Inherent Parallelism and Speedup Estimation of Sequential Programs

Sesa Kalyur* and Nagaraja G.S

*CSE Dept. R.V. College of Engineering, Bangalore, India
SesaKalyur@Outlook.Com, nagarajags@rvce.edu.in
*Correspondence: SesaKalyur@Outlook.Com

Received: 29th January 2021; Accepted: 12th March 2021; Published: 1st April 2021

Abstract: Although several automated Parallel Conversion solutions are available, very few have attempted, to provide proper estimates of the available Inherent Parallelism and expected Parallel Speedup. CALIPER which is the outcome of this research work is a parallel performance estimation technology that can fill this void. High level language structures such as Functions, Loops, Conditions, etc which ease program development, can be a hindrance for effective performance analysis. We refer to these program structures as the Program Shape. As a preparatory step, CALIPER attempts to remove these shape related hindrances, an activity we refer to as Program Shape Flattening. Programs are also characterized by dependences that exist between different instructions and impose an upper limit on the parallel conversion gains. For parallel estimation, we first group instructions that share dependences, and add them to a class we refer to as Dependence Class or Parallel Class. While instructions belonging to a class run sequentially, the classes themselves run in parallel. Parallel runtime, is now the runtime of the class that runs the longest. We report performance estimates of parallel conversion as two metrics. The inherent parallelism in the program is reported, as Maximum Available Parallelism (MAP) and the speedup after conversion as Speedup After Parallelization (SAP).

Keywords: Estimation; Parallel; Performance; Prediction; MAP; SAP

1. Introduction

Performance study of a program whether serial or parallel in nature involves one of the following methods: through code Analysis, by Profiling or with the help of detailed Simulation. Each technique has its pros and cons, and one of them is chosen based on when we need the information [1].

The universal method, of estimating the performance of a program, is the wall clock method, where the time spent by the program, from start to finish, provides the measure. But when computers of different speeds are involved, a little more work is needed, in the form of converting, run times to normalized cycles, before we can compare. When we need fine grained performance, we can use specialized counters, to further our quest. However, empirical studies of program performance are biased towards the choice of input samples used, which is an inherent limitation of this method.

As an alternative, study of program characteristics, through static analysis, is encouraged. The process seems simple, but tricky, since the cycles, are hidden in program structures, such as Procedures, Loops, Recursion and Conditions to name a few. This is even more evident, when we undertake performance study, of parallel programs and serial programs that are scheduled for, parallel conversion. It is an unfortunate paradox that, the syntax features of an imperative language, designed to boost programmer productivity, can be a hindrance to quality analysis and performance studies. We are at the mercy of Analysis phases later on in the compilation chain to

supply the information for estimation. Many of these phases also perform non trivial program transformations to assist the analysis step further, reducing the relevance of an estimation phase. If performance estimates are available early, they could be used to determine, the choice of transformations to apply. How do we get past this dichotomy? By realizing that syntactic structures are the cause, and finding a cure for it. From the perspective of a modern imperative language, this means cleaning up syntax through Procedure Expansion or Function In-lining, Loop Unrolling, Recursion to Loop Conversion, and Control Predication prior to the analysis and study phase.

Performance estimation and prediction of code that is free of syntactic structures of high level languages are easy. Thus, converting code with these structures to sequential code is the first step in our measurement process. We use a process called Program Shape Flattening, to eliminate the estimation hurdles. These syntactic structures, their number and placement which add a unique character to the program under study together, is referred to as the Program-Shape.

Next, we use the concept of Equivalence Class to solve the central problem that is addressed in the paper namely, the coarse assessment of parallel performance and providing estimation and prediction to programmers. We define Equivalence Class as a class that holds objects that share a common property. In the current context, it holds program statements that share dependency between them. We call such a class as a Parallel Equivalence Class or Parallel Dependence Class. Together, the Parallel Equivalence Classes that belong to a program hold all the statements in the given program. These Parallel Equivalence Classes can be run in parallel and hence the name. The number of Parallel Equivalence Classes and the instructions belonging to each are good indicators, of the parallel behaviour of the program. A large number of Parallel Equivalence Classes with less number of statements in each indicates that the given program is parallel conversion friendly.

Finally, we define ready to remember and easy to use parallel performance indicators to aid the parallel programmer referred to as, Maximum Available Parallelism which in short-form is referred to as MAP and Speedup After Parallel Conversion which is abbreviated as SAP. The sections which follow shall provide details of our research activities and their outcomes.

The paper is organized as follows: Section 2 which follows, examines the state of the art, in the domain of performance assessment in general, and parallel measurement in particular. Section 3 briefly looks at Asterix, our parallel compiler and transformation infrastructure. Section 4 discusses in detail, the workings of CALIPER, which is an important piece, in the overall solution, provided by Asterix. Section 5 presents CALIPER in action, from the concept of an example program, in a higher level, imperative language. Section 6 is dedicated to Competitive Analysis, where CALIPER is compared against other state-of-the-art solutions. Finally we conclude the paper, after highlighting the contributions of our work, with the research community, in perspective.

2. Previous Work

Early Parallel Conversion of programs was entirely a manual activity. Parallel code paths in the program were identified and each path was handed out to a task. Tasks were implemented as fill fledged Processes or Threads. The latter being the more efficient Counter-part in terms of resources consumed [2]. Procedure is error-prone and tedious and so research was carried out to seek better techniques.

The next step in the evolution of Parallel Programming was the advent of special parallel languages or existing language constructs, offered as directives to the compiler and parallel conversion supervised by the programmer [3-6]. Notable among them are the OpenMP and MPI which are also industry-standard technologies [7-10].

Automated Parallel Conversion arrived later with static analysis as the basis for generating information about the program, such as Flow and Dependence analyses, which provided the impetus for transformations. Analysis techniques were algebraic such as Linear or Polyhedral, and algorithmic such as Trees and Graphs [11-19]. While translating high-level languages to an intermediate representation (IR) and transforming and optimizing the IR is the norm, several researchers have tried the source-to-source conversions as a basis for optimizations and parallel conversions [20-31].
Researchers encountered irregular programs next which were hard to analyze statically. Efforts to augment static with runtime data were started, which was possible through the Sampling and Profiling activity [32-39]. This led to a burst in new research projects. However sampled data is comparatively biased towards the inputs used and the program coverage accomplished [40-47]. A few others used both static and dynamic data for analysis purposes [48-51].

Performance estimation and measurement are important from two angles. Measurement done early in the compilation cycle can aid the choice of optimization and conversion techniques. Measurement done later in the pipeline can be more accurate and can help ascertain the quality and accurateness of earlier projections. A lot of research has been expended in the area of performance assessment, including parallel performance [52-64].

3. Asterix

Caliper is a parallel measurement, prediction and estimation module. It is part of the compilation pipeline, of Asterix our compiler, optimizer and parallel converter. We provide a high-level view, of each of the Asterix modules next:

3.1. Paracite

This module is essentially, the front end of Asterix, where the lexical analysis, syntax analysis and semantics analysis occur. The input to this phase is the program in an imperative language, and the outcome of the phase, is the equivalent program in ASIF, the Intermediate Representation (IR) in Asterix [65].

3.2. ASIF

ASIF is an acronym and stands for Asterix Intermediate Format the language that mainly includes an IR instruction set invented for the Asterix compiler suite. It is based on the three-address instruction format, with explicit Operand followed by the Result, And two Source operands.

3.3. Caliper

Caliper reads the code in the ASIF format, and does a coarse estimation, of the nascent parallel opportunities, that exist in the given program. This provides a starting point, for the users, to position their reference performance. The following section discusses exhaustively on the topic. [1].

3.4. Graft

Graft performs the bulk of the analysis work, on the IR code in ASIF format. The result of the analysis is represented in the form of several tables and graphs which are consulted, for identifying code transformation opportunities, including optimizations and parallel conversions.

3.5. 3PO

3PO stands for Parallel Performance Predictor and Oracle. This module is a fine grain, performance estimation and prediction module, which reports at the local block level, and also at the global program level and uses several mathematical models, one for each transformation category, for its operation. The various 3PO sub-models are categorized based on the nature of the transformation, or parallel conversion. Accordingly, we have transformations that improve instruction counts, transformations that improve cache latency, and transformations that enable other transformations including parallel conversions [66].

The main performance numbers reported are, Inherent Parallel Potential (IPO) and the Expected Speedup from Parallel Conversion (ESP) with obvious connotations for parallel conversion. For transformations, the numbers are similar but with slightly different semantics, and they are, Inherent Speedup Potential (ISP) and the Expected Speedup from Transformation (EST) using the appropriate category model.
3.6. Transgraph

Transgraph module is in charge of generating code transformations that are beneficial, from a performance perspective. Some of the transformations are solely concerned, about generating code that is parallel friendly. The input and output for the module, is IR in ASIF code, and supplementary IR structures data such as graphs and tables.

3.7. Paragraph

Paragraph module actually generates the parallel code. The basic unit of parallel code which is conceptually a task is called a Prune after morphing the phrase, Parallel IR Unit. Each Prune is assigned, to an independent processing element, in a virtual topology and this mapping is preserved, for the entire duration, of the application existence. The input for the module is IR code and IR supplements, from Transgraph. Output is IR in Prune form.

3.8. Pigeon

Pigeon is a word that originates from the phrase, Parallel Code Generator. It is the module that converts Prunes, to executable versions of Prunes. These executable Prunes are called Proxies, singular is Proxy. The name evolved from the phrase, Parallel Execution Unit, are generated and assigned, to respective execution units, in an actual physical topology in a later phase. These mappings are subject to change, during the life cycle of the application.

3.9. AIDE

AIDE stands for, Asterix Integrated Development Environment, is a graphical tool to display the important results, of the compilation process, starting from the source code, to the generation of Prunes and Proxies and their interdependence [67]. The various views include, Annotated Source and ASIF IR, Caliper Predictions, 3PO Oracles, Prunes, Proxies, their distribution and orchestration.

3.10. Concerto

This module as the name suggests is the Distributor, Coordinator and Orchestration Manager of the Proxies in action. It chooses the mapping of Proxies to their respective processing elements manages their remote executions and also provides synchronization primitives. In a NUMA distributed environment, it also decides on how to partition data, between the Proxies, manages mapping to processing elements and provides communication primitives for data sharing [68]. Actual mapping is handled by a sub module of Concerto called the Topology Mapper, TOPMAP for short and offers a choice of, different mapping algorithms. [69-70].

![Figure 1. Phases of ASTERIX Compiler Chain][1].

The Figure-1 illustrates the different phases involved in the operation of the Asterix compiler in pictorial form and is intuitive for the most part. Readers can correlate the figure with the description immediately above.
CALIPER module, is responsible for providing the user, with a base expectation of parallel performance that is inherent in the program, under consideration. This prediction can help dictate, the choice of transformations to apply on the program, including the parallel conversion decisions. The higher-level syntactic structures, of an imperative program, offer impedance, to the effective computation of, performance estimates, and prediction. Each program is unique, from the perspective of the collection of the syntactic structures, constituting the program, which offer unique difficulties, for estimation and prediction. We refer to this trait of the program, as the Shape of the program. The transformations applied to a program, to strip the Shape of a program as the Program-Shape-Flattening.

Input to the CALIPER module, consists of IR in ASIF format. It performs the following, Program-Shape-Flattening transformations such as, Function-Call-Expansion, Loop-Unrolling and Control-Predication, which are described individually later. The output from the CALIPER module is the performance estimation, in the form of Maximum-Available-Parallelism (MAP), and the performance prediction, in the form of Speedup-After-Parallel Conversion (SAP). These two terms, are described later. The following paragraphs describe the steps involved in CALIPER operation followed by the definitions of Performance Metrics reported by CALIPER.

4.1. Function Call Expansion

The purpose of Function-Call-Expansion is to replace, all function calls, with the code, that constitutes the function block. It should be noted that, it is a recursive process, and the process stops only, after all user defined functions, have been expanded. Library Functions and System Calls are normally not considered for call expansion. They are essentially treated as any other instruction, which suffices for coarse estimates. A user program that is loaded with library calls and system calls may skew the prediction somewhat, but it is usually not the case, with a majority of the real-world programs.

4.2. Loop Unrolling

As a result of Loop-Unrolling, all Loops and Multi-Loops are replaced with their respective code blocks, and the instructions making up the Entry, Exit Conditions and the Loop Back Jumps removed.

4.3. Control Predication

Control Predication is a transformation that replaces Conditional Blocks, with equivalent Predicated Blocks. The Conditional Statements are another hindrance, to the correct estimation, of performance. However, most of the architectures provide support for Predicated-Execution of instructions, with varying degree of support. However all of them support Conditional-Move instruction which is a powerful construct when used with predicates, to compute the condition of the move, and combined with regular instructions, computing to temporary result variables, offer a powerful and compelling solution, to implement Control-Predication.

4.4. Maximum Available Parallelism

Maximum-Available-Parallelism (MAP) is a metric that reports the amount of parallelism present, in a given program, as a percentage. For instance, a MAP of 33% means that, one third of the code is parallel convertible, and the other two thirds of the code, 66% is serial in nature. It should be noted, that this number, takes in to consideration, all the dependencies, that exist in the program, which includes, both the data, and the control kinds.

4.5. Speedup After Parallel Conversion

Speedup-After-Parallel Conversion (SAP), is a metric that reports the benefits of parallel conversion. In the example discussed earlier, since 33% is subject to parallel conversion, the
effective run time is determined by the 66% of the serial part, and the expected speedup, would be 1.52 and reported as a floating point number.

The Figure-2 illustrates the different steps involved, in the operation of the CALIPER module. As you can see, translated IR code in ASIF format is fed to the Inliner module, which carries out the expansion of all function calls, and this modified IR is fed to the next module in the chain, which is the Unroller. This module unrolls all loops, and its output is sent to the next module in the chain, which is the Predicator. The purpose of this module is to convert all conditionals in the IR to Predicated statements. The output from this module, is shape sanitized IR that is ready for performance estimation.

**Figure 2.** CALIPER operation [2]

### 4.6. Performance Estimation Equations

Performance estimation and prediction, for both serial and parallel versions, revolve around the following parameters, which are defined below, and also given are the equations for computing them.

#### 4.6.1. Serial Execution Cycles

Since we are measuring performance, in coarse fashion here, we are not accounting, for the individual instruction differences. Each instruction counts as one cycle, and we are also not considering, the memory hierarchy, into these computations. Fine grained estimations, are for a later pass, where they use the 3PO model which has an in built cycle accurate simulator, we call Kinetics, for accurate estimates. It includes hardware accurate models of cache, memory and storage supporting the simulator. The workings of 3PO and Kinetics, are subject matter of a different paper, and we shall not discuss them any further here. The following equation, describes the process, for the equation for Serial-Execution-Cycles:

\[
C_{\text{SER}} = N_{\text{INC}}
\]

(1)

Here, \(C_{\text{CYC}}\) is the count of cycles, to run the serial version of the program, and \(N_{\text{INC}}\) is the instruction count, for the given program.

#### 4.6.2. Parallel Execution Cycles

Computation of the parallel execution cycles, is more involved, and requires a check, for data dependence between operands and results, belonging to different instructions. Since we have eliminated, control dependencies of all kinds, through Shape-Flattening, this is not an issue any more. A later subsection, shall describe the Shape-Flattening algorithm in more detail. Calculating Parallel-Execution-Cycles involves, classifying instructions, based on their data dependence, into different equivalence classes. Instructions belonging to the same equivalence class are data dependent with one another, and so we have to honour, their ordinal order of issue, to maintain correctness. However instructions belonging to different classes, have no data dependencies, and hence allow concurrent execution between them. Once the equivalence classes, have been finalized, the execution time is dictated by, the longest running equivalence class. The algorithm for creating equivalent dependence classes shall be given later in a following subsection.

The equation for computing, the parallel execution cycles, is given below,

\[
C_{\text{PAR}} = \text{MAX}(EQC_1, EQC_2, \ldots, EQC_n)
\]

(2)

Where \(C_{\text{PAR}}\) is the parallel cycle count, \(EQC_1, EQC_2, \ldots, EQC_n\) are the total cycles needed to execute the, individual equivalence class instructions in serial fashion.

The equation to compute Maximum Available Parallelism (MAP) is given on the following line:

\[
\text{MAP} = \left(\frac{C_{\text{SER}} - C_{\text{PAR}}}{C_{\text{SER}}}\right) \times 100
\]

(3)

Where, Maximum Available Parallelism (MAP) is a measure of the inherent parallelism available in a program, and is reported as a percent of the total program instructions. \(C_{\text{PAR}}\) is the number of
cycles required to run the parallel version of the program and \( C_{SER} \) is the cycle count for the serial version of the program.

The equation to compute the Speedup After Parallel Conversion (SAP) is given below:

\[
SAP = \frac{C_{SER}}{C_{PAR}}
\]  

(4)

Where, \( Speedup \ After \ Parallel \ Conversion \ (SAP) \) is an estimate of how much faster the program will run, after parallel conversion, \( C_{PAR} \) is the number of cycles required to run the parallel version of the program and \( C_{SER} \) is the cycle count, for the serial version of the program.

4.7. Program Shape Flattening

As mentioned earlier, program syntax structures such as Functions, Loops and Conditionals, are a hindrance to effective estimation and predictions of performance. So as a first step, it is essential to flatten these high level language structures and then proceed with the estimation.

In the following paragraphs, we will give brief procedures in algorithmic form to perform these preparatory steps towards estimation.

**Algorithm 1. Program Shape Flattening**

1: procedure Flatten_Program
2:   procedure Inline_Function()
3:   procedure Unroll_Loop()
4:   procedure Predicate_Condition()
5: end procedure

6: procedure Inline_Function
7:   for Fnc 1 to n do // sweep through function calls in the program
8:     Get_Function_Definition(Def, Fnc) // fetch code block needed for the call
9:     Replace_Call_With_Definition(Def, Fnc) // replace call with the code block
10: end for
11: end procedure

12: procedure Unroll_Loop
13:   for Glp 1; n do // sweep through loops in the program
14:     Get_Loop_Block(Blk, Glp) // fetch code block for the loop
15:     Replace_Loop_With_Private_Blocks(Blk, Glp) // duplicate code block for each iteration
16: end for
17: end procedure

18: procedure Predicate_Condition
19:   for Cnd 1; n do // sweep through conditionals in the program
20:     Get_Condition_Block(Blk, Cnd) // fetch code block for the conditional
21:     Replace_Condition_With_Predicates(Blk, Cnd) // replace condition with the predicated block
22: end for
23: end procedure

4.8. Parallel Equivalence Classes

*Parallel Equivalence Classes* are a set of items that satisfy a single property. In the context of Parallel Conversions, it means sets of instructions that can be executed concurrently. However it should be noted that, instructions within a particular class, are to be executed in serial, to satisfy the property of an equivalence class. When the instructions of a program, are organized in to equivalence classes, the run time of the program, is reduced from the time spent, by all instructions of the program executing serially, to the run time of the longest running equivalence class.

What follows is the algorithm to create the Equivalence Classes, also referred to as Dependence Classes here. Once created, it becomes trivial to assess the run time and predict performance. The equivalence class creation algorithm is given below:

**Algorithm 2. Parallel Equivalence Classes Creation**

1: procedure Build Parallel Equivalence Classes
2:   Build_Equivalence_Classes()
3:   Merge_Equivalence_Classes()
4: end procedure
5: procedure Build_Equivalence_Classes
6:    for Ins 1; n do // sweep through the program’s instructions
7:       Get Result Operand(R, Ins) // fetch result operand of instruction
8:       Add Instruction(R, Ins) // add instruction to class R of global parallel equivalence class list
9:    end for
10: end procedure

11: procedure Merge Equivalence Classes
12:    for Ins 1; n do // sweep through the program’s instructions
13:       Get Result Operand(R, Ins) // fetch result operand of instruction
14:       Get Source1 Operand(S1, Ins) // fetch source1 operand of instruction
15:       Get Source2 Operand(S2, Ins) // fetch source2 operand of instruction
16:       Merge(R, S1) // merge class S1 to class R and update global parallel equiv. class list
17:       Merge(R, S2) // merge class S2 to class R and update global parallel equiv. class list
18:    end for
19: end procedure

4.9. Long Dependence Sequences

Certain programs exhibit long dependence sequences which can lead to loss of parallelism and produce fewer than optimal number of parallel classes. To prevent this, a heuristic based on the concept of Instruction Threshold (IT) is proposed, where IT is the number of instructions in a class which would ensure or force the class to become an independent parallel class. For instance IT which is a tuneable can be set to 32 instructions, which means that if the class size is less than IT proceed with the merger and in the other case skip merger. To implement this at the time of Parallel Class mergers a check is made to see if the class lengths meet the IT threshold. If the criterion is met then the instruction which acts as the key in both classes is hoisted out of the classes and a unique class is made with the instruction. Dependence is set from the new class with the hoisted instruction to the existing classes. New keys for the two existing classes are defined with the result operand from the least numbered instruction in both classes. This operation is recursively applied to both classes as long as the IT holds. These IT checks are enough to ensure optimum parallelization is preserved. While calculating parallel instruction count, care should be taken to add the serial paths which precede the parallel classes and add the instruction counts to the sum.

5. Analysis

To better understand the working of the internals of Caliper, we study a simple program with a function, loop and conditional to see how it gets transformed as it passes through the shape flattening exercises and finally analyzes the ASIF-IR program to generate the Caliper report.

5.1. Input File to Caliper (calfun.c)

Given below is a simple C program with a function, loop and condition. The program which is passed as input to Caliper is self-explanatory.

```
1: #include <stdio.h>
2: #define LOOP_COUNT 8
3: #define HALF_COUNT LOOP_COUNT/2
4: double
calcul(int x) {
5:     if (x < HALF_COUNT)
6:         return x * x;
7:     else
8:         return 2 * x;
9: }
10: }
11: int
main() {
12:     int i;
13:     double z = 0;
14:     for (i = 0; i < LOOP_COUNT; i++)
15:         z += calfun(i);
16:     printf("z = %lf\n", z);
```
5.2. Calfun.c after Function In-lining by Caliper (calfun_inl.c)

The first transformation applied to calfun.c is the function inlining and the program listed below is output as a result of that transformation. Lines 6-9 of the program represent the function which was inlined.

```
1: int
2: main()
3: {
4:   int i;
5:   double z = 0;
6:   for (i = 0; i < LOOP_COUNT; i++) {
7:     if (i < HALF_COUNT)
8:       z += i * i;
9:     else
10:       z += 2 * i;
11: }
```

5.3. Calfun_inl.c after Control Predication by Caliper (calfun_pred.c)

The program below is output by Caliper as a result of the Control Predication transformation where the If-conditional block is predicated as seen on line 6.

```
1: int
2: main()
3: {
4:   int i;
5:   double z = 0;
6:   for (i = 0; i < LOOP_COUNT; i++)
7:     z += (i < HALF_COUNT)? i * i : 2 * i;
8: }
```

5.4. Calfun_unl.c after Loop Unrolling by Caliper (calfun_unl.c)

The final transform applied by Caliper is the loop unrolling and the following program is output as seen on lines 5-20.

```
1: int
2: main()
3: {
4:   int i;
5:   double z = 0;
6:   /* iteration 0 */
7:   z += (0 < 4)? 0 * 0 : 2 * 0;
8:   /* iteration 1 */
9:   z += (1 < 4)? 1 * 1 : 2 * 1;
10:   /* iteration 2 */
11:   z += (2 < 4)? 2 * 2 : 2 * 2;
12:   /* iteration 3 */
13:   z += (3 < 4)? 3 * 3 : 2 * 3;
14:   /* iteration 4 */
15:   z += (4 < 4)? 4 * 4 : 2 * 4;
16:   /* iteration 5 */
17:   z += (5 < 4)? 5 * 5 : 2 * 5;
18:   /* iteration 6 */
19:   z += (6 < 4)? 6 * 6 : 2 * 6;
20:   /* iteration 7 */
21:   printf("z = %lf\n", z);
22: }
```

5.4. Calfun_unl.c after ASIF-IR generation by Caliper (calfun.s)

The following ASIF-IR is the resulting program after all transformations and high level code are translated to IR. Lines 5-25 show the results. To save space only iterations 0, 1 and 7 are shown with the others snipped.
# 5.5. CALIPER Parallel Estimates (calfun.csv)

After the ASIF-IR code is passed to Caliper it creates the required Equivalence Classes and calculates the MAP and SAP metrics, and the output is generated in the form of CSV file as shown below:

(1), Serial Instruction Count, SIN, 58  
(2), Equivalence Class Count, EQC, 9  
(3), Mean Instruction Count, MIN, 6.44  
(4), Parallel Instruction Count, PIN, 33  
(5), Serial Execution Cycles, SEC, 58  
(6), Parallel Execution Cycles, PEC, 33  
(7), Maximum Available Parallelism, MAP, 43.10  
(8), Speedup After Parallelization, SAP, 1.75

For the given program, Serial Execution Cycles was 58 same as the instruction count and Parallel Execution Cycles was 33. From the Maximum Available Parallelism (MAP) value it is evident that 43.10% of the given program is parallelizable and the Speedup After Parallelization (SAP) is about 1.75.

# 6. Competitive Analysis

Here we compare Asterix/Caliper with other leading compilers both open-sourced and proprietary. While LLVM, GCC and Open64 are open source technologies, ICC, PGI and PathScale offer proprietary products.
As seen from the table, Caliper provides parallel performance estimates which none of the other state-of-the-art compilers provide. However all of them provide optimization related diagnostics at some basic level. Based on our findings, we have to conclude that Caliper is the only working, Parallel Performance Estimation and Prediction Solution available, at this time.

### Table 1. Performance Estimation Support [3]

<table>
<thead>
<tr>
<th>No.</th>
<th>Compiler</th>
<th>Performance Estimation Availability</th>
<th>Optimization and Parallel Diagnostics</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Asterix/Caliper [1]</td>
<td>YES</td>
<td>CALIPER/3PO</td>
<td>Inherent parallelism (MAP) and Expected speedup (SAP) metrics are generated</td>
</tr>
<tr>
<td>2</td>
<td>GCC³</td>
<td>NO</td>
<td>Several –fdump flags such as –fdump-ipv-all and –dump-ipv-inline</td>
<td>Information on in-lined functions etc</td>
</tr>
<tr>
<td>3</td>
<td>CLANG/LLVM²</td>
<td>NO</td>
<td>–emit-llvm and –Rpass, -Rpass-missed and -Rpass-analysis</td>
<td>Instrumented IR and optimization reports</td>
</tr>
<tr>
<td>4</td>
<td>Open64⁵</td>
<td>NO</td>
<td>-CLIST and –FLIST, -LNO:refetch_verbose, -LNO:simd_verbos etc</td>
<td>Prefetch and other optimization specific diagnostics</td>
</tr>
<tr>
<td>5</td>
<td>Intel/ICC⁴</td>
<td>NO</td>
<td>-fverbose-asm and opt-report</td>
<td>Generate all optimization related activity as a report</td>
</tr>
<tr>
<td>6</td>
<td>PGI⁶</td>
<td>NO</td>
<td>-Minfo and –Mneginfo flags provide diagnostics</td>
<td>Informative messages such as, whether a loop was vectorized or not and rationale</td>
</tr>
<tr>
<td>7</td>
<td>Pathscale⁶</td>
<td>NO</td>
<td>-CLIST and -FLIST, options are provided for diagnostics</td>
<td>Information on a specific optimization such as Prefetches</td>
</tr>
</tbody>
</table>

* Estimation capabilities of Modern Compiler Frameworks.

### 7. Conclusion

Caliper was developed to aid the parallel programmer in his endeavours, by providing a yield estimate resulting from parallel conversion of a given program. Caliper works on programs in ASIF-IR format an internal representation developed as part of our compiler framework. Caliper as a preliminary step performs Program Shape Flattening Transformations to ease subsequent steps. It performs symbolic analysis of ASIF-IR instructions representing the given program internally, and classifies them in to Equivalence Classes based on their dependence behaviour. These classes which host dependent instructions are themselves dependence free and are eligible to operate in interleaved fashion with other classes. Once arranged in this fashion it becomes easy to compute Serial and Parallel runtimes. Serial runtime is the sequential runtime of the instructions making up the program and Parallel runtime is the runtime of the class that runs the longest. Based on these two numbers two metrics useful to the programmer are reported. Maximum Available Parallelism (MAP) points out the inherent parallel potential of a given program. Speedup after Parallelization (SAP) complements the earlier metric by reporting the estimated speedup resulting from parallel conversion. At the time of writing there are no known technologies comparable to Caliper and we conclude that Caliper is a one of its kind parallelization technology.

### References


---


© 2021 by the author(s). Published by Annals of Emerging Technologies in Computing (AETiC), under the terms and conditions of the Creative Commons Attribution (CC BY) license which can be accessed at http://creativecommons.org/licenses/by/4.0.