Research Article

Inherent Parallelism and Speedup Estimation of Sequential Programs

Sesha Kalyur* **and Nagaraja G.S**

[1]CSE Dept. R.V. College of Engineering, Bangalore, India
SeshaKalyur@Outlook.Com; nagarajags@rvce.edu.in
*Correspondence: SeshaKalyur@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

Sesha Kalyur and Nagaraja G.S, "Inherent Parallelism and Speedup Estimation of Sequential Programs", *Annals of Emerging Technologies in Computing (AETiC)*, Print ISSN: 2516-0281, Online ISSN: 2516-029X, pp. 62-77, Vol. 5, No. 2, 1st April 2021, Published by International Association of Educators and Researchers (IAER), DOI: 10.33166/AETiC.2021.02.006, Available: http://aetic.theiaer.org/archive/v5/v5n2/p6.html.

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].

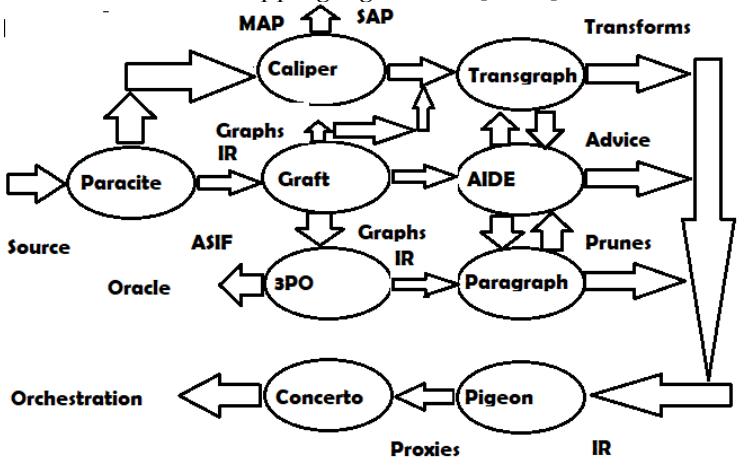

Figure 1. Phases of ASTERIX Compiler Chain[1].

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.

4. CALIPER

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.

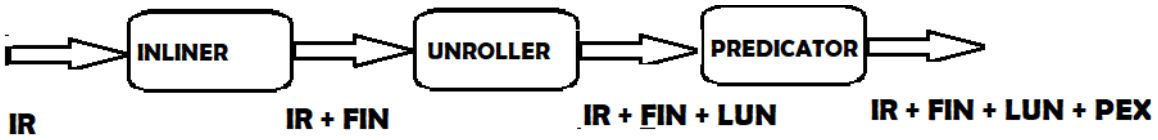

Figure 2. CALIPER operation[2]

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:*

$$C\_{SER} = N\_{INC} \quad (1)$$

Here, C_{CYC} is the count of cycles, to run the serial version of the program, and N_{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,

$$C\_{PAR} = MAX(EQC\_{1}, EQC\_{2}, ..., EQC\_{n}) \quad (2)$$

Where C_{PAR} is the parallel cycle count, EQC_{1}, EQC_{2},..., 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:

$$MAP = (C\_{SER} - C\_{PAR}) / C\_{SER}) \times 100 \quad (3)$$

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_{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:

$$SAP = (C\_SER / C\_PAR) \quad (4)$$

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:    Inline_Function()
3:    Unroll_Loop()
4:    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
5: calfun(int x) {
6:    if (x < HALF_COUNT)
7:       return x * x;
8:    else
9:       return 2 * x;
10: }
11: int
12: main() {
13:    int i;
14:   double z = 0;
15:   for (i = 0; i < LOOP_COUNT; i++)
16:      z += calfun(i);
17:   printf("z = %lf\n", z);
```

```
18: }
```

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

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

The program below if 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:    int i;
4:    double z = 0;
5:    for (i = 0; i < LOOP_COUNT; i++)
6:       z += (i < HALF_COUNT)? i * i : 2 * i;
7:     printf("z = %lf\n", z);
8: }
```

5.4. Calfun_pre.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:    int i;
4:    double z = 0;
5:    /* iteration 0 */
6:    z += (0 < 4)? 0 * 0 : 2 * 0;
7:    /* iteration 1 */
8:    z += (1 < 4)? 1 * 1 : 2 * 1;
9:    /* iteration 2 */
10:   z += (2 < 4)? 2 * 2 : 2 * 2;
11:   /* iteration 3 */
12:   z += (3 < 4)? 3 * 3 : 2 * 3;
13:   /* iteration 4 */
14:   z += (4 < 4)? 4 * 4 : 2 * 4;
15:   /* iteration 5 */
16:   z += (5 < 4)? 5 * 5 : 2 * 5;
17:   /* iteration 6 */
18:   z += (6 < 4)? 6 * 6 : 2 * 6;
19:   /* iteration 7 */
20:   z += (7 < 4)? 7 * 7 : 2 * 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.

```
1: main:
2:    ;DEC i, 4
3:    ;DEC z, 8
4:    ; iteration 0
5:    ; z += (0 < 4)? 0 * 0 : 2 * 0;
6:    MUL T_0, 0, 0
7:    MUL T_1, 2, 0
8:    LTH T_3, 0, 4
9:    ADE T_4, z, T_0
10:   ADE T_5, z, T_1
11:   CMOV z, T_3, T_4
12:   CMOV z, T_3, T_5
13:   ; iteration 1
14:   ;z += (1 < 4)? 1 * 1 : 2 * 1;
15:   MUL T_6, 1, 1
16:   MUL T_7, 2, 1
17:   LTH T_8, 1, 4
18:   ADE T_9, z, T_6
19:   ADE T_10, z, T_7
20:   CMOV z, T_8, T_9
21:   CMOV z, T_8, T_10
22:   ; iteration 2 – removed to save space
23:   ; iteration 3 – removed to save space
24:   ; iteration 4 – removed to save space
25:   ; iteration 5 – removed to save space
26:   ; iteration 6 – removed to save space
27:   ; iteration 7
28:   ;z += (7 < 4)? 7 * 7 : 2 * 7;
29:   MUL T_36, 7, 7
30:   MUL T_37, 2, 7
31:   LTH T_38, 7, 4
32:   ADE T_39, z, T_36
33:   ADE T_40, z, T_37
34:   CMOV z, T_38, T_39
35:   CMOV z, T_38, T_40
36:   ;printf("z = %lf\n", z);
```

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]

No.	Compiler	Performance Estimation Availability	Optimization and Parallel Diagnostics Command line flags or Commands	Outcome
1	Asterix/Caliper [1]	YES	CALIPER/3PO	Inherent parallelism (MAP) and Expected speedup (SAP) metrics are generated
2	GCC[1]	NO	Several –fdump flags such as –fdump-ipa-all and –dump-ipa-inline	Information on in-lined functions etc
3	CLANG/LLVM [2]	NO	-emit-llvm and –Rpass, -Rpass-missed and –Rpass-analysis	Instrumented IR and optimization reports
4	Open64 [3]	NO	-CLIST and –FLIST, -LNO:refetch_verbose, -LNO:simd_verbose etc	Prefetch and other optimization specific diagnostics
5	Intel/ICC [4]	NO	-fverbose-asm and opt-report	Generate all optimization related activity as a report
6	PGI [5]	NO	-Minfo and –Mneginfo flags provide diagnostics	Informative messages such as, whether a loop was vectorized or not and rationale
7	Pathscale [6]	NO	-CLIST and -FLIST, options are provided for diagnostics	Information on a specific optimization such as Prefetches

* 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

[1] Sesha Kalyur and GS Nagaraja, "CALIPER: A coarse grain parallel performance estimator and predictor", In *International Conference for Emerging Technologies in Computing*, Print ISBN 978-3-030-60035-8, Online

[1]GNU-compiler-collection. Available: https://gcc.gnu.org/onlinedocs/ (Accessed on Day 28/01/2021).
[2]LLVM-compiler-collection. Available: https://clang.llvm.org/docs/UsersManual.html (Accessed on 28/01/2021)
[3]Open64-compiler-collection. Available: https://developer.amd.com/x86-open64-compiler-suite/ (Accessed on 28/01/2021)
[4]Intel-ICC-compiler-collection. Available: https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top.html (Accessed on 28/01/2021)
[5]PGI-compiler-collection. Available: https://www.pgroup.com/resources/docs/19.10/x86/pgi-ref-guide/index.htm (Accessed on 28/01/2021)
[6]PathScale-compiler-collection. Available: http://www.scc.kit.edu/scc/docs/HP-XC/pathscale/UserGuide.pdf (Accessed on 28/01/2021)

ISBN 978-3-030-60036-5, pp. 16-39, Springer, 2020, DOI: 10.1007/978-3-030-60036-5_2, Available: https://link.springer.com/chapter/10.1007/978-3-030-60036-5_2.

[2] David Culler, Jaswinder Pal Singh and Anoop Gupta, Parallel computer architecture: a hardware/software approach, California, USA: Morgan Kaufmann Publishers, Inc., 1999.

[3] William Gropp, William D Gropp, Ewing Lusk, Anthony Skjellum and Ewing Lusk, Using MPI: portable parallel programming with the message-passing interface, 2nd ed. Cambridge, UK: MIT press, 1999.

[4] Ran Canetti, L. Paul Fertig, Saul A. Kravitz, Dalia Malki, Ron Y. Pinter, Sara Porat and Avi Teperman, "The parallel c (pc) programming language", *IBM Journal of Research and Development*, pp. 727-741, Vol. 35, No. 5.6, September 1991, DOI: 10.1147/rd.355.0727.

[5] Alan Kaminsky, "Parallel java: A united api for shared memory and cluster parallel programming in 100% java", In *2007 IEEE International Parallel and Distributed Processing Symposium*, Print ISSN: 1530-2075, pp. 1-8, March 2007, DOI: 10.1109/IPDPS.2007.370421.

[6] Eduard Ayguade, Nawal Copty, Alejandro Duran, Jay Hoeinger, Yuan Lin *et al.*, "The design of openmp tasks", *IEEE Transactions on Parallel and Distributed Systems*, Print ISSN: 1045-9219, Electronic ISSN: 1558-2183, pp. 404-418, Vol. 20, No. 3, March 2009, DOI: 10.1109/TPDS.2008.105, Available: https://ieeexplore.ieee.org/abstract/document/4553700.

[7] Leonardo Dagum and Ramesh Menon, "Openmp: an industry standard api for shared-memory programming", *IEEE computational science and engineering*, Print ISSN: 1070-9924, Electronic ISSN: 1558-190X, pp.46-55, 1998, Vol. 5, No. 1, DOI: 10.1109/99.660313, Available: https://ieeexplore.ieee.org/abstract/document/660313.

[8] Hironori Kasahara, Motoki Obata and Kazuhisa Ishizaka, "Automatic coarse grain task parallel processing on smp using openmp", In *International Workshop on Languages and Compilers for Parallel Computing*, Print ISBN 978-3-540-42862-6, Online ISBN 978-3-540-45574-5, pp. 189-207, Springer, 2000, DOI:10.1007/3-540-45574-4_13, Available: https://link.springer.com/chapter/10.1007/3-540-45574-4_13.

[9] Mitsuhisa Sato, "Openmp: parallel programming API for shared memory multiprocessors and on-chip multiprocessors", In *Proceedings of the 15th international symposium on System Synthesis*, pp. 109-111, 2002, DOI: 10.1145/581199.581224, Available: https://dl.acm.org/doi/10.1145/581199.581224.

[10] Seyong Lee, Seung-Jai Min and Rudolf Eigenmann, "Openmp to gpgpu: a compiler framework for automatic translation and optimization", *ACM Sigplan Notices*, pp.101-110, Vol. 44, No. 4, 2009, DOI: 10.1145/1594835.1504194, Available: https://dl.acm.org/doi/abs/10.1145/1594835.1504194.

[11] Sesha Kalyur and G. S. Nagaraja, "A survey of modeling techniques used in compiler design and implementation", In *International Conference on Computation System and Information Technology for Sustainable Solutions* (CSITSS), pp. 355-358, October 2016, DOI: 10.1109/CSITSS.2016.7779385, Available: https://ieeexplore.ieee.org/abstract/document/7779385.

[12] Uday Bondhugula, Muthu Baskaran, Albert Hartono, Sriram Krishnamoorthy, J. Ramanujam *et al.*, "Towards effective automatic parallelization for multicore systems", In *Parallel and Distributed Processing, 3 2008, IPDPS 2008, IEEE International Symposium* on, Print ISBN:978-1-4244-1693-6, Print ISSN: 1530-2075, pp. 1-5, April 2008, DOI: 10.1109/IPDPS.2008.4536401.

[13] Paul Lokuciejewski, Daniel Cordes, Heiko Falk and Peter Marwedel, "A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models", In *Code Generation and Optimization, 2009, CGO 2009, International Symposium* on, pp. 136-146, March 2009, DOI: 10.1109/CGO.2009.17, Available: https://dl.acm.org/doi/10.1109/CGO.2009.17.

[14] Susan Horwitz and Thomas Reps, "The use of program dependence graphs in software engineering", In *Proceedings of the 14th international conference on Software engineering*, pages 392-411, 1992, DOI: 10.1145/143062.143156, Available: https://dl.acm.org/doi/abs/10.1145/143062.143156.

[15] Vivek Sarkar, "Automatic partitioning of a program dependence graph into parallel tasks", *IBM Journal of Research and Development*, pp. 779-804, Vol. 35, No. 5.6, 1991, DOI: 10.1147/rd.355.0779, Available: https://ieeexplore.ieee.org/abstract/document/5389740.

[16] Angeles Navarro, Emilio Zapata and David Padua, "Compiler techniques for the distribution of data and computation", In *Parallel and Distributed Systems, IEEE Transactions on*, pp. 545-562, Vol. 14, No. 6, June 2003, DOI: 10.1109/TPDS.2003.1206503, https://ieeexplore.ieee.org/abstract/document/1206503.

[17] Roxana E. Diaconescu, Lei Wang, Zachary Mouri and Matt Chu, "A compiler and runtime infrastructure for automatic program distribution", In *Parallel and Distributed Processing Symposium, 2005, Proceedings. 19th IEEE International*, ISBN:0-7695-2312-9, Print ISSN: 1530-2075, pp. 52a-52a, April 2005, DOI: 10.1109/IPDPS.2005.7, https://ieeexplore.ieee.org/abstract/document/1419872.

[18] Thomas Fahringer, "Using the p3t to guide the parallelization and optimization report under the vienna fortran compilation system", In *Scalable High Performance Computing Conference, 1994, Proceedings of the*, Print ISBN: 0-8186-5680-8, pp. 437-444, May 1994, DOI: 10.1109/SHPCC.1994.296676, Available: https://doi.org/10.1109/SHPCC.1994.296676.

[19] Thomas Fahringer, "On estimating the useful work distribution of parallel programs under p3t: A static performance estimator", In *Concurrency: Practice and Experience, Wiley Online Library*, ,pp. 261-282, Vol. 8, No. 4, May 1996, DOI: 10.1002/(SICI)1096-9128(199605)8:4<261::AID-CPE205>3.0.CO;2-6, Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.3470&rep=rep1&type=pdf.

[20] Dan Quinlan, "Rose: Compiler support for object-oriented frameworks", *Parallel Processing Letters*, pp.215-226, Vol. 10, No. 02n03, 2000, DOI: 10.1142/S0129626400000214, Available: https://doi.org/10.1142/S0129626400000214.

[21] Vikram Adve, Guohua Jin, John Mellor-Crummey and Qing Yi, "Design and evaluation of a computation partitioning framework for data-parallel compilers", *Technical report, Department of Computer Science, Rice University*, Tech Rep: CS-TR01-382, 2001.

[22] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann *et al.*, "Cetus: A source-to-source compiler infrastructure for multicores", *IEEE Computer*, Print ISSN: 0018-9162, Electronic ISSN: 1558-0814, pp. 36-42, Vol. 42, No. 12,2009, DOI: 10.1109/MC.2009.385, Available: https://ieeexplore.ieee.org/abstract/document/5353460.

[23] Robert L Bocchino Jr, Vikram S Adve, Sarita V Adve and Marc Snir, "Parallel programming must be deterministic by default", In Proceedings *of the First USENIX conference on Hot topics in parallelism*, pp. 4-4, April 2009.

[24] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado *et al.*, "Polyhedral parallel code generation for cuda", *ACM Transactions on Architecture and Code Optimization (TACO)*, pp. 1-23, Vol. 9, No. 4, 2013, DOI: 10.1145/2400682.2400713, Available: https://dl.acm.org/doi/abs/10.1145/2400682.2400713.

[25] Tim A. Wagner, Vance Maverick, Susan L. Graham and Michael A. Harrison, "Accurate static estimators for program optimization", *SIGPLAN Notices*, pp. 85-96, Vol. 29, No. 6, June 1994, DOI: 10.1145/178243.178251, Available: https://dl.acm.org/doi/abs/10.1145/178243.178251.

[26] Vasanth-Balasundaram, Georey Fox, Ken Kennedy and Ulrich Kremer, "A static performance estimator to guide data partitioning decisions", *SIGPLAN Notices*, pp. 213-223, Vol. 26, No. 7, April 1991, DOI: 10.1145/109625.109647, Available: https://dl.acm.org/doi/abs/10.1145/109625.109647.

[27] Thomas Fahringer and Hans P. Zima, "A static parameter based performance prediction tool for parallel programs", In *Proceedings of the 7th International Conference on Supercomputing, ICS '93*, pp. 207-219, New York, NY, USA, 1993, ACM, DOI: 10.1145/165939.165971.

[28] D. Arapattu and Dennis Gannon, "Building analytical models into an interactive performance prediction tool", In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, Supercomputing '89*, Print ISBN:0-89791-341-8, pp. 521-530, New York, NY, USA, 1989, ACM, DOI: 10.1145/76263.76321, Available: https://ieeexplore.ieee.org/abstract/document/5348984.

[29] Christophe Dubach, John Cavazos, Bjorn Franke, Grigori Fursin, Michael F.P. O'Boyle *et al.*, "Fast compiler optimisation evaluation using code-feature based performance prediction", In *Proceedings of the 4th International Conference on Computing Frontiers, CF '07*, pp. 131-142, New York, NY, USA, 2007, ACM, DOI: 10.1145/1242531.1242553, Available: https://dl.acm.org/doi/abs/10.1145/1242531.1242553.

[30] Donghwan Jeon, Saturnino Garcia, Chris Louie and Michael Bedford Taylor, "Kismet: Parallel speedup estimates for serial programs", *SIGPLAN Notices*, pp. 519-536, Vol. 46, No. 10, October 2011, DOI: 10.1145/2048066.2048108, Available: https://dl.acm.org/doi/abs/10.1145/2048066.2048108.

[31] Nathan R. Tallent and John M. Mellor-Crummey, "Effective performance measurement and analysis of multithreaded applications", In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '09*, pp. 229-240, New York, NY, USA, 2009, ACM, DOI: 10.1145/1504176.1504210, Available: https://dl.acm.org/doi/abs/10.1145/1504176.1504210.

[32] Aparna Kotha, Kapil Anand, Matthew Smithson, Greeshma Yellareddy and Rajeev Barua, "Automatic parallelization in a binary rewriter", In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Electronic ISSN: 2379-3155, Print ISSN: 1072-4451, pp. 547-557, December 2010, DOI: 10.1109/MICRO.2010.27, Available: https://ieeexplore.ieee.org/abstract/document/5695565.

[33] William Blume and Rudolf Eigenmann, "An overview of symbolic analysis techniques needed for the effective parallelization of the perfect benchmarks", In *Proceedings of the 1994 International Conference on Parallel Processing*, Print ISBN:0-8493-2493-9, pp. 233-238, Vol. 02, ICPP '94, Washington, DC, USA, 1994, DOI: 10.1109/ICPP.1994.59, Available: https://ieeexplore.ieee.org/abstract/document/5727792.

[34] Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jaejin Lee *et al.*, "Restructuring programs for high-speed computers with Polaris", In *International Conference on Parallel Processing, 1996, Proceedings of the 1996 ICPP Workshop on Challenges for*, Print ISBN: 0-8186-7623-X, Print ISSN: 1530-2016, pp. 149-161, August 1996, DOI: 10.1109/ICPPW.1996.538601.

[35] Mihai T. Lazarescu and Luciano Lavagno, "Dynamic trace-based data dependency analysis for parallelization of c programs", In *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th*

International Working Conference on, Electronic ISBN:978-0-7695-4783-1, Print ISBN:978-1-4673-2398-7, pp. 126-131, September 2012, DOI: 10.1109/SCAM.2012.15.

[36] Clemens Hammacher, Kevin Streit, Sebastian Hack and Andreas Zeller, "Profling java programs for parallelism", In *Multicore Software Engineering, 2009, IWMSE '09, ICSE Workshop on*, Print ISBN:978-1-4244-3718-4, pp 49-55, May 2009, DOI: 10.1109/IWMSE.2009.5071383, Available: https://ieeexplore.ieee.org/abstract/document/5071383.

[37] Saturnino Garcia, Donghwan Jeon, Christopher Louie and Michael B. Taylor, "The kremlin oracle for sequential code parallelization", *IEEE Micro*, Print ISSN: 0272-1732, Electronic ISSN: 1937-4143, pp.42-53, Vol. 32, No. 4, July 2012, DOI: 10.1109/MM.2012.52, Available: https://ieeexplore.ieee.org/abstract/document/6235946.

[38] Lucian Codrescu and D. Scott Wills, "On dynamic speculative thread partitioning and the mem-slicing algorithm", In *Parallel Architectures and Compilation Techniques, 1999, Proceedings. 1999 International Conference on*, Print ISBN:0-7695-0425-6, Print ISSN: 1089-795X, pp. 40-46, 1999, DOI: 10.1109/PACT.1999.807404, Available: https://ieeexplore.ieee.org/abstract/document/807404.

[39] Borys J. Bradel and Tarek S. Abdelrahman, "Automatic trace-based parallelization of java programs", In *2007 International Conference on Parallel Processing (ICPP 2007)*, Print ISSN: 0190-3918, Electronic ISSN: 2332-5690, pp. 26-26, September 2007, DOI: 10.1109/ICPP.2007.21, Available: https://ieeexplore.ieee.org/abstract/document/4343833.

[40] Calin Cascaval, Luiz De Rose, David A. Padua and Daniel A. Reed, "Compile-time based performance prediction", In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing, LCPC '99*, Print ISBN: 978-3-540-67858-8, Online ISBN: 978-3-540-44905-8, pp. 365-379, Berlin, Heidelberg, 2000, Springer-Verlag, DOI: 10.1109/ICPP.2007.21, Available: Available: https://link.springer.com/chapter/10.1007/3-540-44905-1_23.

[41] Pedro C. Diniz, "A compiler approach to performance prediction using empirical-based modeling", In Proceedings *of the 2003 International Conference on Computational Science: PartIII, ICCS'03*, Print ISBN: 978-3-540-40196-4, Online ISBN: 978-3-540-44863-1, pp. 916-925, Berlin, Heidelberg, 2003, Springer-Verlag, DOI: 10.1007/3-540-44863-2_90, Available: https://link.springer.com/chapter/10.1007/3-540-44863-2_90.

[42] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk, "Prospector: A dynamic data-dependence profiler to help parallel programming", *2010 Usenix*, January 2010, Available: https://static.usenix.org/events/hotpar10/final_posters/Kim.pdf.

[43] Barton P. Miller, Mark D. Callaghan, Jonathon M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin *et al.*, "The paradyn parallel performance measurement tool", *Computer*, Print ISSN: 0018-9162, Electronic ISSN: 1558-0814, pp. 37-46, Vol. 28, No. 11, November 1995, DOI: 10.1109/2.471178, Available: https://ieeexplore.ieee.org/abstract/document/471178.

[44] Luiz A. de Rose and Daniel A. Reed, "Svpablo: A multi-language architecture-independent performance analysis system", In Proceedings of the 1999 International Conference on Parallel Processing, ICPP '99, Print ISBN: 0-7695-0350-0, Print ISSN: 0190-3918, pp. 311-, Washington, DC, USA, 1999, IEEE Computer Society, 6, DOI: 10.1109/ICPP.1999.797417.

[45] J. Zhai, W. Chen, W. Zheng and K. Li, "Performance prediction for largescale parallel applications using representative replay", *IEEE Transactions on Computers*, Print ISSN: 0018-9340, Electronic ISSN: 1557-9956, pp. 2184-2198, Vol. 65, No. 7, July 2016, DOI: 10.1109/TC.2015.2479630, Available: https://ieeexplore.ieee.org/abstract/document/7271042.

[46] Nicholas Nethercote and Julian Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation", *SIGPLAN Notices*, pp. 89-100, Vol. 42, No. 6, June 2007, DOI: 10.1145/1273442.1250746, Available: https://dl.acm.org/doi/abs/10.1145/1273442.1250746.

[47] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation", *SIGPLAN Notices*, pp. 190-200, Vol. 40, No. 6, June 2005, DOI: 10.1145/1064978.1065034, Available: https://dl.acm.org/doi/abs/10.1145/1064978.1065034.

[48] Ko-Yang Wang, "Precise compile-time performance prediction for superscalar-based computers", In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI '94*, pp. 73-84, New York, NY, USA, 1994, ACM, DOI: 10.1145/178243.178250, Available: https://dl.acm.org/doi/abs/10.1145/178243.178250.

[49] Gabriel Marin and John Mellor-Crummey, "Cross-architecture performance predictions for scientific applications using parameterized models", In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '04/Performance '04*, pp. 2-13, New York, NY, USA, 2004, ACM, DOI: 10.1145/1005686.1005691.

[50] Zhonglei Wang, Antonio Sanchez and Andreas Herkersdorf, "Scisim: A software performance estimation framework using source code instrumentation", In *Proceedings of the 7th International Workshop on Software*

and Performance, WOSP '08, pp. 33-42, New York, NY, USA, 2008, ACM, DOI: 10.1145/1383559.1383565, Available: https://dl.acm.org/doi/abs/10.1145/1383559.1383565.

[51] Ko-Yang Wang, "A performance predication model for parallel compilers", *Department of Computer Science Technical Reports, Perdue University*, 1990.

[52] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser *et al.*, "Logp Towards a realistic model of parallel computation", In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 1-12, 1993, DOI: 10.1145/155332.155333, Available: https://dl.acm.org/doi/abs/10.1145/155332.155333.

[53] Thomas Fahringer, "Evaluation of benchmark performance estimation for parallel fortran programs on massively parallel simd and mimd computers", In *2nd Euromicro Workshop on Parallel and Distributed Processing*, pp. 449-456. July 1994.

[54] Kattamuri Ekanadham, Vijay K Naik and Mark S Squillante, "Pet: A parallel performance estimation tool" In *Proceedings of the 7th SIAM conference for Parallel Processing for Scientific Computing (PPSC)*, pp. 826-831. Citeseer, 1995.

[55] Tony Hey, Alistair Dunlop and Emilio Hernandez, "Realistic parallel performance estimation", *Parallel Computing*, pp. 5-21, Vol. 23, No. 1-2, 1997, DOI: 10.1016/S0167-8191(96)00093-2, Available: https://www.sciencedirect.com/science/article/abs/pii/S0167819196000932.

[56] Lei Hu and Ian Gorton, "Performance evaluation for parallel systems: A survey", *University of New South Wales, School of Computer Science and Engineering Reports*, 1997, Available: https://cgi.cse.unsw.edu.au/~reports/papers/9707.pdf

[57] Vikram S Adve and Mary K Vernon,"A deterministic model for parallel program performance evaluation" *ACM Transactions on Computer Systems, 1998*, Available: https://scholarship.rice.edu/handle/1911/96503.

[58] Narasimhan Sreraman and Ramaswamy Govindarajan, "A vectorizing compiler for multimedia extensions", *International Journal of Parallel Programming*, pp. 363-400, Vol. 28, No. 4, 2000, DOI: 10.1023/A:1007559022013, Available: https://link.springer.com/article/10.1023/A:1007559022013.

[59] Ilya Sharapov, Robert Kroeger, Guy Delamarter, Razvan Cheveresan and Matthew Ramsay, "A case study in top-down performance estimation for a large-scale parallel application", In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 81-89, 2006, DOI: 10.1145/1122971.1122985, Available: https://dl.acm.org/doi/abs/10.1145/1122971.1122985.

[60] Henry Kasim, Verdi March, Rita Zhang and Simon See, "Survey on parallel programming model", In *IFIP International Conference on Network and Parallel Computing*, Print ISBN: 978-3-540-88139-1, Online ISBN: 978-3-540-88140-7, pp. 266-275. Springer, 2008, DOI: 10.1007/978-3-540-88140-7_24, Available: https://link.springer.com/chapter/10.1007/978-3-540-88140-7_24.

[61] Donghwan Jeon, "Parallel speedup estimates for serial programs", *PhD thesis, UC San Diego*, 2012, Available: https://escholarship.org/uc/item/66h1d17x

[62] Dustin Feld, Thomas Soddemann, Michael Junger and Sven Mallach, "Hardware-aware automatic code-transformation to support compilers in exploiting the multi-level parallel potential of modern cpus", In *Proceedings of the 2015 International Workshop on Code Optimization for Multi and Many Cores*, pp. 1-10, 2015, DOI: 10.1145/2723772.2723776, Available: https://dl.acm.org/doi/abs/10.1145/2723772.2723776.

[63] Xiaowen Chen, Zhonghai Lu, Axel Jantsch, Shuming Chen, Yang Guo *et al.*, "Performance analysis of homogeneous on-chip large-scale parallel computing architectures for data-parallel applications", *Journal of Electrical and Computer Engineering*, 2015, DOI: 10.1155/2015/902591, Available: https://www.hindawi.com/journals/jece/2015/902591/.

[64] Kumar Vipin P and Gupta Anshul, "Analyzing Scalability of Parallel Algorithms and Architectures", *Journal of Parallel and Distributed Computing*, Vol. 22, No. 3, Pages 379-391, September 1994, DOI: 10.1006/jpdc.1994.1099.

[65] Sesha Kalyur and G. S. Nagaraja, "Paracite: Auto-parallelization of a sequential program using the program dependence graph" In *2016 International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS)*, Electronic ISBN:978-1-5090-1022-6, Print ISBN:978-1-5090-1020-2, pp. 7-12, October 2016, DOI: 10.1109/CSITSS.2016.7779431, Available: https://ieeexplore.ieee.org/abstract/document/7779431.

[66] Sesha Kalyur and G. S. Nagaraja, "A taxonomy of methods and models used in program transformation and parallelization", In Navin Kumar and R. Venkatesha Prasad, editors, *Ubiquitous Communications and Network Computing*, Print ISBN: 978-3-030-20614-7, Online ISBN: 978-3-030-20615-4, pp. 233-249, Cham, 2019, Springer International Publishing, DOI: 10.1007/978-3-030-20615-4_18, Available: https://link.springer.com/chapter/10.1007/978-3-030-20615-4_18.

[67] Sesha Kalyur and G. S. Nagaraja, "Aide: An interactive environment for program transformation and parallelization", In *2017 2nd International Conference on Computational Systems and Information Technology for*

Sustainable Solution (CSITSS), Electronic ISBN: 978-1-5386-2044-1, ISBN: 978-1-5386-2045-8, pp. 199-203, Dec 2017, DOI: 10.1109/CSITSS.2017.8447848.

[68] Sesha Kalyur and G. S. Nagaraja, "Concerto: A program parallelization, orchestration and distribution infrastructure", In *2017 2nd International Conference on Computational Systems and Information Technology for Sustainable Solution (CSITSS)*, Electronic ISBN: 978-1-5386-2044-1, ISBN: 978-1-5386-2045-8, pp. 204-209, Dec 2017, DOI: 10.1109/CSITSS.2017.8447691.

[69] Sesha Kalyur and G. S. Nagaraja, "Efficient graph algorithms for mapping tasks to processors", In Navin Kumar and R. Venkatesha Prasad, editors, *Ubiquitous Communications and Network Computing*, Print ISBN: 978-3-030-47559-8, Online ISBN: 978-3-030-47560-4, pp.467-491, Cham, 2020, Springer International Publishing, DOI: 10.1007/978-3-030-47560-4_35, Available: https://link.springer.com/chapter/10.1007/978-3-030-47560-4_35.

[70] Sesha Kalyur and G. S. Nagaraja, "Evaluation of graph algorithms for mapping tasks to processors", In Navin Kumar and R. Venkatesha Prasad, editors, *Ubiquitous Communications and Network Computing*, Print ISBN: 978-3-030-47559-8, Online ISBN: 978-3-030-47560-4, pp. 423-448, Cham, 2020, Springer International Publishing, DOI: 10.1007/978-3-030-47560-4_33, Available: https://link.springer.com/chapter/10.1007/978-3-030-47560-4_33.