Discovery of Potential Parallelism in Sequential Programs

Zhen Li , Ali Jannesari , and Felix Wolf

German Research School for Simulation Sciences, Aachen, Germany
RWTH Aachen University, 52062 Aachen, Germany
{*z.li, a.jannesari, f.wolf*}*@grs-sim.de*

***Abstract*—Although multicore CPUs are dominating the market of desktops and servers, writing programs that utilize the available hardware parallelism on these architectures still remains a challenge. In this paper, we present a dynamic approach for automatically identifying potential parallelism in sequential programs. Our method is based on the notion of computational units, which are small sections of code following the read-compute-write pattern that can form the atoms of concurrent scheduling. In contrast to earlier approaches, our method can identify parallelism between code sections of arbitrary granularity and does not rely on a predefined notion of language constructs subject to parallelization. Experimental results show that reasonable speedups can be achieved by parallelizing sequential programs manually according to our findings. By comparing our findings to known parallel implementations of sequential programs, we demonstrate that we are able to detect the most important code locations to be parallelized.**

I. Introduction

Stagnating performance of uni-processors together with the prevalence of multicore processors on today's desktops and servers leaves parallelism as the the main vehicle for further improving program speed. Unfortunately, writing efficient parallel code is still an art mastered only by experienced programmers. In particular, the parallelization of the myriads of sequential legacy programs that exist today presents a major technical and economic challenge. This is even more true given that today software development is mostly the transformation of programs written by someone else rather than starting from scratch [1]. At the same time, the sheer number of sequential programs in need of further speedup provides a unique opportunity for large-scale program optimization if the available parallelism can be systematically identified and exploited. Given the task of parallelizing a legacy program, a developer has to understand the program first. Parallelization can therefore be very time consuming, especially if the program is not well documented. Tools that can identify parallelism automatically would therefore be a major asset.

Existing approaches limit the scope of their search for parallelism to predefined language constructs. For example, the method proposed in [2] is designed to find parallelism only between functions. Other approaches such as [3], [4], [5] are more flexible in that they consider multiple and also in principle arbitrary construct types. Common to all of them, however, is the restriction that they can only answer questions of the following type: (i) Can a construct or region with *given* entry and exit points be parallelized? (ii) Can a construct with *given* entry and exit points run asynchronously with other parts of the program? Thus, their underlying strategy is always *top down*. First identify the regions of investigation usually following the structure of the programming language and then reason about their parallelization. Possible answers are always confined to supported construct types and therefore do not drill deeper than the smallest level of language construct that is available. Unstructured code sections inside those constructs are not broken down any further. To do this, these methods would have to try code sections with arbitrary entry and exit points, potentially leading to a combinatorial explosion.

In this paper, we present *DiscoPoP (Discovery of Potential Parallelism)*, a tool for automatically finding potential parallelism in sequential programs. In contrast to the methods cited above, the underlying approach works *bottom up*, which eliminates the need to specify the desired granularity in advance. In essence, we reverse the idea underlying data-race detectors: Instead of looking for dependencies that jeopardize the correctness of parallel code, we verify their absence between portions of sequential code, which usually indicates an opportunity for parallelization. While the target program is executing under the control of our tool, we identify small pieces of code called *computational units* (CUs) that we arrange in a dynamic dependency graph. Since a CU does not exhibit any true data dependency inside that is essential to the data flow of the program, all dependencies that may prevent parallelization appear explicitly as edges in the graph. From this representation, we can easily derive potential parallelism available on varying levels of the code, including nested regions. As a result, we can tell the programmer which instructions (program lines) can be grouped as a task and run concurrently with other tasks. Moreover, given that CUs never cross control-region boundaries, our approach naturally supports the identification of parallelism also among higher-level constructs such as branches, loops, and functions.

We validate our approach with eight programs in two series of experiments. In the first series, we parallelize four programs manually by adopting the suggestions of our tool. The results show linear speedup for programs that expose possible data parallelism. In the second series, we compare the suggestions generated for four open-source programs with pre-existing parallel implementations. Here, we find that our tool identifies the most important source-code locations to be parallelized.

The remainder of the paper is organized as follows: In the next section, we classify our approach and review related work. In Section III, we introduce the notion of a CU, which is central to our method, and explain how we discover parallelization potential in a dependency graph of CUs. An experimental evaluation of our method follows in Section IV. Finally, we conclude the paper and discuss possible extensions and optimizations in Section V.

II. Related Work

Methods to discover potential parallelism fall into one of two categories: static and dynamic methods. Being closely related to compiler technology, static approaches analyze source or intermediate code and are restricted to information that can be obtained before running the program. Auto-parallelizing compilers such as Paralex [6], which was designed for irregular pointer-intensive programs, belong to this class. Static approaches are usually good at finding the complete control-flow graph. However, there are several disadvantages associated with them. First, when the program is large and has many branches, the solution search space becomes too big, a problem known as *branch explosion*. Second, they cannot determine dependencies among objects allocated or identifiable only at runtime. This is why static approaches are usually too conservative in their assessment of parallelization opportunities.

In contrast, dynamic methods instrument the code and track dependencies at runtime. They treat the execution of a user program as an instruction stream interrupted by previously inserted calls to instrumentation functions that help detect dependencies. Dynamic approaches identify control and data dependencies accurately, that is, only if they really exist at runtime. Since they track only precisely the branches that are actually executed, they do not suffer the branch explosion problem. On the other hand, that the control-flow graph is usually incomplete may present a limitation. In general, dynamic methods are *input sensitive*, that is, their outcome may depend on the particular execution configuration chosen to run the program, a disadvantage traded in for not being overly pessimistic like purely static methods. A straightforward compromise is selecting a range of representative inputs and repeating the analysis with all of them. Those parallelization opportunities that remain stable across all inputs are likely candidates for parallelization, but should still be subject to final validation by a human developer.

Currently, DiscoPoP follows a purely dynamic approach, but it is not the only tool based on this philosophy. Kremlin [3], which uses the LLVM compiler infrastructure [7] for instrumentation, discovers parallelism based on knowledge of the critical path and supports the programmer in ranking different parallelization opportunities. To address dependencies in nested code regions, Kremlin extends the traditional critical-path analysis [8] by making it hierarchical. For this purpose, it introduces a metric called self-parallelism, which quantifies the parallelism of a parent region independent of its children. Both Kremlin and DiscoPoP can find parallelism on different code levels. However, Kremlin can only decide whether a region between two given endpoints derived from the structure of the programming language is parallelizable or not, whereas DiscoPoP can find arbitrary parallelization opportunities even within unstructured code.

Like our tool, Alchemist [4] is built on top of Valgrind to discover parallelism and issue corresponding recommendations. In comparison to our work, the unit of parallelization is again a code region with predefined endpoints derived from the structure of the programming language. For each region, Alchemist decides whether the region can run asynchronously with its dynamic context by checking the distance between memory references inside and references to the same location that occur during the region's continuation. It thus follows the parallelization strategy underlying the use of futures. A future of a write operation is the code section or construct that contains further reads of the written variable. DiscoPoP represents once more the more general approach because of its ability to recognize arbitrarily fine pieces of independent code. We expect that this advantage will allow more specific recommendations on how to exploit parallelism and also simplify the detection of possible parallelization patterns. AutoFutures [5] adopts the same idea as Alchemist, but goes one step further in that it automatically transforms the code. Yet, it is still in a preliminary stage with negative speedup results for some of the test programs.

Further dynamic approaches are custom tailored to specific types of programs and programming languages. JavaSlicer [9] traces Java programs to find parallelism, exploiting knowledge of the critical path. ParaMeter [10] is a tool aiming to find parallelism in tasking-based applications where computational tasks are added dynamically. It employs a speculative scheduler to decide whether two tasks can be executed concurrently. Both [11] and [12] target coarse-grained pipeline-style parallelism in multimedia applications. Another approach [13] applies machine learning techniques to find parallelism in loops and to automatically parallelize them using OpenMP. In comparison, DiscoPoP's design is not limited to specific classes of applications and parallelism.

III. Approach

When trying to find potential parallelism in a sequential program, we first need to answer two questions: On which granularity level do we analyze the program and on which level do we reflect parallelism? For example, analyzing and reflecting parallelism among functions requires only coarse granularity [2]. In contrast, [4] and [5] analyze the program on the level of individual instructions and reflect parallelism in terms of futures. Obviously, the amount and type of parallelism that can be identified depends on the level we choose.

A. Computational Units

Here, we define a new language-independent code-granularity level for both program analysis and reflection of parallelism, which we call *computational unit* (CU). A CU is the smallest unit of code we map onto a thread, that is, while potentially running in parallel to other CUs, a CU itself is not subject to any further (internal) parallelization—at least not within the scope of our method. The notion of CUs was inspired by our earlier work [14], where a variation of this concept was applied to detect data races on correlated variables. The definition of a CU is simple and natural because it reflects the building blocks of most computer programs:

- CUs follow the *read-compute-write* pattern: a program state is first read from memory, the new state is computed, and finally written back. Therefore, there are no true data dependencies *inside* a CU that are relevant to the flow of data.
- All operations inside a computational unit are related through anti-dependencies (write after read).

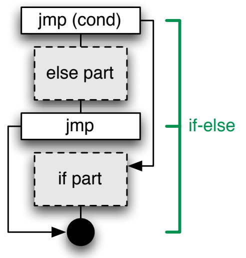
While the sequential target program is executed under the control of our tool, we read the instruction stream and track data and control dependencies. We transform the instruction stream into a graph of CUs whose edges are true data dependencies. While true data dependencies may prevent parallelization, parallelization opportunities can be derived from their absence. We argue that because CUs do not hide any true dependencies inside that are essential to the data flow of the program, all relevant parallelization opportunities can be analyzed on the level of the CU graph.

Moreover, via control-flow analysis we ensure that CUs never cross the boundaries of a control region. While being small enough, typically not covering more than a few lines of code, to express very fine-grained parallelism, this property ensures that CUs can be easily combined to higher-level constructs such as loops or functions. This allows the reflection of parallelism to be lifted to arbitrarily high levels of abstraction, making our approach most general. Currently, our implementation considers only branches and loops on such a higher level. Explicit support for other abstractions such as functions is in progress.

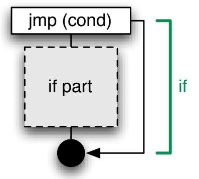
B. Control and Data Dependencies

In general, knowing control and data dependencies is necessary to decide whether a part of the program can be isolated as an independent parallel task or which kind of dependencies prevent parallelism between tasks. In particular, we need to identify control-region boundaries since CUs are not allowed to cross them.

A control dependency between two instructions op_i and op_j exists if op_j is conditionally guarded by op_i. However, to decide whether an instruction is conditionally guarded we need to know the *re-convergence point*, which is the point where the different branch alternatives end and unconditional execution resumes. To circumvent that dynamic analysis has usually no access to the complete control-flow graph because not all branches of the program are actually executed, we use a look-ahead technique. Before the real branch is executed, we follow every possible branch first and terminate this look-ahead once we encounter the re-convergence point, which is the first instruction that comes after the basic blocks defined by the branch alternatives. Looking ahead is possible since Valgrind has already disassembled the basic blocks belonging to all branch alternatives when the branch is encountered by our tool. We traverse the blocks representing the the branch alternatives without actually executing them, simply following jump instructions until we find the re-convergence point. The look-ahead is performed in a special mode where we only track control dependencies while deferring the tracking of data dependencies. An example of finding the re-convergence point of an `if-else` and a simple `if` statement is shown in Figure 1.



(a) `if-else` construct



(b) `if` construct

Figure 1. Finding the re-convergence point (solid black circle).

We instrument jump operations and maintain a stack where we record the scope of the currently active control regions. When we encounter a control region, we push a triple $< start, type, end >$ onto the stack. When we leave a control region, we remove the topmost entry. Then we determine the type of region (branch or loop) and its re-convergence point using our look-ahead technique. We also respond to function calls. If a function is called inside a control region, we simply keep the top of the stack unchanged and continue pushing control regions we find in the callee on the stack. When the callee function returns, all control regions it contains should also terminate and the calling region is again on the top of the stack.

The example shown in Figure 2 illustrates our algorithm. It contains several control and data dependencies. Applying our algorithm for finding re-convergence points to the example yields Figure 3, where the re-convergence points (solid black circles) are exactly the first lines encountered after the corresponding control structure ends.

```
1   for (i = 0; i < MAX_ITER; i++) {
2       if (i == 0)
3           x = 3
4       a = x + rand() / x
5       b = x -  rand() / x
6       x = a + b
        ...
k   }
```

Figure 2. A simple code example.

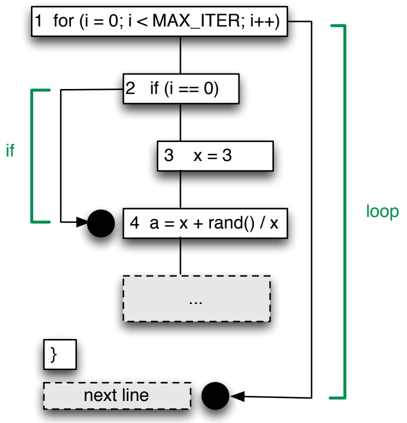


Figure 3. Re-convergence points of the example in Figure 2.

Data dependencies fall into one of four categories: true dependencies (read after write or short RAW), anti-dependencies (write after read or short WAR), output dependencies (write after write or short WAW) and input dependencies (read after read or short RAR). We insert instrumentation calls for all reads and writes of both memory and registers and record data dependencies in shadow memory. For every variable, we maintain a field called *last write operation*, which is updated whenever the variable is written. Table I shows the result of our data dependency analysis when applied to the example from Figure 2. Together, control and data dependencies form the prerequisites needed to identify computational units.

Table I
DATA DEPENDENCIES OF THE EXAMPLE IN FIGURE 2.

Line No.	Code	Memory Ops.	Dependencies
3	x = 3	init x	-
4	a = x + rand() / x	init a, read x	x:RAW
5	b = x - rand() / x	init b, read x	x:RAW
6	x = a + b	write x read a read b	x:WAR a,b: RAW

C. Building the CU Graph

Given that we have now both control and data dependencies, the isolation of CUs becomes straightforward. The basic idea is that a write operation is much more important than a read operation when trying to find parallelism. Whenever a write operation happens, we check the dependencies of the written variable and start observing its further use. Based on these criteria, CUs can be computed automatically using the online algorithm below, which we apply recursively to the operations of the current control region, starting with the main function.

Imagine the execution of the program as a sequence of dynamic operations (instructions) $\{op_0, op_1, \ldots, op_{n-1}\}$. Let R be the subsequence of these operations that belong to the current control region, not including operations belonging to regions nested inside. Let us assume we have already processed all operations in R up to but not including op_i. Then we can apply the following algorithm to op_i and all remaining operations in R:

1) When we encounter a new control region nested inside the current one, we suspend the current region until we have processed the region nested inside.
2) When an operation $op_i \in R$ is executed:
    - If the variable v that op_i operates on is defined in other variable's use, ignore op_i and exclude dependencies on v for building CUs. Otherwise we build a CU that just contains op_i.
    - We *merge* the CU of op_i with all the CUs of dynamic operations $op_{j<i} \in R$ that op_i directly depends on via anti-dependencies.
    - If op_i directly depends on $op_{j<i}$ via a true data dependency, we create a directed edge from the CU of op_i to the CU of op_j, expressing that op_i truly depends on op_j. Note that op_j does not necessarily have to be an element of R.
    - If op_i is the first write of a variable in R, we mark it as *initialization*.
    - Repeat the algorithm for all remaining operations in R.
3) At the end, *merge* the CUs of all adjacent *initialization* operations into one INIT node. Adjacent means that their operations form a contiguous subsequence of R.

Note that we ignore intermediate variables that are introduced to make the code more readable, but which are not essential to describe the state of the program. Considering them would increase the granularity of CUs to the level of individual instructions, where reasoning about parallelism in a meaningful way becomes difficult for the programmer. In particular, the boundaries of computational units could break individual source-code statements. When op_i is ignored because it operates on v, which is defined in w's use, the association of op_i with the current CU of w (i.e., the CU which only contains read operations of w, waiting for a write operation) will be saved. When representing CUs later on the source-code level, this association will allow the full code coverage to be restored, including the code for op_i, as we will see later in an example.

Moreover, a CU is not able to cross control-region boundaries. Of course, dependency edges between different control regions are possible. Furthermore, if a variable is written several times and no read occurs in between, the first write is used to terminate the CU and the last written value is kept. All write operations in between are ignored, since their values are overwritten. Finally, all read-after-read dependencies are ignored.

As an example, consider again the code inside the loop in Figure 2. Figure 4 shows the identification process. On the left side, we find control and data dependencies. From line 4 to line 6, x is read before a new value for x is computed and eventually written. Note that line 6 is data dependent on both line 4 and 5 in the form of a WAR dependency. According to our algorithm above, we build a CU for the computation of x. The first write of x in line 3 is identified as INIT node. INIT is the only special node type found in a CU graph.

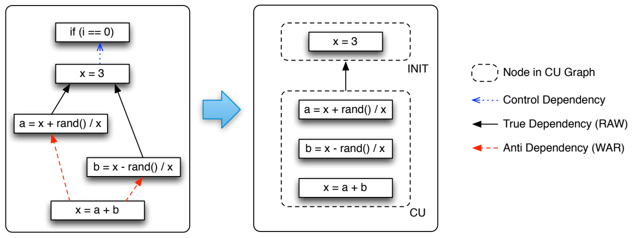


Figure 4. Identifying CUs.

To exclude temporary (intermediate) variables (i.e., variables defined in another variable's use) when building a CU, we trace the *definition-use chain* [15] for each variable dynamically. A definition-use chain consists of a definition D of a variable and all its uses U. U is reachable from the definition D without any other intervening definitions. The term definition here refers to the assignment of some value to a variable. If a variable x_2 is defined in the use of another variable x_1, we do not build a CU for x_2 until x_2 is defined in no one else's use. In our example, a and b are defined in x's use, and they are temporary variables for updating x. Therefore, we do not build a CU for them.

D. Detection of Parallelism

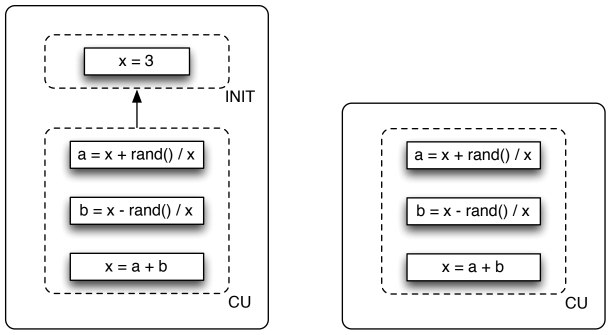
Having identified all relevant dependencies, we can now form the CU graph. Given the graph and the defined terminology, we present the following rules for detecting parallelism:

1. If CU_2 truly depends on CU_1, then CU_1 must be executed before CU_2. They cannot run concurrently.
2. Different paths of the CU graph can be executed in parallel after inserting a synchronization barrier into the common node where these paths join.
3. A loop in the user program cannot be parallelized if there is a CU inside the loop that holds a variable without an INIT node preceding it.

The reason for rule 1 is simple. A read instruction has to wait until the latest value has been produced, otherwise the behavior of the program will be nondeterministic. The rationale behind rule 2 is also obvious. Two CUs reside on different paths because there is no dependency between them. However, synchronization barriers must be inserted into the node where the two paths join, since this common node depends on the values produced by the two paths.

Rule 3, however, is not straightforward. Suppose a CU holding variable v exists inside a loop but without any INIT node preceding it. This implies that every time when the loop starts a new iteration, the first access of v will be a read operation. During the first iteration, it should read the value produced outside the loop. But during later iterations, it should read the value produced by the previous iteration. In this situation, iteration $(n+1)$ must wait until iteration n has produced the latest value. However, if an INIT node exist at the very beginning, v will be assigned a new value in each iteration, and the iterations of the loop become independent. Therefore, rule 3 means that we propose the parallelization of the loop only if its iterations are independent of each other. Note that we do not treat WAW dependencies between iterations as an obstacle on the way to parallelization because resolving WAW dependency is usually easy. In addition, adding synchronization primitives to protect the shared data structure will not significantly lower the speedup if the amount of work in each iteration is big enough. In this way, we still report loops having only WAW dependencies between iterations as parallelizable.

Let us use the former example again to see how the above rules work in practice. When the loop finishes the first iteration, the partial CU graph of the first iteration is shown on the left side of Figure 5. Rule 1 applies here: Since the CU truly depends on the INIT node, the CU node must be executed after the INIT node. This means that no parallelism is discovered inside the loop. Then the loop iterates again. The partial CU graph for the second iteration is shown on the right side of Figure 5. Now we apply rule 3: There is a CU inside the loop that holds a variable without preceding INIT node. This means this loop cannot be parallelized between iterations. From the third iteration on, iterations are merely repeated, which is why we refrain from showing them.



(a) 1^{st} iteration (b) 2^{nd} iteration

Figure 5. The first two iterations of the example in Figure 2.

Now we explain how we can use rule 2 to identify parallelism between CUs. We can find task parallelism in code sections even if they contain no control region. Our tool simply tells the programmer which source lines can be grouped as a task. To determine paths in the CU graph that can run concurrently, we treat the sequence of executed CUs as a time-ordered sequence with dependencies, and transform it into *Foata normal form* (FNF) [16]. FNF is a normalized representation of traces in trace theory. There, a trace is defined as an equivalence class of strings according to a given independence relation. The FNF divides a trace in stages based on the independence relation, and every stage is a set of letters sorted from left to right by cardinality. We consider every CU identifier as a letter of the string with all available CU identifiers forming the alphabet. With this property, we are able to determine the maximum number of tasks that a certain code section can be divided into (the maximum size of FNF stages). This is exactly what we want in rule 2.

```
1     if (i == 0) {
2         x = 3
3         y = 4
4     }
5     a = x + rand() / x
6     b = x -  rand() / x
7     x = a + b
8     a = y + rand() / y
9     b = y -  rand() / y
10    y = a + b
```

Figure 6. Finding task parallelism.

Whereas we could not identify any potential parallelism in the loop example, we demonstrate how we can eventually find parallelism between CUs using a similar example, which is shown in Figure 6. The new example extends code from Figure 2 by introducing another variable y, but with the calculations of x and y being independent. The corresponding CU graph is depicted in Figure 7. Although CU_x and CU_y both depend on the INIT node, they are independent from each other. The CU graph can be expressed as string $s = "INIT,\ CU_x,\ CU_y"$, with the alphabet $\Sigma = \{INIT, CU_x, CU_y\}$ and the dependency relation $D = \{(INIT, CU_x), (INIT, CU_y)\}$. We place CU_x in front of CU_y because the calculation of x starts earlier than the calculation of y. The FNF of the string $"INIT,\ CU_x,\ CU_y"$ will be $\{INIT\}$, $\{CU_x,\ CU_y\}$. According to the FNF representation, we conclude that we must run the INIT node first, after which we can run CU_x and CU_y in parallel. Since we maintain the source-code location of each CU, we are able to suggest to the programmer that line 5, 6 and 7 can form one task, while line 8, 9 and 10 can form another.

Note that both variables a and b are defined in another variable's use. Hence, we refrain from building CUs for

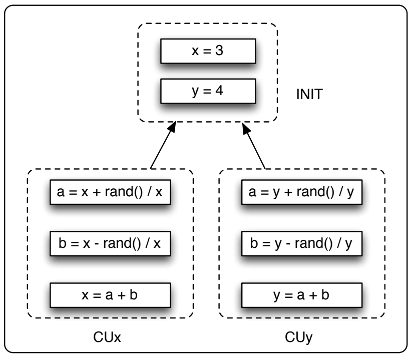


Figure 7. CU graph for the example in Figure 6.

them, as we explained in Section III-C. Now, we are able to find parallelism between CU_x and CU_y even if there are data dependencies on variables a and b. CU_x and CU_y can be easily parallelized by making a and b private to each thread.

Finally, we would like to point out that it is straightforward to find parallelism also on higher levels. For example, a function is a subgraph of the whole CU graph. Treating every subgraph representing a function as a single node and only considering dependencies between these function nodes will yield the CU graph on the level of entire functions. The above rules still apply. Currently, the highest levels we support are loops and branches.

Of course, not all parallelization opportunities that we identify are worth pursuing. Right now, we assess the significance of our findings based on a very coarse cost model, taking into account the number of instructions inside a CU and the number of iterations in the case of a loop.

E. Implementation

DiscoPoP is implemented on top of Valgrind 3.7.0, an instrumentation framework for building dynamic analysis tools [17]. Valgrind first translates the binary code to a special intermediate representation called VexIR. Before Valgrind executes the program on a synthetic CPU, DiscoPoP automatically inserts instrumentation to track memory accesses. Hence, DiscoPoP does not require any annotations of the source code.

IV. Evaluation

To validate our approach, we (i) measure the speedup achieved for smaller test applications after adopting suggestions made by DiscoPoP and (ii) compare DisoPoP's suggestions to known parallel implementations of larger applications. We perform our tests 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). All test programs are compiled with GCC 4.6.3. Whenever possible, we try different inputs to diminish input sensitivity.

Table II
SPEEDUP ACHIEVED WHEN ADOPTING DISCOPOP'S SUGGESTIONS.

Benchmark	LOC	Input size	Number of suggestions	# Adopted	Seq. time (s)	Par. time (s)	Speedup (4x)
histogram	102	50,000,000 numbers	5	1	0.36	0.098	3.67
mandelbrot	521	1024 x 1024 matrix	2	2	46.02	22.73 (11.61)	2.02 (3.96)
light propagation	74	500,000 random points	1	1	5.67	2.33	2.43
ANN training	107	50 x 500 x 4 matrix	10	2	5.11	1.66	3.07

A. Speedup Potential of DiscoPoP's Suggestions

In the first series of experiments, we use DiscoPoP to find potential parallelism in four test programs, which were created as laboratory-course assignments. We parallelize the programs manually by adopting the suggestions generated by our tool and measure the speedup we gain. The parallelization is either based on Pthreads or OpenMP. Although dynamic program analysis is usually input sensitive, the simplicity of the test programs lets the output of our tool remain stable under varying inputs. The inputs are chosen to cover all possible branches. Table II summarizes our results. Values shown in the table are averages of five runs. Details for each programs are discussed below.

Histogram visualization: This program receives an array whose elements can belong to N different types and sorts them into buckets, putting data with type N_i into the i^{th} bucket. The items in every bucket are counted to produce the histogram. We use this example to illustrate details of the suggestions produced by DiscoPoP, which are shown in Table III. Our tool successfully finds the main computational loop starting at line 46 as a good candidate to be parallelized. The loop iterates over the input array with no data dependencies inside, indicating the numbers in the array can be processed in parallel. The structures identified by suggestions ranked 2, 3, and 4 (suggestions are ranked by #iterations * size in terms of #instructions) are also parallelizable, but belong either to the initialization or output stage and do not promise significant speedup for larger input problems. Moreover, we do not follow suggestion 5 because the loop contains only one line without function call and iterates four times. To measure the speedup, we use an array of 50,000,000 numbers as input. The serial version of the program runs in 0.36 seconds, whereas the parallel version with four threads runs in 0.098 seconds, resulting in a speedup of 3.67.

Mandelbrot set: The Mandelbrot set is the set of values c in the complex plane for which the orbit of zero under iteration of the complex quadratic polynomial $z_{n+1} = z_n^2 + c$ remains bounded. Our test program produces a 1024 x 1024 resolution image for the Mandelbrot set. The program iterates over rows and columns, checking whether a point belongs to the set. The problem exhibits a high degree of data parallelism, since every point on the plane can be examined independently. Our tool reports that the innermost loop starting at line 27 cannot be parallelized because of

Table III
SUGGESTIONS FOR HISTOGRAM VISUALIZATION.

Rank	Location	# Iter.	Size	Adopted	Reason
1	line 46	50	6 lines	Yes	-
2	line 21	50	3 lines	No	init.
3	line 54	53	1 line	No	output
4	line 34	50	1 line	No	output
5	line 44	4	1 line	No	too small

RAW dependencies between iterations, involving variables z_{real} and z_{imag}. This loop iterates 50,000 times at most to test whether the complex number $z_{real} + z_{imag}i$ satisfies the equation. However, the outer loops starting at line 16 and 18 are reported as parallelizable. The loop starting at line 16 iterates over the rows of the matrix, and the loop starting at line 18 over its columns. We parallelize the program with Pthreads by dividing the matrix among four threads. While the serial version of the program takes 46.02 seconds, the parallel version takes 22.73 seconds, resulting in a speedup of 2.02. With the fastest thread running only 0.15 seconds, the disappointing speedup is the result of imbalanced workload. After introducing a dynamic load-balancing scheme, the four threads consume about the same time, resulting in an almost linear speedup of 3.96.

Simulation of light propagation using Monte Carlo: This program simulates light propagation from a point source in an infinite medium with isotropic scattering using the Monte Carlo method. Photons are modeled as pairs of randomly produced numbers and each photon is simulated independently. Nevertheless, a global array of $heat$ must be calculated. It is therefore possible that two photons write the same element of the $heat$ array. When the number of photons is small, our tool reports that the loop that iterates over the photons can be parallelized because the probability of writing the same element of the heat array is low. However, when the number of photons increases, the loop is reported as unparallelizable. After getting this report from DiscoPoP, we parallelize the main loop using OpenMP, with the array $heat$ as shared variable. We protect each element of $heat$ with a separate lock. We run the parallel version with four threads. This simple approach results in a slowdown of 15.75. The serial version runs only 5.67 seconds, but the parallel version runs 89.28 seconds. After replacing all occurrences of $rand()$ with a thread-safe alternative $rand\_r()$, the adjusted parallel version runs 2.33 seconds, resulting a speedup of 2.43.

Artificial Neural Network training: The Artificial Neural Network training algorithm adjusts the weight matrixes of the network by iteratively examining training data provided as input. Because new weight values always depend on their former values, it is hard to run different iterations in parallel. However, during the same iteration, it is possible to parallelize the calculation of the weight matrix in one dimension. Our tool successfully points to two loops starting at line 50 and line 81, respectively. Both of them iterate along one dimension of the weight matrices. Adopting the suggestion from DiscoPoP, we parallelize the training program using OpenMP and run it with four threads. Because the training algorithm usually needs quite a long time to reach convergence if it reaches it at all, we took the liberty of placing an upper bound on the number of iterations to make the program terminate in reasonable time. Our neural test network comprises 50 x 500 x 4 neurons. The serial version runs 5.11 seconds, while our parallel version runs 1.66 seconds, resulting in a speedup of 3.07. This is actually quite close to the results in [19]. In [19], three parallel versions of ANN training are tested and approach B is almost the same as when following DiscoPoP's suggestions. The test of approach B in [19] with 40 x 100 x 10 neurons using OpenMP resulted in a speedup of about 3.

B. Comparison to Known Parallel Implementations

Whereas the former section presented smaller examples that can be quickly parallelized, our second series of experiments compares the recommendations DiscoPoP generates for four larger test programs to their known parallel implementations. Our goal is to see whether DiscoPoP proposes the same parallelization strategies. Two of the test cases belong to the Parsec Benchmark suite [23] with parallel versions already included in the suite. The two other programs are well-known open-source programs, of which popular parallel implementations exist [4], [20]. Given that the efficient parallelization of larger programs usually involves major refactoring beyond eliminating a few dependencies, as shown in an empirical study [21], we refrain from creating parallel versions ourselves this time. Instead, we just verify whether DiscoPoP reproduces the ideas underlying the already existing parallel versions.

Table IV contrasts the recommendations of DiscoPoP with their main parallelization approach. The number of iterations tells how many times the structure specified as matching suggestion is executed. The size corresponds to the number of source lines this structure covers. We also estimate the effort to parallelize the suggested structure manually by assigning values ranging from "low" to "high". In the future, we plan to provide such an estimate automatically. Although the suggestions obtained for different inputs vary slightly, the output is stable enough to justify our conclusions. The details of each test case are discussed below.

parsec.blackscholes: Blackscholes, which calculates the price for a portfolio of European stock options using the Black-Scholes partial differential equation, is part of the Parsec benchmark suite. The suite includes three parallel versions based on Pthreads, OpenMP, and Intel TBB, respectively. Since it comes closest to the serial program, we picked the OpenMP version for our study. To reduce the time needed for our analysis, we run blackscholes using the "test" input available in Parsec. The only place parallelized in the OpenMP version is a loop starting at line 238. Given its extent (20 lines of code) and the number of iterations (400), it is easy to identify this loop as the most promising target among the suggestions generated by DiscoPoP. Iterating over the lines of input and solving the equation, it represents the main computation and therefore consumes most of the execution time. DiscoPoP also discovers another potential parallel loop starting at line 233 (100 iterations and 22 lines of code), which runs the test NUM_RUNS times as the basis of the performance statistics. Since it belongs to the Parsec framework and bears no relation to the actual application, we refrained from parallelizing it.

parsec.streamcluster: Streamcluster, a computational kernel that clusters continuously arriving data points into k clusters, is another program from the Parsec benchmark suite. It comes with two parallel versions based on Pthreads and TBB. We let the program randomly produce a small number of points as input. In the Pthreads version, threads are created at line 1723, which is surrounded by a loop starting at line 1714. In the serial version, this loop calls a function to cluster the points. DiscoPoP successfully classifies this loop as parallelizable. However, the existing parallel version is not as simple as blackscholes and includes synchronization primitives such as barriers. In addition to this loop, our tool makes fifteen other proposals. After a thorough check, we find that they are also parallelizable. Nevertheless, these locations usually consist of small loops not covering more than one or two lines and without function calls inside. Neither are they parallelized in any of the existing parallel versions nor do we we believe that the small amount of work they perform justifies parallelization.

gzip 1.3.5: gzip is a widely used file-compression tool and pigz [22] a popular parallel implementation based on Pthreads. In gzip, files are broken down into blocks, and the algorithm iterates over blocks, compressing them one by one. We run gzip 1.3.5 under the control of DiscoPoP in order to see whether DiscoPoP's recommendations match the locations parallelized in pigz 2.2.4. In the output of DiscoPoP, we find that the loop starting at line 1595 iterates 284 times while other structures are usually executed not more than ten times. Although four dependencies are reported inside the loop, the fact that it contains more than 100 lines of code and iterates 284 times makes it an attractive parallelization target. After analyzing the code in detail, we realize that all four dependencies refer to global variables, which are used when compressing individual file blocks. Based on these insights, we think that in spite of the four dependencies this structure is worth to be parallelized, given

Table IV
DISCOPOP'S SUGGESTIONS COMPARED TO KNOWN PARALLEL IMPLEMENTATIONS.

Benchmark	Number of Suggestions	Location Parallelized in Parallel Implementation	Matching Suggestion	Details of Matching Suggestion		
				# Iterations	Size (# Lines)	Effort
blackscholes	2	blackscholes.c: 238	blackscholes.c: 238	400	20	Low
streamcluster	16	streamcluster.cpp: 1723	streamcluster.cpp: 1714	5	8	Medium
gzip 1.3.5	43	pigz.c: 1478	gzip.c: 1595	284	101	High
bzip2 1.0.2	62	bzip2smp.c: 81	bzip2.c: 3793	104	34	High

the large amount of work it performs. And indeed, the loop is parallelized in pigz. In function parallel_compress() at line 1478 in pigz.c, pigz breaks the input into blocks of 128 KB and compresses them concurrently. However, this function does more than what has been suggested by our simple discovery. It also calculates the individual check values for each block in parallel, and contains some optimizations for parallel IO. Nevertheless, the main idea of the underlying parallelization strategy is correctly identified. Note that DiscoPoP also lists other interesting places as potential parallelization targets. For example, there is a loop in the main function starting at line 3400, which iterates over user input files after processing user options. Obviously, it would also be a good parallelization candidate since compressing different files exhibits data parallelism. But it would require some effort to resolve dependencies, since the buffers in the sequential program are reused. pigz does not parallelize this part. In the parallel implementation suggested by [20], this part is also identified.

bzip2 1.0.2: bzip2 is another well-known compression tool. A number of parallel implementations exist, but their approaches differ. We chose bzip2SMP [24], a parallel implementation based on Pthreads, for comparison. Our analysis with DiscoPoP indicates that the loop starting at line 3793 inside the function $handle\_compress()$ iterates hundreds times and consumes 83% of the function's execution time. Although several dependencies prevent parallelism, all of them can be located in the structure $EState * s$. The loop contains two parts: one for the preparation of a new block and the other for the compression of the block. They exchange state information through s, leading to a RAW dependency between iterations. By examining the call tree starting from $handle\_compress()$, we find calls to $BZ2\_compressBlock()$ and $BZ2\_blockSort()$. Dependencies inside them are also anchored in the structure s, since the pointer of s is passed to these two functions as a parameter. According to our understanding of the original bzip2 algorithm, we believe that by duplicating the $EState$ structure the block sorting stage of the pipeline can be parallelized, which means that the blocks of a file can be compressed in parallel. Bzip2SMP adopts exactly the same idea. The function performing the parallel block sort is $threadFunction()$ starting at line 81 in bzip2smp.c. However, the real parallel strategy is much more complex than we expected. The same parallelization target was also found in [4].

C. Performance of the Analysis

To evaluate the performance of DiscoPoP itself, we run each benchmark five times and calculate the average of both the time elapsed and the memory consumed. As can be seen in Figure 8, the slowdown caused by DiscoPoP varies between 206x (histogram) and 1189x (streamcluster). However, most of the cases (five out of eight) stay under 400x, which is normal compared to other approaches. For example, the slowdown reported for Alchemist [4] ranges from 166x to 712x. The highest slowdown rates exhibit light propagation, bzip2, and streamcluster.

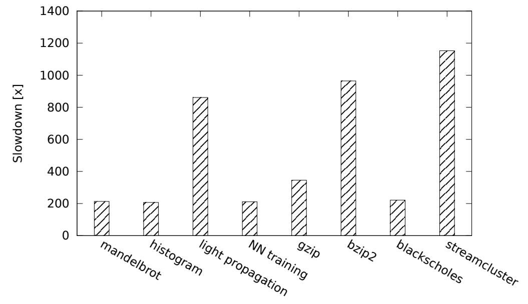


Figure 8. Slowdown when using DiscoPoP.

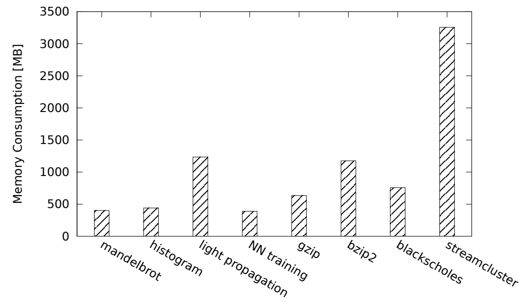


Figure 9. Memory consumption of DiscoPoP.

Figure 9 shows the memory consumption of our tool, which ranges from 399MB (ANN training) to 3.33GB (streamcluster). The memory consumption shares the same general trend with the slowdown rates. The three programs consuming more than 1GB memory are again light propagation, bzip2 and streamcluster. The reason why these three programs suffer from both higher slowdown rates and higher memory consumption are loops with many iterations. Our approach currently produces CUs and builds dependencies for every iteration anew, which is often not necessary. For example, a preceding static dependency analysis could already identify loop patterns that allow the number of iterations we need to analyze at runtime to be restricted to just a few. This would seriously speedup the analysis and reduce the size of the CU graph.

Although the memory consumption is large in absolute terms, it is actually small in comparison to traditional shadow memory techniques based on the pairwise method. For example, for bzip2 pairwise shadow memory consumes about 4GB memory, and native shadow memory consumes more than 10GB [25]. We use less memory ($<$ 1.5GB for bzip2) mainly because apply the analysis region by region, reclaiming the memory used for previously processed regions. We designed our shadow memory in the form of a multi-layer table that allocates memory only for addresses that are actually being used. We also implemented a simple run-time garbage collector that returns addresses of unreferenced variables back to the free list.

V. Conclusion and Outlook

In this paper, we introduced a novel dynamic approach to the discovery of potential parallelism in sequential programs, which rests on the notion of computational units. Recommendations on how to parallelize the program are derived on-the-fly from a trace-representation of their dependencies in Foata normal form. A major advantage of this very general concept is that it allows arbitrarily fine code sections that can run concurrently with each other to be precisely delineated—opening the way for very specific instructions on how to parallelize the program. In comparison to previous approaches, DiscoPoP does not restrict the analysis to language-defined code levels. Our experiments confirm that our method can find the most important parallelization opportunities, leading to competitive and in some cases even near-linear speedup when being implemented.

Nonetheless, several enhancement opportunities arise. In general, we believe that combining our method with static techniques will substantially reduce the time and space overhead. Moreover, designing the shadow memory in a more efficient way could reduce the memory footprint. Further efforts will be directed towards a more precise estimation of parallelization effort and expected speedup in order to automatically identify the most profitable parallelization opportunities. Overall, we believe that the work presented here will provide the foundation for a both comprehensive and practical tool that can significantly help programmers parallelize large numbers of sequential legacy codes.

References

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

[2] S. Rul, H. Vandierendonck, and K. De Bosschere, "Function level parallelism driven by data dependencies," *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 55–62, Mar. 2007.

[3] S. Garcia, D. Jeon, C. M. Louie, and 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*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 458–469.

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

[5] K. Molitorisz, J. Schimmel, and F. Otto, "Automatic parallelization using autofutures," in *International Conference on Multicore Software Engineering, Performance, and Tools (MSEPT'12)*, May 2012, publication.

[6] H. Vandierendonck, S. Rul, and K. De Bosschere, "The paralax infrastructure: Automatic parallelization with a helping hand," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 389–400.

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

[8] M. Kumar, "Measuring parallelism in computation-intensive scientific/engineering applications," *IEEE Trans. Comput.*, vol. 37, no. 9, pp. 1088–1098, Sep. 1988.

[9] [Online]. Available: http://www.st.cs.uni-saarland.de/javaslicer/

[10] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval, "How much parallelism is there in irregular applications?" in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 3–14.

[11] G. Tournavitis and B. Franke, "Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 377–388.

[12] W. Thies, V. Chandrasekhar, and S. Amarasinghe, "A practical approach to exploiting coarse-grained pipeline parallelism in c programs," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 356–369.

[13] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle, "Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 177–187.

[14] A. Jannesari, M. Westphal-Furuya, and W. F. Tichy, "Dynamic data race detection for correlated variables," in *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ser. ICA3PP'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 14–26.

[15] M. J. Harrold and M. L. Soffa, "Efficient computation of interprocedural definition-use chains," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 2, pp. 175–204, Mar. 1994.

[16] G. Rozenberg and A. Salomaa, Eds., *Handbook of formal languages, vol. 3: beyond words*. New York, NY, USA: Springer-Verlag New York, Inc., 1997.

[17] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, Jun. 2007.

[18] [Online]. Available: http://www.cmiss.org/openCMISS/wiki/

[19] A. Strey, "A comparison of OpenMP and MPI for neural network simulations on a sunfire 6800," in *PARCO*, 2003, pp. 201–208.

[20] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang, "Software behavior oriented parallelization," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 223–234.

[21] V. Pankratius, A. Jannesari, and W. F. Tichy, "Parallelizing bzip2: A case study in multicore software engineering," *IEEE Softw.*, vol. 26, no. 6, pp. 70–77, Nov. 2009.

[22] [Online]. Available: http://zlib.net/pigz/

[23] [Online]. Available: http://parsec.cs.princeton.edu/index.htm

[24] [Online]. Available: http://bzip2smp.sourceforge.net/

[25] M. Kim, H. Kim, and C.-K. Luk, "Sd3: A scalable approach to dynamic data-dependence profiling," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 535–546.