Using Static Code Analysis to Improve Coarse Task Granularity in Bobox

Lukas Krizik, Filip Zavoral and Miroslav Cermak
Charles University in Prague, Czech Republic
Email: krizik,zavoral,cermak@ksi.mff.cuni.cz

Abstract—Long-running tasks inhibit a parallel execution in some cases. A finer task granularity can significantly improve execution times in parallel environments. That is even more significant when using cooperative scheduling and it puts higher requirements on user code. In this paper we present a method of user code optimization using static code analysis in order to reduce long-running tasks. The method is based on evaluation of runtime code complexity and yielding a task execution in an appropriate place.

Keywords—static code analysis; optimization; code complexity; parallelism

I. INTRODUCTION

The Bobox scheduler [5] is a cooperative scheduler used in the Bobox framework [1], thus applications based on it inherit its behavior (see Sec. I-B). The efficiency of programs running in a parallel environment with cooperative scheduling tightly depends on the quality of a user code. A task must finish or give up its execution in order to execute a different task on the same CPU. If tasks depend on each other in some way, they should keep balanced execution times as much as possible. Too short execution times cause that scheduling exceeds the real execution in the CPU consumption [8]. On the other hand, long execution times can inhibit parallelism by keeping dependent tasks out of an execution. Long execution times of tasks producing data can also congest framework internal structures.

The optimization method presented in this paper aims to handle long execution times of the tasks. The main goal is to detect such complex tasks and yield their execution in appropriate places in the code. To achieve this, it is necessary to analyze execution paths, in our case by traversing Control Flow Graph (CFG), and to evaluate code complexity of each path.

A. Related Work

There are multiple ways how to measure the code complexity. Ideally, if the input data are known, the program can run using this data and its performance is measured during the execution. This process is called profiling. The quality of optimizations based on profiling is very high but it requires a human to analyze data and update the code. Most popular compilers provide a features for optimizations based on profiling data, Profile Guided Optimizations (PGO) [7], [10], but this approach cannot replace the higher level look on algorithms used in an application provided by a programmer.

Another approach for measuring the complexity is to compile the code and consider the number of generated instructions as the magnitude of the complexity. This assumption is naive and imprecise. The main source of the complexity in all applications comes from loops, repeatedly executed code paths, and this information is not present in this metric. For snippets of a code without a loop, this method is precise enough even if there are multiple execution paths. In bigger code samples, there is probably a bigger amount of loops, thus the code is more complex. Nevertheless, the number of reflected instructions is less precise.

The complexity measurement can be also based on a source code indentation. A bigger maximal indentation usually means more complex code. This approach makes strong assumptions about source code formatting and its precision tightly depends on how the source code follows company programming rules often contain a rule related to the concept called cyclomatic complexity [9]. It measures the logical complexity as the number of linearly independent paths through the code. This optimization method aims to reduce a different kind of the code complexity. More code branches decreases readability for a human, but it has almost no effect on runtime.

B. Bobox Framework

Bobox is a framework [1], [2], [6] for task-based parallel programming [4]. The framework uses a number of worker threads, each one with its own cooperative scheduler. Communication between tasks uses a column-based data model, the most significant implementation detail that favors data processing problems. A task is scheduled to be executed when it has an unprocessed input. The runtime environment handles the implementation details of the task-based parallel environment such as scheduling and the parallel execution of tasks, data transport and control...
flow. A programmer uses a declarative way to setup the environment with a model which defines the way individual tasks are interconnected. The model is used to create a model instance which is used for execution of the tasks connected by the data streams.

The basic element of a model instance is a task called a box. Boxes are executed in three steps: (1) in prologue the box provides access to its inputs for the user code; (2) action is the main place for user code; (3) epilogue handles the scheduling of the next task based on two criteria - a task is scheduled again if it has an unprocessed input and it has processed some input in the action step or if it requested to be scheduled again.

Boxes are main objects of the interest for optimization because they are the main location for the user code. Based on the static analysis of the action step, additional code can be injected in order to inform Bobox about the properties of the task.

II. YIELD COMPLEX METHOD

Control Flow Graph (CFG) is used to represent user code. CFG is a graph representation of computation and control flow in the program. The nodes are basic blocks, usually straight-line, or single-entry code, with no branching except at the end of the sequence. The edges represent possible flow of control from the end of one block to the beginning of the other. There may be multiple incoming/outgoing edges for each block. The optimizer has to be able to estimate the complexity of all paths in the CFG passing through every graph node to find the proper place for a call to the yield operation. The goal is to cut long execution paths exceeding some predefined threshold.

For simplification, the optimizer can assume that all paths have the same complexity and probability. Since multiple execution paths often pass through a specific code block represented by a graph node, insertion of the yield call into this block cuts all paths passing through this block. Such reduction of the running time of a single path can cause too short execution times of some other paths; therefore, the benefit could be lower than the cost.

Figure 1 shows an example of the described situation. The single long execution path is the thinner dashed curved path. Thicker lines represent multiple short execution paths. The yielded block cuts many short paths in order to get rid off the single long path. More short paths mean a lower probability of the long path being taken, thus a lower probability of the optimization benefit. Furthermore, it causes more scheduling; it can even slow down the overall performance.

A. Block Complexity

In order to measure complexity of paths in CFG, the optimizer must be able to measure the complexity of the basic construction elements of a path, a code block. Only single execution path passes through a code block. Statements in a code block are executed one by one. When a control flow enters the block, each statement is executed exactly once until the control flow exits the block.

TABLE I. Example of the estimated complexities of statements in a block.

<table>
<thead>
<tr>
<th>Entry</th>
<th>many short</th>
<th>long</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>yield</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(assuming that no exception occurs). For such a code block, the best approach to measure the complexity is an approach similar to measuring the complexity of a code by the number of generated instructions.

Code blocks consist of statements. Most statements generate zero or more instructions with constant execution time. Problematic statements are call expressions because they effectively transfer the execution out of CFG. Since their complexity is unknown it has to be estimated. We assign fixed (i.e., parametrized - the optimizer implementation allows a user to change all values by providing a custom configuration) estimated values for different types of call expressions. A block complexity is a sum of complexities of all statements in this block.

Since trivial call expressions generate no instructions, their default complexity is zero. A compiler evaluates constant call expressions during a compilation and does not generate any instructions for them, thus their default complexity is zero. The complexity of inlined functions and the general form of call expressions cannot be stated precisely. Although complexities of such functions vary, their values are usually in some reasonable sized range. Values for inlined call expressions and the rest of call expressions in Table I were calculated based on statistics gathered from a code used in benchmarks, see Section IV. The function complexity is calculated as a number of statements in its body.

B. Path Complexity

A code block is the basic construction element of a code path. Using the definition of the code block complexity, it is possible to define a code path complexity as well. Every CFG constructed in the analyzer contains two specially designed empty blocks, i.e., Entry and Exit blocks. Control
flow enters the graph through the Entry block and leaves the graph through the Exit block. However, if the CFG contain loops, there is an infinite number of paths from Entry to Exit blocks.

Therefore, the loop bodies are evaluated as independent CFG making source CFG acyclic. When a path enters a node with a loop statement as a terminator, this path creates a new path for every path in the loop body. The path that skips a loop body is omitted from the analysis. Usually, this path has a very low probability, but it affects the results significantly. New paths behave as they skip a loop body, but their complexity is a sum of the source path complexity, the block complexity and the body path complexity multiplied by a predefined constant from Table II.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>for</td>
<td>5</td>
</tr>
<tr>
<td>while</td>
<td>15</td>
</tr>
</tbody>
</table>

Figure 2 shows an example of a part of CFG with a for loop statement and an if selection statement in the body of this loop. The numbers next to the edges represent paths complexity and the numbers in graph nodes represent blocks complexities. There is a single path with the complexity of 5 entering the node with the for statement terminator. The loop body itself contains two different paths with complexities of 2 and 3. Two paths leaving the block with the for statement terminator have complexities of 16 and 21.

The calculation for those paths is the for loop multiplier * the body path complexity + the entering path complexity + the complexity of the block with the for statement terminator.

**C. Quality of CFG**

The algorithm for reduction of complex paths tries to inject the yield function call into some CFG block to make it better than source CFG. The algorithm needs a value to quantify the CFG quality for comparison of original and optimized graphs.

The goal of this optimization method is to reduce a number of complex paths in long-running tasks. For simplification, any path with complexity bigger than some predefined constant is considered to be too complex. Figure 3 shows one approach for the CFG quality evaluation. The top horizontal line represents either Entry block or a block with the yield call expression, i.e., a block where a control flow enters CFG. The vertical lines represent different paths and line lengths represent paths complexity. The threshold horizontal line represents the constant complexity value for distinguishing too complex paths. The parts of paths exceeding the threshold are highlighted, these parts are called penalty. Thus, the goal of the optimizing algorithm is to minimize the length of penalties.

**III. Optimizing Algorithm**

Using a metric for the CFG quality, it is possible to describe the algorithm for the yield complex optimization formally and in more details, see Figure 5. The algorithm runs the optimization step on CFG until this optimization step returns CFG with a better quality. The distance variable is a sum of distances of paths complexity from a threshold value, see Figure 5. This approach takes into an account also the disadvantage of placing the yield call expression into some block. The optimizer achieves very good results with this proposed metric of the CFG quality, see Section IV.
optimization step places zero or one yield into data representing CFG. Data returned from the optimization step is then evaluated and tested whether its quality increased.

1) \( \text{cfg} = \text{Build CFG data.} \)
2) \( \text{distance} = \text{Calculate the quality of } \text{cfg}. \)
3) \( \text{temp cfg} = \text{Run the optimization step on } \text{cfg}. \)
4) \( \text{temp distance} = \text{Calculate the quality of } \text{temp cfg}. \)
5) If \( \text{temp distance} < \text{distance} \) then a) \( \text{distance} = \text{temp distance}. \)
     b) Swap \( \text{cfg} \) with \( \text{temp cfg}. \)
     c) Continue in the step 3.
6) Else finish, \( \text{cfg} \) is optimized.

A. Calculation of Complexity and Quality Evaluation

The optimizer uses the depth-first search algorithm to analyse CFG and calculate complexities. The analysis has to handle yield calls in blocks which are already in the code or which are planned for insertion by the optimizer. The CFG quality evaluation gets paths and their complexities as an input and calculates a value representing the CFG quality described in Section II-C.

B. Optimization Step

The only goal of the optimization step is to decrease the distance value representing the quality of CFG. First, it collects all blocks where at least one path ends, i.e., the Exit block and the blocks with the Planned or the Present yield state. Then, it processes every block with at least one path with complexity higher than the threshold value and calculates what happens if the yield call expression is placed into that block. A block with the best outcome has its yield state set to Planned.

The complexity of the algorithm tightly depends on the complexity of the user code and on the structure of its CFG. Every block is visited twice in each optimization step. All paths complexities are recalculated for every block with at least one path with complexity higher than the threshold value. However, the next optimization step is executed only if the previous optimization step has changed the yield state of one block into the Planned state.

Thus, the optimization step has increased a number of paths, but greatly decreased a number of blocks with too complex paths.

C. Default Threshold

Estimation of the default threshold value is crucial for a performance of the optimization and it is usually task dependent. To determine its value for benchmarks, we did measurements on non-optimized code using various input data. Then we analyzed result, with focus on typical run time of the tasks to not create too short task and to split just enough tasks to potentially fill gaps in the computation.

D. Code Injection

The last step of the optimization process is the injection of the yield call expression to blocks with the Planned yield state. A block with the yield state can be empty, it can contain a single statement in a condition expression of a selection statement, the right-hand side expression in a binary expression, the else branch in a conditional expression, or any other language structure where the C++ language grammar does not allow to chain statements.

The easy solution is to find a compound statement where the injection of the yield call is as good as the injection directly to the chosen block. The injection of the yield call expression into a compound statement is then simple and safe.

First, the optimization method collects all compound statements in the function body compound statement. Then, the optimizer collects all blocks with the Planned yield state. For every collected block, the optimizer checks all compound statements whether the relation of the collected block and the compound statement matches any of special cases (see Figure 6).

1) If the block with the Planned yield state is the condition expression of the if, switch or while statement which is a child statement of the compound statement, the yield call expression is injected just before the if, switch or while statement.
2) If the block with the Planned yield state is the initialization statement or the condition expression of the for statement which is a child of the compound statement, the yield call expression is injected just before the for statement.
3) If the block with the Planned yield state is the incremental expression of the for statement and the compound statement is its body, the yield call expression is injected as the last statement of the compound statement.

IV. Results

The goal of this optimization method is to reduce a number of long-running tasks. Such tasks can inhibit a parallel execution, thus vastly increase execution times. On the other hand, long-running tasks are more cache friendly, however the speedup from a parallelism should be much bigger than the one from a better cache usage.
if/switch/while (cond) {}

for (init; cond; inc)
{
...
}

Fig. 6. The injection of the yield call for some special cases of statements.

A. Model

Our benchmarks were based on a model with one important task. This task is essential for a parallel execution and it should run as often as possible. Other long-running tasks prevent scheduling of this essential task, thus they inhibit parallelism. We choose this model since the scenario is typical for long-running parallel data processing.

Figure 7 shows an example of a model with one important task. The source box is a box that computes data for a parallel processing and distributes this data to worker boxes. Worker boxes process data in parallel. If there is the same number of worker boxes as the number of logical threads, there is a big possibility that the scheduler immediately schedules all worker boxes to all logical threads. The worker box is the long-running task. Thus, all worker boxes keep the source box out of an execution. This source box creates the bottleneck for a parallel execution.

B. Benchmarks

The results were measured on a parallel environment with 8 logical threads. Therefore, the model consists of 8 stateless worker boxes and single source box. The worker box calculates for 10 seconds and the source box calculates for 5 seconds. This optimization method causes the worker box to yield after 5 seconds.

The first results were a bit different from the expected outcome. If the source box is the bottleneck of an execution, it should inhibit the parallel execution for 5 seconds each iteration. So the speedup for ten iterations should be approximately 50 seconds. The measured result was approximately 20 seconds.

If worker boxes finish approximately at the same time, the source box is scheduled and until it finishes, nothing runs in parallel. But, if one worker box yields its execution and let the source box to run, data for the next bunch of parallel worker boxes are ready sooner. Even thought the particular worker box ends later, it does not inhibit parallelism as there are always data for other workers. A bigger task granularity greatly helps in this particular model example.

The problem was at the implementation of the source box for these particular measurements. The source box calculates something for five seconds, then it sends data to all worker boxes and it yields immediately. But, the Bobox scheduler does not schedule all worker boxes immediately after the source box yields. The source box was often scheduled with the bunch of worker boxes and delays an execution of one of them. However, there was still a significant speedup in the execution.

The source box has to do some additional work before it yields to achieve the described behavior. It gives time to the Bobox scheduler to schedule seven worker boxes and the yield in the source box schedules the last worker box. Figure 8 shows measured results with modified implementation and the speedup approximately matches expected numbers.
V. Conclusion

Long-running tasks can occur in a longer development process when programmers extend a task to do more and more work with increasing requirements for application features. There may be various reasons such as a lack of time or even laziness to write the code directly to an existing task. A programmer can also develop a long-running task because the work done by this task is monolithic and it is natural to write the code in a single function. The programmer may not realize consequences of such code to parallelism; he may unintentionally omit to yield its execution.

The finer granularity can greatly improve execution times when using cooperative scheduling. It is necessary to balance the granularity, as too small or too coarse has negative impact to performance. This paper contain a continuation of our previous work [8] that was focused on short running tasks. In this paper, we propose an optimization method for the opposite problem: long running tasks that can create bottlenecks for some scenarios with cooperative scheduling.

A. Future Work

The current state of the optimization method can be still improved. Some indications about further improvements were already mentioned, e.g., producers can congest internal framework structures, or different path probabilities.

1) Runtime Checks: If auto-vectorization back-end optimizers cannot prove that operations in a loop body do not overlap in the compilation process, they generate runtime checks and both versions of a loop, original and vectorized. There is a lot of estimations such as call expression complexities or loop body multipliers. It would be possible to handle suspicious cases more precisely using runtime checks.

2) Probabilities: The optimization method considers that all paths in CFG have the same probability. It is a simplification. For example, functions often contain multiple checks of their arguments on the beginning of their bodies, but the analysis can assume that these branches will not be taken in majority of function calls because inputs are expected to be correct. Developers of branch predictors on modern processors confront the similar task [13]. Some ideas for a branch prediction could be reused for the static analysis, but most mechanisms used for predictions are based on runtime information.

3) Identify Producers: There are multiple drawbacks of long-running tasks mentioned in the introduction of this chapter. One of them is a possible congestion of framework internal structures. The analysis can assign more weight to paths that produce data for other tasks in order to ease their yield. Loops producing data deserves more recognition than loops performing calculations.

Acknowledgments

This work was supported by the Czech Science Foundation (GACR), project P103-13-08195S, by Charles University Grant Agency (GAUK) project no. 472313, and by Specific Research project SVV-2015-260222.

References