since DiscoPoP is tested on a different benchmark, we cannot directly compare the result between DiscoPoP and Parwiz. However, the result of Parwiz can still be a reference because the type of computations performed by SPEC OMP-2001 and NPB are similar, and the average numbers of source lines are also close to each other.

Another approach of dynamic data-dependency profiling is pursued in SD3 [8]. Although its primary aim is the reduction of memory overhead, they authors also quantified the time overhead. In their study, bzip2 is the only benchmark program shared with us. According to [8], the serial version of SD3 causes a 380x slowdown on bzip2, while the 8-task parallel version causes a slowdown of 95x. With the same constrains, the slowdown of DiscoPoP on bzip2 is 184x—faster than the serial version of SD3 but slower than the 8-task parallel version. Note that the time overhead of DiscoPoP on bzip2 reported here is different from the result we compared to Alchemist because the input size was different in each case.

Table 3: DiscoPoP's time overhead in comparison to other approaches in term of slowdown (x).

Benchmark	DiscoPoP	Kremlin	Alchemist	Parwiz	SD3 (seq./par.)
NPB	137	50*	–	–	–
SPEC OMP 2001	–	–	–	120	–
Gzip 1.3.5	28	–	265	–	–
Bzip2 1.0.2	26	–	713	–	–
401.bzip2	184	–	–	–	380/95

* Compared to `gprof`-instrumented code.

4.4.2. Memory

To capture the memory overhead, we use the value of "Maximum resident set size" from the `time` tool with the `-v` option specified. Memory consumption of DiscoPoP when executed with NPB is listed in the column set "Memory" in Table 2. Apparently, DiscoPoP consumes between 65.58 MB and 2.59 GB (memory consumption of the benchmarks themselves excluded), with an average of only 540 MB on NPB. The memory requirements are quite reasonable even for an ordinary computer, mainly because DiscoPoP uses a signature structure instead of a traditional double-layer table as in classic shadow memory.

As far as we know, none of the existing parallelism-discovery tools reported their memory consumption except for Parwiz. According to [3], Parwiz consumes between 200 MB and 4.7 GB (1.55 GB on average) on SPEC OMP-2001 (memory consumed by benchmark programs themselves are al-

ready excluded). As we mentioned, SD3 is well known for its low memory overhead of dynamic data dependency profiling. According to [8], the serial version of SD3 consumes about 3 GB memory on bzip2. With the same size of input and EFPR = 1.0%, DiscoPoP consumes 2.81 GB memory, slightly less than SD3.

5. Conclusion and Outlook

We introduced a novel dynamic tool for the discovery of potential parallelism in sequential programs. Building the concept of computational units (CUs) and embedding it in a framework of combined static and dynamic analysis, we can reconcile the identification of parallel tasks in the form of CU chains with efficiency both in terms of time and memory. CU chains are not confined to predefined language constructs but can spread across the whole program. Our approach found 92.5% of the parallel loops in NAS Parallel Benchmark (NPB) programs and successfully identified tasks spanning several language constructs. It also helped parallelizing a loop in spite of initial dependencies. Furthermore, we provide an effective ranking method, selecting the most appropriate parallel opportunities for the user. Our results show that 70% of the implemented parallelism in NPB can be explored by examining only the top 30% of our suggestions.

In the future, we want to extend our static analyses to reduce the number of instrumented functions—with the aim of further lowering the time and space overhead. Most important, we want to analyze CU graphs of more application examples, in particular data-intensive streaming applications, to identify patterns such as pipelining and support their implementation.

[1] R. E. Johnson, Software development is program transformation, in: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER '10, ACM, New York, NY, USA, 2010, pp. 177–180. doi:10.1145/1882362.1882400.

[2] D. Sanchez, L. Yen, M. D. Hill, K. Sankaralingam, Implementing signatures for transactional memory, in: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 40, IEEE Computer Society, Washington, DC, USA, 2007, pp. 123–133. doi:10.1109/MICRO.2007.24.

[3] A. Ketterlin, P. Clauss, Profiling data-dependence to assist parallelization: Framework, scope, and optimization, in: Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 45, IEEE Computer Society, Washington, DC, USA, 2012, pp. 437–448. doi:10.1109/MICRO.2012.47.

[4] X. Zhang, A. Navabi, S. Jagannathan, Alchemist: A transparent dependence distance profiling infrastructure, in: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 47–58. doi:10.1109/CGO.2009.15.

[5] S. Garcia, D. Jeon, C. M. Louie, M. B. Taylor, Kremlin: Rethinking and rebooting gprof for the multicore age, in: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, ACM, New York, NY, USA, 2011, pp. 458–469. doi:10.1145/1993498.1993553.

[6] Z. Li, A. Jannesari, F. Wolf, Discovery of potential parallelism in sequential programs, in: Proceedings of the 42nd International Conference on Parallel Processing, PSTI '13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 1004–1013.

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, S. K. Weeratunga, The NAS parallel benchmarks, The International Journal of Supercomputer Applications.

[8] M. Kim, H. Kim, C.-K. Luk, SD3: A scalable approach to dynamic data-dependence profiling, in: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 43, IEEE Computer Society, Washington, DC, USA, 2010, pp. 535–546. doi:10.1109/MICRO.2010.49.

[9] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, R. Peri, Shadow profiling: Hiding instrumentation costs with parallelism, in: Proceedings of the 5th International Symposium on Code Generation and Optimization, CGO '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 198–208. doi:10.1109/CGO.2007.35.

[10] R. Vanka, J. Tuck, Efficient and accurate data dependence profiling using software signatures, in: Proceedings of the 10th International Symposium on Code Generation and Optimization, CGO '12, ACM, New York, NY, USA, 2012, pp. 186–195. doi:10.1145/2259016.2259041.

[11] M. Kim, H. Kim, C.-K. Luk, Prospector: Discovering parallelism via dynamic data-dependence profiling, in: Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism, HOTPAR '10, 2010.

[12] C. Lattner, V. Adve, LLVM: A compilation framework for lifelong program analysis & transformation, in: Proceedings of the 2nd International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, CGO '04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 75–.