CALIPER: A Coarse Grain Parallel Performance Estimator and Predictor

Sesha Kalyur[✉] and G.S Nagaraja

Department of Computer Science and Engineering, R. V. College of Engineering, VTU, Bangalore, India
Sesha.Kalyur@Gmail.Com, nagarajags@rvce.edu.in

Abstract. Empirical studies of Program Performance, are limited by the choice and the resulting bias, from the input samples used in the experiment. Estimation and Prediction based on static analysis, are more universal, superior and widely accepted. However the higher language artifacts such as Procedures, Loops, Conditionals and Recursion which ease program development can be an hindrance to quality analysis and performance study, both in terms of time and effort spent and in some extreme cases making it impractical. However, we could transform the program, eliminate the constraints imposed by these program structures and greatly ease the process of quality analysis and performance study. This process may also reduce the errors in the estimation, and help deliver timely results, when there is still an opportunity to use them in a later analysis phase. We propose transformations prior to estimation, such as *Procedure Call Expansion*, *Loop Unrolling* and *Control Predication* collectively referred to as *Program Shape Flattening* here with the structural hindrances themselves referred to as the *Program Shape*. The outcome of this transformation, is sequential code that is easy to work with. Specifically, for parallel performance estimations, we now have code that is free from *Control Dependencies*. We use the concept of *Equivalence Classes* to group statements based on their *Data Dependence* behavior. Statements that belong to an *Equivalence Class* are mutually dependent directly or transitively. On the other hand statements that belong to separate *Equivalence Classes* are dependence free and can be run in parallel without compromising on the program correctness. With this arrangement of program statements we claim that the program run time is now equal to the run time of the class that runs the longest. While this scheme of grouping program instructions, can be viewed as a method of parallel conversion, we use this method here specifically for parallel performance estimation and prediction. After surveying the published literature, and searching for similar commercial products, we did not find a comparable technology, to assess the contributions made by Caliper, at the time of writing, and so we claim that Caliper is the only product of its kind today.

M. H. Miraz et al. (Eds.): iCETiC 2020, LNICST 332, pp. 16–39, 2020.
https://doi.org/10.1007/978-3-030-60036-5_2

Keywords: Analytical Model · Coarse model · Mathematical model · Maximum Available Parallelism · MAP · Parallel estimation · Parallel prediction · Performance estimation · Performance prediction · Program Shape · Program Shape Flattening · Speedup After Parallel Conversion · SAP

1 Introduction

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*. 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 behavior 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 coming sections, shall provide details, of our research activities and their outcomes.

The paper, is organized as follows: Sect. 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 which follows, presents Caliper in action, from the concept of an example program, in a higher level, imperative language. Section 6 is dedicated to Competitive Analysis, which is a study to assess, how Caliper fares against the opposition, in academia and industry. Finally we conclude the paper, after highlighting the contributions of our work, with the research community, in perspective.

2 Previous Work

Early methods of converting, serial code to a parallel form, was a manual process, was error prone and tedious, and not very productive. Its successor, was a parallel conversion process, that involved both, the programmer and the compiler. The programmer supplied the hints and pointed out sections of code and data, that were parallel friendly, and the compiler provided a working solution.

There are many research projects that tried out the semi-manual, hints based approach. For instance the authors of [8], added parallel extensions, to a standard imperative language, to empower the language for deployment, in both shared memory by providing thread support, and in a distributed setting, through message passing and mailbox support. The authors of [30] propose a parallel conversion library, for Object Oriented Software, which supports both, Shared Memory and Cluster Paradigms. Standardization efforts, led to the design of OpenMP, which is based on the Shared Memory Tasking model, and uses clauses added to existing imperative languages [2]. This was complemented later, by the Message Passing Interface (MPI), a solution for use in the Distributed Environments, and is structured as a library [21]. OpenMP and MPI have been used in the production of, industrial strength software, especially in the scientific domain. However the fact remains that, all these approaches require programmer time and effort, to supply the hints, which translates to a loss of programmer productivity, and this prompted researchers to find better solutions.

Automated Parallel Conversion, does not involve the programmer, and the solution is entirely provided by the compiler, and has received attention of several research groups, over the years. The central aspect of this approach, is the underlying mathematical model used. Models based on Symbolic Algebra, Linear Algebra, Polyhedra and Graphs are popular [26], and the quality of the results generated, are in most cases, closely tied with the model. Polyhedra model is attractive for parallel conversion, due to its simplicity. The models has been used to extract data dependencies, and to enable transformations for shared memory multi-core targets [6]. Polyhedral models also referred to as the Polytope models, have been extensively used for loop optimizations, including Unrolling, Slicing, Sanitization, etc [34]. However the model is limited to affine expressions involving index and induction variables, and may not yield results, when dealing with irregular loop expressions, such as accesses involving sparse matrices.

Graphs are used extensively, to represent various kinds of information, including Program Dependence and Flow [23,42]. Graph models are used to detect, iteration dependencies of loops [38]. Graphs were used to generate, data dependencies and profiles of Object Oriented Programs, to solve data partitioning problem, and generate synchronization and communication [13]. Researchers have used Graph Models along with Analytical Models, to solve, Work Distribution, Communication Overheads, and Data Locality issues in Distributed Environments [16,18]. Graphs are very intuitive, widely used and the most popular of all the models, for studying program behavior. However, Graphs without a good representation, can consume large amounts of memory.

When we have a choice of models to use, for dependency analysis, how do we pick? Authors of [40] present results of an experimental evaluation, to help choose an appropriate model. How can a complete Parallel Optimization solution be structured? Authors of [11] present an Integrated Graphical Environment, based around an code editor metaphor, with support for debugging and user feedback, through code annotations.

Recent Parallel Optimization related research efforts, seem to increasingly pivot around, the data gathered at run time by profiling and sampling. Examples include, the Binary Rewrite approach to parallel conversion [32], Inter-procedural Analysis [5], Parallel conversion of Irregular Loops [4] Real World Loops with irregular structure [33]. Analysis of Object Oriented Programs through speculation [22], Combined static analysis with sampling [20], Hardware Centric Dynamic approach [10], Dynamic Feedback through sampling [7]. However all dynamic schemes, which gather behavior data, using Profile and Sampling, are biased towards the input samples used, and the resulting program coverage. Schemes that use executables as source of parallel transformations are limited by the extent of the metadata present. Speculation centric schemes are simple to implement, since they bypass extensive analysis upfront, but pay a price when hit with program dependency conflicts, which requires sufficient time and effort handling rollbacks. Schemes that focus entirely on one particular aspect of a program, such as Loops, pay a price when confronted with, programs of different genres.

When it comes to Performance Studies and Predictions how do we get the necessary information? Should we use Static Analysis Methods or resort to Dynamic Schemes? Opinions are divided across the research community. A great many researchers tread the middle ground and use both methods referred to here as Hybrid Schemes, in their research.

Authors have used Static Schemes since the beginning for Performance Estimation and Prediction. Here are some examples, Support Vector Machines (SVM) kernel techniques were used to reach Data Partitioning decisions [3], Machine Learning coupled with Performance Models were used to predict speedup [15], Static analysis was used to decide Program Distribution on Message Passing Architectures [19], Loop Distribution and Array Access Patterns were studied using static methodologies [17], Algebraic Expressions of Variables coupled with Analytical Models to collect Execution Times of different Program Sections [1], Call Graphs combined with Markov Model of Control Flow to generate Function Call Frequency estimates [44] Critical Path Analysis along with Execution Time model was used to predict run times [24], A Postmortem Analysis Tool, based on idleness and overheads to point out areas of improvement, [43]. Static Schemes are based on Code and Data Analysis, and the knowledge gained as a result of the analysis. They are more universal, since they do not use any dynamic data, collected on a target machine, by sample runs of the given program.

Other researchers have used Dynamic Schemes, for Performance Estimation and Reporting. Here are a few examples, Compile Time Models, augmented with profile data, was used for performance prediction [9], Compiler Generated Instrumentation was used to develop Performance Models [14], Parallelism Identification and Advice Tool was created using profile data [31], Instrumentation Tool was created to operate at the Program Section level using trace techniques [37], Visualization System was developed that uses Software Instrumentation and Hardware Counters [41], Deterministic Replay Debug Tool was created using Trace Driven Simulations and Models [47] A Dynamic Binary Instrumentation (DBI) framework was created, to build heavy weight tools, for analysis and profiling [39], Industrial strength Software Instrumentation Tool Set, was created for profiling, performance study, and defect fixing [35]. Information collected by Dynamic Schemes are tainted by the Sample Inputs used, and the Hardware bias of the target machine, used in the experiment. But researchers have carefully constructed input sets, and laboriously formulated data gathering scenarios to limit the sampling errors.

Hybrid Schemes have been employed by a few researchers in recent times. Here are a few examples, A Transformation Framework that uses Architecture Specific Cost Models, for estimation coupled with Dynamic Feedback as a supplement [45], Source Code Instrumentation along with Instruction Scheduling on Simulated Architectures to estimate performance [46], Performance Prediction Tool using Code Analysis and Trace Simulation [12], A toolkit for Static Analysis and Dynamic Measurements to develop Architecture Neutral Models [36]. Hybrid Schemes are attractive at a higher level, since they are supposed to provide, the benefits of both Static and Dynamic Schemes. But merging information collected through two separate sources, is always a tricky problem, exacerbated by the scenario, when the results from the two sources of collection, do not converge.

How important are the Performance Estimation and Prediction steps, in the overall Parallel Conversion process? A programmer, would like to know how much parallel potential, a program has, before he commences the conversion work. This information provided when required, can boost programmer productivity. Researchers have usually avoided this step, due to the unavailability of all the pieces of information, in the early phases of the Program Compilation process. When it is eventually available, several phases downstream, they feel that the information is outdated and useless. This has led to the wide spread belief that performance estimation is hard.

All prior research studies in the estimation and prediction domain, have relegated the performance estimation, to a later phase in the compilation process, when detailed analysis results are available, essentially ignoring the benefits of early assessment. In this research work we perform our estimation and prediction when it matters, before detailed analysis and program transformations have been carried out, and there is still an opportunity, to use the predictions for driving future transformations, including parallel conversions. This paper is entirely dedicated, to the problem of performance estimation and reporting, which is just one solution, in the tool chest of a compiler. We have reported our other research findings, elsewhere in other publications.

3 Asterix

Caliper is a parallel opportunity, 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:

- *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 [25].
- *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.
- *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.

- *Graft*
  This module 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.
- *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, transformations that enable other transformations including parallel conversions [29]. 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.
- *Transgraph*
  This is the module 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.
- *Paragraph*
  This is the module, that 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.
- *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.
- *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 [27]. The various views include, Annotated Source and ASIF

IR, Caliper Predictions, 3PO Oracles, Prunes, Proxies, their distribution and orchestration

- *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 [28]. 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.

The Fig. 1 illustrates the different phases involved in the operation of the Asterix compiler.

The block diagram is intuitive for the most part. As seen from the figure, the higher level source program is input to the *Paracite* module and passed through a series of modules, each represented by a block in the diagram. The arrows pointed towards and away from their blocks, signify either the input(s) or the output(s) from the module respectively. The final step in the chain is the Orchestration of all the parallel run time components and combining the results in a coherent fashion, handled by the *Concerto* module. Readers may refer to the earlier descriptions of the phases of Asterix, which are synonymous with the modules here.

4 Caliper

The main objective of the Caliper module, is to provide 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.

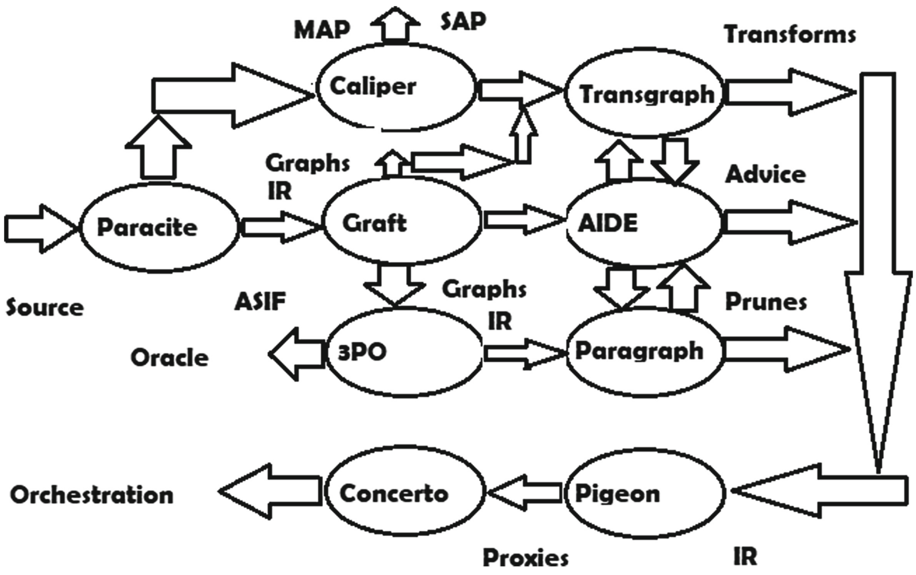

Fig. 1. Block diagram of Asterix phases and operation

The Caliper operation is characterized by the following phases:

– *Function Call Expansion*
  The purpose *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.
– *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.
– *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, com-

puting to temporary result variables, offer a powerful and compelling solution, to implement Control-Predication.

We next describe the purpose of the following performance metrics:

- *Maximum Available Parallelism*
  Maximum-Available-Parallelism, *MAP* for short, 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.
- *Speedup After Parallel Conversion*
  Speedup-After-Parallel Conversion, *SAP* in short form, 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 fraction.

The Fig. 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.

4.1 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.

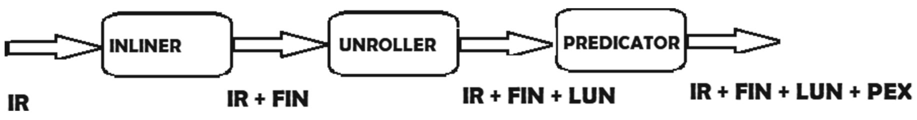

Fig. 2. Block diagram of Caliper steps and operation

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 measuring *Serial-Execution-Cycles*. Here C_{CYC} is the count of cycles, to run the serial version of the program, N_{INC} is the instruction count, for the given program.

$$C_{SER} = N_{INC} \tag{1}$$

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 honor, 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} 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.

$$C_{PAR} = \max\left(EQC_1, EQC_2, \ldots, EQC_n\right) \tag{2}$$

3. *Maximum Available Parallelism*
Maximum Available Parallelism, abbreviated as MAP is a measure of the inherent parallelism available in a program, and is reported as a percent of the total program instructions. The following equation precisely defines the metric. 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.

$$MAP = (C_{SER} - C_{PAR}) \div C_{SER}) \times 100 \tag{3}$$

4. *Speedup After Parallel Conversion*
Speedup After Parallel Conversion, SAP for brevity, is an estimate of how much faster the program will run, after parallel conversion. The equation that follows, describes the metric. 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.

$$SAP = (C_{SER} \div C_{PAR}) \tag{4}$$

4.2 Program Shape Flattening

As alluded to 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. Refer to Algorithm 1 for the detailed steps.

4.3 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

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, n do                    ▷ sweep through function calls in the program
 8:         GET_FUNCTION_DEFINITION(Def, Fnc)    ▷ fetch code block defined 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
```

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. Refer to Algorithm 2 for the detailed steps.

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
    equivalence class list
17:         MERGE(R, S2)      ▷ merge class S2 to class R and update global parallel
    equivalence class list
18:     end for
19: end procedure
```

5 Analysis

Given below is a code listing of a program, in a popular imperative language, which is used for illustrating the workings of Caliper. For listing see Listing 1.1 The program has the three structural components we alluded to earlier, which are hindrances for estimation and prediction purposes, namely a function, loop and a conditional block. See lines numbered 2, 9 and 24 for these blocks.

Listing 1.1. A program with a function, loop and condition

```c
#include <stdio.h>
float cal_function() {
  int a, b, c, d, e;
  a = 10; b = 20;
  c = 30; d = 40;
  e = a + b * c / d;
  return e;
}
float cal_loop() {
  int i, j, k;
  float sum[3][3][3];
  float ssum = 0.0;
  for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
      for (k = 0; k < 3; k++) {
        sum[i][j][k] = i + j + k;
      }
  for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
      for (k = 0; k < 3; k++)
        ssum += sum[i][j][k];
  return ssum;
}
float cal_condition() {
  int x = 10;
  float y = 0.0;
  if (x < 10) {
    y = 2 * x;
  }
  else if (x == 10) {
    y = x * x;
  }
  else {
    y = x * x * x;
  }
  return y;
}
int main() {
  float x, y, z;
  x = cal_function();
  y = cal_loop();
  z = cal_condition();
  printf("x = \%f\n", x);
  printf("y = \%f\n", y);
  printf("z = \%f\n", z);
}
```

The next listing consists of, the equivalent program in ASIF, with all the program structures preserved, namely the functions, loops and conditionals untouched. See Listing 1.2 for the code listing. The translated higher language code for the function, loop and conditional, can be found under the respective labels named as such. See lines 4, 18, 70 in the listing. ASIF code is easy to follow. Each block starts with a label, and so there is one for each function, loop and conditional. See lines 5, 20, 71 and 91. At the start of the block are the declarations, DCL is the opcode to define an integer and FDCL for a float. STP is the push and POP is pop. The arithmetic operators have easy to spot Mnemonics. FNC is the Call and RET is the return opcode.

Listing 1.2. ASIF Code

```
! start entry
start:
! cal_function entry
cal_function:
  DCL a, 4
  DCL b, 4
  DCL c, 4
  DCL d, 4
  DCL e, 4
!
  MUL T1, b, c
  DIV T2, T1, d
  ADD e, a, T2
!
  STP e
  RET
! cal_loop entry
cal_loop:
!
  DCL i, 4
  DCL j, 4
  DCL k, 4
  FDCL sum, 4, 27
  FDCL ssum, 4
  MOV i, @0
for1_entry:
  MOV j, @0
for2_entry:
  MOV k, @0
for3_entry:
  ADR T5, &sum
  MUL T5, i, j
  MUL T5, T5, k
  ADD T6, i, j
  ADD T6, T6, k
  MST T5, T6
  INC k
  BLE for3_entry, k, 3
for3_exit:
  INC j
  BLE for2_entry, j, 3
for2_exit:
  INC i
  BLE for1_entry, i, 3
for1_exit:
  MOV i, @0
!
for4_entry:
  MOV j, @0
for5_entry:
  MOV k, @0
for6_entry:
  ADR T7, &sum
  MUL T7, i, j
  MUL T7, T7, k
  MLD T8, T7
  ADD, ssum, ssum, T8
  INC k
  BLE for3_entry, k, 3
for6_exit:
  INC j
  BLE for2_entry, j, 3
for5_exit:
  INC i
  BLE for1_entry, i, 3
for4_exit:
  STP ssum
  RET
! cal_condition entry
cal_condition:
  DCL x, 4
  FDCL y, 4
  JGE LB1, x, @10
  NOP
  MUL T3, @2, x
  MOV y, T3
  JMP LB2
LB1:
  NEQ LB2, x, @10
  MUL T4, x, x
  MOV y, T3
  JMP LB3
LB2:
  MUL T4, x, x
  MOV y, T3
LB3:
  STP y
  RET
```

```
! main entry
main:
  FDCL x
  FDCL y
  FDCL z
  FNC cal_function
  POP x
  FNC cal_loop
  POP y
  FNC cal_condition
  POP z
! end entry
end:
```

The next program listing consists of the ASIF code after it has passed through the structure filter which expands all functions, unrolls all loops and coverts conditionals to predicated blocks. See Listing 1.3 for reference. See line 8 for the inlined function cal_function. See lines starting from 25 for the first of the unrolled loop nests. The second unrolled loop nest starts from line 76. Each unrolled iteration is marked with a number under comment to designate the code for the corresponding iteration. At the end of each unrolled iteration is the handling (management) of index (induction) variables appropriate for the iteration. Finally the condition block starting at line 123 starts the block. PGE is a predicate which evaluates to True or False depending on the condition check and set the result variable appropriately. CMOV is the conditional move that moves the value to the result variable if the earlier predicate evaluated to true and not otherwise. There are some architectures belonging to the Very long instruction word (VLIW) class which allow predicated versions of all arithmetic operators in which case CMOV will be unnecessary. But the important thing to notice is the absence of jumps and labels which have been removed prior to estimation and prediction.

Listing 1.3. Flattened ASIF Code

```
! start entry
start:
! main entry
main:
  FDCL x
  FDCL y
  FDCL z
  !x = FNC cal_function
  DCL a, 4
  DCL b, 4
  DCL c, 4
  DCL d, 4
  DCL e, 4
  !
  MUL T1, b, c
  DIV T2, T1, d
  ADD e, a, T2
  MOV x, e
  !y = FNC cal_loop
  DCL i, 4
  DCL j, 4
  DCL k, 4
  FDCL sum, 4, 27
  FDCL ssum, 4
! FIN block starts, #0
  MOV i, @0
  MOV j, @0
  MOV k, @0
  ADR T5, &sum
  MUL T5, i, j
  MUL T5, T5, k
  ADD T6, i, j
  ADD T6, T6, k
  MST T5, T6
  INC k
! #1
  ADR T5, &sum
  MUL T5, i, j
  MUL T5, T5, k
  ADD T6, i, j
  ADD T6, T6, k
  MST T5, T6
  INC k
! #2
  ADR T5, &sum
  MUL T5, i, j
  MUL T5, T5, k
  ADD T6, i, j
  ADD T6, T6, k
  MST T5, T6
  MOV k, @0
  INC j
! #3 <#4 - #7 snipped>
! #8
  ADR T5, &sum
  MUL T5, i, j
  MUL T5, T5, k
  ADD T6, i, j
  ADD T6, T6, k
  MST T5, T6
  INC i
  MOV j, @0
  MOV k, @0
! #9 <#10 - #25 snipped>
! #26
  ADR T5, &sum
  MUL T5, i, j
  MUL T5, T5, k
  ADD T6, i, j
  ADD T6, T6, k
  MST T5, T6
  INC i
  MOV j, @0
  MOV k, @0
! FIN block ends
! FIN block starts, #0
  MOV i, @0
  MOV j, @0
  MOV k, @0
  ADR T7, &sum
  MUL T7, i, j
  MUL T7, T7, k
  ADD T8, i, j
  ADD T8, T8, k
  MLD T8, T7
  ADD ssum, ssum, T8
  INC k
! <#1 snipped> #2
   ADR T7, &sum
   MUL T7, i, j
   MUL T7, T7, k
   ADD T8, i, j
   ADD T8, T8, k
   MLD T8, T7
   ADD ssum, ssum, T8
   MOV k, @0
   INC j
! #3 <#4 - #7 snipped>
! #8
  ADR T7, &sum
  MUL T7, i, j
  MUL T7, T7, k
  ADD T8, i, j
  ADD T8, T8, k
  MLD T8, T7
  ADD ssum, ssum, T8
  INC i
  MOV j, @0
  MOV k, @0
! <#9 - #25 snipped>, #26
  ADR T7, &sum
  MUL T7, i, j
  MUL T7, T7, k
  ADD T8, i, j
  ADD T8, T8, k
  MLD T8, T7
  ADD ssum, ssum, T8
  INC i
  MOV j, @0
  MOV k, @0
! FIN block ends
  MOV y, ssum
  !z = FNC cal_condition
  DCL x3, 4
  FDCL y3, 4
  PGE TP1, x1, @10
  MUL T3, @2, x3
  CMOV y3, T3, TP1
  PEQ TP2, x3, @10
  MUL T4, x3, x3
  CMOV y3, T3, TP2
  AND TP3, TP1, TP2
  NOT TP3, TP3
  MUL T4, x3, x3
  CMOV y3, T3, TP3
  MOV y, y3
!
end:
```

5.1 Reporting Estimates and Prediction

For the present, Caliper reports interesting numbers in plain text in Csv format as shown below:

```
CALIPER,,,
(Performance_Estimation_and_Prediction_Tool),,,

1., Serial Instruction Count, SIN,  472
2., Equivalence Class Count, EQC,  10
3., Mean Instruction Count, MIN,  65.8
4., Parallel Instruction Count, PIN,  249
5., Serial Execution Cycles, SEC,  472
6., Parallel Execution Cycles, PEC,  249
7., Maximum Available Parallelism, MAP,  47.25
8., Speedup After Parallel Conversion, SAP,  1.9
```

However the future version will be enhanced, to report more information in graphical format, and will be integrated with AIDE.

Table 1. Caliper performance estimates and prediction

CALIPER (performance estimation and prediction tool)			
Sl. no.	Metric name	Code	Value
1	Serial Instruction Count	SIN	472
2	Equivalence Class Count	EQC	10
3	Mean Instruction Count	MIN	65.8
4	Parallel Instruction Count	PIN	249
5	Serial Execution Cycles	SEC	472
6	Parallel Execution Cycles	PEC	249
7	Maximum Available Parallelism	MAP	47.25
8	Speedup After Parallel Conversion	SAP	1.9

Table 1 displays the same numbers in tabular form for clarity purposes.

Serial Instruction Count is the number of instructions, detected by Caliper. Equivalence Class Count is the number of Parallel Equivalence Classes detected. Mean Instruction Count is the average count of instructions, in each class. Parallel Instruction Count is the maximum value of instruction count, among all the classes. Serial Execution Cycles is the number of execution cycles consumed, by the given program, when operating in serial mode. It should be noted that, since Caliper is designed to be a Coarse Performance Estimator and Predictor, we treat all instructions the same. Each instruction is assumed to take a cycle, for its execution and so this metric has the same value as the Serial Instruction Count. Parallel Execution Cycles is the total processor cycles, required to run the program in parallel mode, which is same as the Parallel Instruction Count. Maximum Available Parallelism as defined earlier, is a measure of the inherent parallel potential, of the given program. Speedup After Parallel Conversion is a multiple, that measures how much faster, the parallel version of the program runs, in comparison to the serial version.

The column with the heading *Code*, is the abbreviation for the metric and the column with the heading *Value* is the value reported for the corresponding metric. The program fed to Caliper, is the same program we saw earlier, and the numbers reported, are the estimates from Caliper. The two most interesting numbers are MAP which is reported as 47.25 and a SAP of 1.9. This means that, about half of the program is parallel convertible, and it will run 1.9 times faster than the serial version, after parallel conversion.

6 Competitive Analysis

We started a search of the research publications, for a solution similar to Caliper. Since the survey revealed, the absence of a comparable product in the research domain, we focused our search to the state of the art, in the industry.

We short listed the competition to the following major players in the domain, Gcc from GNU, Clang from LLVM group and Parallel Studio and Icc from Intel. We learned that none of them have a Parallel Performance, Estimation and Prediction component, that is comparable with Caliper.

However we wanted to study the results of parallel conversions, made by these compilers. We first searched for a compiler flag, that can emit diagnostics, specific to the Optimizations and Parallel Conversions, being carried out. We found a few flags we thought were relevant, such as the -openmp-report from icc. But either they were non operational, or the information required, for the comparative study was missing. Similarly, the flag to turn on the Auto Parallel conversion feature was either a place holder or missing at this point. However all of the above compilers are enabled for OpenMP and use them for parallel conversions.

We continued our quest for a parallel build enabling the openmp feature with these compilers. We used the following pragma or directive to enable the following block for possible parallel conversion.

#pragma omp parallel

We experimented by placing the pragma at various points in the source code such as the following, at Main function entry, at other Function entry points, at the start of Loops and Multi-level loops, and Control entry points, to see if it makes any difference to the LLVM assembly generated by the Clang compiler, since the -emit-llvm flag to Clang was the only viable option, to generate diagnostics. We did not see any trends in the parallel code generated, which could be used to make any useful comparison with Caliper's estimates or prediction.

The Intel Parallel Studio and its associated icc provide a flag called -openmp-report, which apparently is supposed to generate diagnostics, but we had trouble enabling after installation, so we could not get any useful data on icc also.

At this point we have to conclude the study, and claim that in comparison with the available state of the art, both in academia and industry, Caliper is the only working, Performance Estimation and Prediction Solution available, at the time of writing.

7 Conclusion

Caliper is a coarse performance estimator and predictor solution, for a given serial program, that is scheduled for parallel conversion. It operates on the IR code generated from the program, after the translation phase of compilation and provides an early indicator, about the program's expected parallel behavior, after transformation. Our solution, to the performance assessment problem, involves two phases. We perform a coarse assessment of performance, at the start of the compilation pipeline, and postpone the detailed estimation and prediction for a phase that follows the analysis phase. That way, the results of coarse prediction, are available to drive transformation, and parallel conversion decisions. The structural components of a program such as Procedures, Loops and Conditionals, their count and placement adds an unique character to the program which are referred to collectively as the *Program-Shape*. *Program-Shape* is an hindrance to effective performance estimation and prediction. So prior to the estimation phase, we perform transformations, such as *Function Call Expansion*, *Loop Unrolling* and *Control Predication* that specifically target these program structures. The transformations are referred to collectively as *Program-Shape-Flattening*. The transformation eases the process of measurement and prediction, by transforming the given program, to straight line code. After flattening transformations, we are also freed of the concerns, of Control Dependencies and they stop being a factor, in parallel performance. To collect the parallel run time estimates, we use the concept of *Equivalence-Classes*. An *Equivalence Class* collects objects, that share the class property. We use *Equivalence Classes* here to group instructions, that are mutually data dependent on each other. We refer to these classes either as *Dependence-Classes* or *Parallel-Classes*. Since instructions belonging to separate *Dependence Classes* share no dependencies, they can be safely scheduled for parallel execution. Effectively, we have converted a serial program to a set of *Dependence Classes* that can be run concurrently. Such a scenario of parallel execution of a program allows us to conclude, that the run time of the program, is now equal to, the run time of the *Dependence-Class* that runs the longest. Finally we report two numbers that we believe are of interest to a parallel programmer namely, *Maximum-Available-Parallelism* which is a measure of the inherent parallel conversion potential of the program, also referred to in short form as *MAP* and *Speedup-After-Parallel Conversion*, which predicts the expected speedup after parallel conversion, also referred to as *SAP* for the purpose of brevity. After a thorough search of published literature and comparative study of the practicing state of the art, we are convinced that there is no viable research or commercial product, that can be compared to Caliper at the time of writing.

References

1. Arapattu, D., Gannon, D.: Building analytical models into an interactive performance prediction tool. In: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, Supercomputing 1989, pp. 521–530. ACM, New York (1989). https://doi.org/10.1145/76263.76321. http://doi.acm.org/10.1145/76263.76321
2. Ayguade, E., et al.: The design of OpenMP tasks. IEEE Trans. Parallel Distrib. Syst. **20**(3), 404–418 (2009). https://doi.org/10.1109/TPDS.2008.105
3. Balasundaram, V., Fox, G., Kennedy, K., Kremer, U.: A static performance estimator to guide data partitioning decisions. SIGPLAN Not. **26**(7), 213–223 (1991). https://doi.org/10.1145/109626.109647. http://doi.acm.org/10.1145/109626.109647
4. Blume, B., et al.: Polaris: the next generation in parallelizing compilers. In: Proceedings of the Workshop on Languages and Compilers for Parallel Computing, p. 10-1. Springer, Heidelberg (1994)
5. Blume, W., Eigenmann, R.: 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 - Volume 02, ICPP 1994, pp. 233–238. IEEE Computer Society, Washington, DC (1994). https://doi.org/10.1109/ICPP.1994.59

6. Bondhugula, U., et al.: Towards effective automatic parallelization for multicore systems. In: 2008 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, pp. 1–5 (2008). https://doi.org/10.1109/IPDPS.2008.4536401
7. Bradel, B.J., Abdelrahman, T.S.: Automatic trace-based parallelization of Java programs. In: 2007 International Conference on Parallel Processing (ICPP 2007), p. 26 (2007). https://doi.org/10.1109/ICPP.2007.21
8. Canetti, R., et al.: The parallel C (pC) programming language. IBM J. Res. Dev. **35**(5.6), 727–741 (1991). https://doi.org/10.1147/rd.355.0727
9. Cascaval, C., DeRose, L., Padua, D.A., Reed, D.A.: Compile-time based performance prediction. In: Carter, L., Ferrante, J. (eds.) LCPC 1999. LNCS, vol. 1863, pp. 365–379. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44905-1_23. http://dl.acm.org/citation.cfm?id=645677.663790
10. Codrescu, L., Wills, D.S.: On dynamic speculative thread partitioning and the MEM-slicing algorithm. In: Proceedings of 1999 International Conference on Parallel Architectures and Compilation Techniques, pp. 40–46 (1999). https://doi.org/10.1109/PACT.1999.807404
11. Cooper, K.D., et al.: ParaScope: a parallel programming environment. Proc. IEEE **81**(2), 244–263 (1993)
12. Cornea, B., Bourgeois, J.: A framework for efficient performance prediction of distributed applications in heterogeneous systems. J. Supercomput. **62**, 1609–1634 (2012). https://doi.org/10.1007/s11227-012-0823-5
13. Diaconescu, R., Wang, L., Mouri, Z., Chu, M.: A compiler and runtime infrastructure for automatic program distribution. In: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium, p. 25a (2005). https://doi.org/10.1109/IPDPS.2005.7
14. Diniz, P.C.: A compiler approach to performance prediction using empirical-based modeling. In: Sloot, P.M.A., Abramson, D., Bogdanov, A.V., Gorbachev, Y.E., Dongarra, J.J., Zomaya, A.Y. (eds.) ICCS 2003, Part III. LNCS, vol. 2659, pp. 916–925. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44863-2_90. http://dl.acm.org/citation.cfm?id=1762418.1762519
15. Dubach, C., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M.F., Temam, O.: Fast compiler optimisation evaluation using code-feature based performance prediction. In: Proceedings of the 4th International Conference on Computing Frontiers, CF 2007, pp. 131–142. ACM, New York (2007). https://doi.org/10.1145/1242531.1242553. http://doi.acm.org/10.1145/1242531.1242553
16. Fahringer, T.: Using the P3T to guide the parallelization and optimization effort under the Vienna Fortran compilation system. In: Proceedings of the Scalable High-Performance Computing Conference, pp. 437–444 (1994). https://doi.org/10.1109/SHPCC.1994.296676
17. Fahringer, T.: On estimating the useful work distribution of parallel programs under P3T: a static performance estimator. Concurr. Pract. Exp. **8**, 261–282 (1996)
18. Fahringer, T., Scholz, B.: Symbolic evaluation for parallelizing compilers. In: Proceedings of the 11th International Conference on Supercomputing, ICS 1997, pp. 261–268. ACM, New York (1997). https://doi.org/10.1145/263580.263648. http://doi.acm.org/10.1145/263580.263648
19. Fahringer, T., Zima, H.P.: A static parameter based performance prediction tool for parallel programs. In: Proceedings of the 7th International Conference on Supercomputing, ICS 1993, pp. 207–219. ACM, New York (1993). https://doi.org/10.1145/165939.165971. http://doi.acm.org/10.1145/165939.165971

20. Garcia, S., Jeon, D., Louie, C., Taylor, M.B.: The kremlin oracle for sequential code parallelization. IEEE Micro **32**(4), 42–53 (2012). https://doi.org/10.1109/MM.2012.52
21. Gropp, W., Gropp, W.D., Lusk, E., Skjellum, A., Lusk, A.D.F.E.E.: Using MPI: Portable Parallel Programming with the Message-Passing Interface, vol. 1. MIT Press, Cambridge (1999)
22. Hammacher, C., Streit, K., Hack, S., Zeller, A.: Profiling Java programs for parallelism. In: 2009 ICSE Workshop on Multicore Software Engineering, IWMSE 2009, pp. 49–55 (2009). https://doi.org/10.1109/IWMSE.2009.5071383
23. Horwitz, S., Reps, T.: The use of program dependence graphs in software engineering. In: Proceedings of the 14th International Conference on Software Engineering, pp. 392–411 (1992)
24. Jeon, D., Garcia, S., Louie, C., Taylor, M.B.: Kismet: parallel speedup estimates for serial programs. SIGPLAN Not. **46**(10), 519–536 (2011). https://doi.org/10.1145/2076021.2048108. http://doi.acm.org/10.1145/2076021.2048108
25. Kalyur, S., Nagaraja, G.S.: 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), pp. 7–12 (2016). https://doi.org/10.1109/CSITSS.2016.7779431
26. Kalyur, S., Nagaraja, G.S.: A survey of modeling techniques used in compiler design and implementation. In: 2016 International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), pp. 355–358 (2016). https://doi.org/10.1109/CSITSS.2016.7779385
27. Kalyur, S., Nagaraja, G.S.: AIDE: an interactive environment for program transformation and parallelization. In: 2017 2nd International Conference on Computational Systems and Information Technology for Sustainable Solution (CSITSS), pp. 1–5 (2017). https://doi.org/10.1109/CSITSS.2017.8447848
28. Kalyur, S., Nagaraja, G.S.: Concerto: a program parallelization, orchestration and distribution infrastructure. In: 2017 2nd International Conference on Computational Systems and Information Technology for Sustainable Solution (CSITSS), pp. 1–6 (2017). https://doi.org/10.1109/CSITSS.2017.8447691
29. Kalyur, S., Nagaraja, G.S.: A taxonomy of methods and models used in program transformation and parallelization. In: Kumar, N., Venkatesha Prasad, R. (eds.) UBICNET 2019. LNICST, vol. 276, pp. 233–249. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-20615-4_18
30. Kaminsky, A.: Parallel Java: a unified API for shared memory and cluster parallel programming in 100% Java. In: 2007 IEEE International Parallel and Distributed Processing Symposium, pp. 1–8 (2007). https://doi.org/10.1109/IPDPS.2007.370421
31. Kim, M., Kim, H., Luk, C.K.: Prospector: a dynamic data-dependence profiler to help parallel programming. In: HotPar 2010: Proceedings of the USENIX Workshop on Hot Topics in Parallelism (2010)
32. Kotha, A., Anand, K., Smithson, M., Yellareddy, G., Barua, R.: Automatic parallelization in a binary rewriter. In: 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 547–557 (2010). https://doi.org/10.1109/MICRO.2010.27
33. Lazarescu, M.T., Lavagno, L.: Dynamic trace-based data dependency analysis for parallelization of C programs. In: 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 126–131 (2012). https://doi.org/10.1109/SCAM.2012.15

34. Lokuciejewski, P., Cordes, D., Falk, H., Marwedel, P.: A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In: International Symposium on Code Generation and Optimization, CGO 2009, pp. 136–146 (2009). https://doi.org/10.1109/CGO.2009.17
35. Luk, C.K., et al.: Pin: building customized program analysis tools with dynamic instrumentation. SIGPLAN Not. **40**(6), 190–200 (2005). https://doi.org/10.1145/1064978.1065034. http://doi.acm.org/10.1145/1064978.1065034
36. Marin, G., Mellor-Crummey, J.: 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 2004/Performance 2004, pp. 2–13. ACM, New York (2004). https://doi.org/10.1145/1005686.1005691. http://doi.acm.org/10.1145/1005686.1005691
37. Miller, B.P., et al.: The paradyn parallel performance measurement tool. Computer **28**(11), 37–46 (1995). https://doi.org/10.1109/2.471178
38. Navarro, A., Zapata, E., Padua, D.: Compiler techniques for the distribution of data and computation. IEEE Trans. Parallel Distrib. Syst. **14**(6), 545–562 (2003). https://doi.org/10.1109/TPDS.2003.1206503
39. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. **42**(6), 89–100 (2007). https://doi.org/10.1145/1273442.1250746. http://doi.acm.org/10.1145/1273442.1250746
40. Psarris, K., Kyriakopoulos, K.: An experimental evaluation of data dependence analysis techniques. IEEE Trans. Parallel Distrib. Syst. **15**(3), 196–213 (2004). https://doi.org/10.1109/TPDS.2004.1264806
41. de Rose, L.A., Reed, D.A.: SvPablo: a multi-language architecture-independent performance analysis system. In: Proceedings of the 1999 International Conference on Parallel Processing, ICPP 1999, pp. 311–318. IEEE Computer Society, Washington (1999). http://dl.acm.org/citation.cfm?id=850940.852859
42. Sarkar, V.: Automatic partitioning of a program dependence graph into parallel tasks. IBM J. Res. Dev. **35**(5.6), 779–804 (1991)
43. Tallent, N.R., Mellor-Crummey, J.M.: Effective performance measurement and analysis of multithreaded applications. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2009, pp. 229–240. ACM, New York (2009). https://doi.org/10.1145/1504176.1504210. http://doi.acm.org/10.1145/1504176.1504210
44. Wagner, T.A., Maverick, V., Graham, S.L., Harrison, M.A.: Accurate static estimators for program optimization. SIGPLAN Not. **29**(6), 85–96 (1994). https://doi.org/10.1145/773473.178251. http://doi.acm.org/10.1145/773473.178251
45. Wang, K.Y.: Precise compile-time performance prediction for superscalar-based computers. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI 1994, pp. 73–84. ACM, New York (1994). https://doi.org/10.1145/178243.178250. http://doi.acm.org/10.1145/178243.178250
46. Wang, Z., Sanchez, A., Herkersdorf, A.: SciSim: a software performance estimation framework using source code instrumentation. In: Proceedings of the 7th International Workshop on Software and Performance, WOSP 2008, pp. 33–42. ACM, New York (2008). https://doi.org/10.1145/1383559.1383565. http://doi.acm.org/10.1145/1383559.1383565
47. Zhai, J., Chen, W., Zheng, W., Li, K.: Performance prediction for large-scale parallel applications using representative replay. IEEE Trans. Comput. **65**(7), 2184–2198 (2016). https://doi.org/10.1109/TC.2015.2479630