IR-Level Dynamic Data Dependence Using Abstract Interpretation Towards Speculative Parallelization

Rasha Omar, Ahmed El-Mahdy, Erven Rohou

IR-Level Dynamic Data Dependence Using Abstract Interpretation Towards Speculative Parallelization

Rasha Omar*†, Ahmed El-Mahdy*‡, and Erven Rohou§
* Department of Computer Science and Engineering
Egypt-Japan University of Science and Technology
Alexandria, Egypt
† Faculty of Computers and Artificial Intelligence, Benha University, Egypt
‡ Alexandria University, Egypt
§ Univ Rennes, Inria, CNRS, IRISA

***Abstract*—Recently, with the wide usage of multicore architectures, automatic parallelization has become a pressing issue. Speculative parallelization, one of the most popular automatic parallelization techniques, depends on estimating probably-parallelized code parts. This in turn motivates the employment of data dependence detection techniques for these code parts to report whether they contain dependence or not in order to be parallelized. In this paper, we propose a runtime data-dependence detection technique that is based on abstract interpretation at the intermediate representation (IR) level. We apply our proposed approach on the most frequently visited blocks of the code, hot loops. Unlike most existing approaches in which data analysis occurs at compile time, our proposed method conducts the analysis immediately while interpreting the code, which in turn saves the analysis time for potentially parallelized loops. Specifically, the proposed technique depends on the concept of abstract interpretation to analyze the hot loops at runtime. This process is done by firstly computing the abstract domain for each hot loop program points. Each abstract domain is incrementally computed, till a fixpoint is achieved for all program points, and correspondingly the analysis terminates in order to consecutively detect the existence of data dependence. Once the analysis result reports a parallelization possibility for the finished hot loop, the interpreter invokes the compiler to resume the execution in a parallel fashion as recommended by our proposed approach. The proposed technique is implemented on LLVM compiler, then used to test the dependence detection for a set of kernels on the Polybench framework, and the data dependence analysis required for each kernel is studied in terms of the computation overhead.**

Keywords*—*Abstract Interpretation, Dependence Analysis, Dynamic Analysis, Parallelization.

I. Introduction

Nowadays, parallelization in multicore systems represent one of the challenging research topics. All parallelized programs need various particular preparations to run efficiently and correctly. Therefore, there are different techniques to enhance the usage of multicore systems. One of these techniques, Speculative Parallelization (SP), is used to anticipate whether the instruction pair could be parallelized or not [1]. Dynamic profiling and SP are more popular than static approaches because of their ability to handle any analysis during runtime [2]. The SP mainly requires analysis for every program point, an arc between pair of instructions, in order to detect whether there is a dependence between instructions or not. Therefore, SP could decide whether the code part could be parallelized or not based on computations carried out during analysis. This analysis might be implemented statically at compile-time or dynamically at run-time [3], [4].

There are several techniques that are used to analyze programs to extract dependent instructions statically. One of the most well-known techniques is Abstract Interpretation (AI). AI is a static analysis approach that combines ideas from compiler optimization and verification communities; it relies on the abstraction (or approximation) of program states (program semantics) to generate a superset of all possible states (abstract collective semantics) at arbitrary program points [5], [6]. While data-flow analysis is currently the dominant analysis approach, AI is showing strong potential owing to its strong linkage between language and analysis semantics [7].

This work conducts AI dynamically to be applied for dynamic analysis. We propose to extract hot loops (HLs) in order to be analyzed at runtime, during interpretation. The HLs are represented by the most seen strongly-connected iterated basic blocks. The analysis is applied on the hot trace of HLs' program points. Our system mainly employs AI during execution to compute abstract states, abstract intervals, correctly. Therefore, the analysis will then use the computed abstract states to detect dependence correctly. Moreover, our dependence approach would recommend parallelizing/serializing the currently executed analyzed HL. The interpreted code would pause till invoking the compilation using JIT compiler to run the analyzed HL using SP exploiting the produced analysis during interpretation without requiring further compilation analysis pass, as in typical interpretation/compilation systems.

Our approach is fully automatic, so there is no need to stop the current run to insert any safeguards or directives manually. Therefore, the system could be used with SP without re-compilation or re-execution. This approach receives the source code to analyze, then SP system could proceed with

parallelization within the same run[1].

The AI dependence at runtime subsystem is implemented using the LLVM Compilation Framework. LLVM is able to support programs with lifelong analysis and transformation for arbitrary programs by supplying the compiler transformations with high-level information. This information could be provided at compile-time, link-time, runtime, or at idle time. The main privileges of using this compiler are its open-source property and platform independence. Therefore, it could be run by different front-end, high-level programming languages and it is used with a wide range of hardware architectures. Moreover, our technique is applied on LLVM Intermediate Representation (IR) which is a powerful code representation. The IR is human-readable, so it is able to supply the means to debug and display the performed transformations [8]. The system could detect the loop-carried dependencies and intra-iteration dependencies.

The main contributions of our paper are as follows:

- A dynamic dependence analysis method based on AI on interpreter and compilation styled system.
- Implement the corresponding analyzer into the LLVM compilation framework (for the interpreter engine).
- Conduct an initial study about the performance of the analyzer in terms of correctness and overhead.

The paper also explains how our system would be exploited by the speculative parallelizer to execute the parallel loops. We propose to resume executing HLs using a speculative parallelizer JIT compiler.

This analysis is accurate for the current loop execution, but not necessarily for future loop executions. However, we could guarantee the correctness by inserting guards on the trace entry. These guards check whether the current input trace as well as the program state (or procedure input), etc, are changed or not. If there is a change, the analysis should be redone, taking into consideration both current and earlier collected abstract semantics.

The remainder of the paper is organized as follows: Section II is for the work related to the dependence analysis at compile-time and runtime. Section III provides the required background of parallelization, AI, and LLVM. Section IV explains the core concept proposed by our approach of dynamic AI analysis. Section V explains the main proposed system design and the implementation details for our dependence detection technique. Section VI presents the results and a comparison with static AI analysis. Section VII includes conclusion for our work and illustrates the intended future work.

II. Related Work

Bhattacharyya et al. [9] propose a technique using the polyhedral model to analyze the program statically, at compile-time. This approach could execute a program using automatic SP by Polly's polyhedral dependence analysis. There are two different heuristics to find the speculative parallelizable code parts. The first heuristic is called may-dependence to run the loops speculatively. However, the other heuristic extracts the cold loops using the profile information. The loops with actual runtime dependence are excluded because these loops are not appropriate to run with the speculative parallelizer.

Rugina et al. [10] proposed a technique to parallelize the recursive functions. The technique utilizes the pointer and symbolic analyses to provide the system with the independent recursive calls. The provided information permits the compiler to extract the procedure calls to be executed in parallel. The method is statically applied to generate the code which is re-executed concurrently without any violation.

Gupta et al. [11] studied the algorithms of divide-and-conquer. These algorithms are applied using a static analysis technique at compile-time in order to utilize the analysis of the symbolic arrays to detect dependence. This technique is implemented for an SP system.

Bondhugula et al. [1] proposed an approach that used the polyhedral model to implement a source-to-source transformation framework. This framework is end-to-end fully automatic which computed using an integer optimization framework. The optimization finds the best option for tiling. Tiling is applied to improve locality aspects utilizing affine transformations to generate parallel code for imperfectly nested loops

There are well-known approaches which are based on the polyhedral model. These approaches analyze the code statically by formatting the loop nests in a mathematical representation as polyhedra. The main computed facets are produced from performing some computations on loop bounds. The polyhedral transformations are commonly utilized in the analysis performed in static compilers' intermediate representation [12].

Pradelle et al. [13] proposed an approach to parallelize statically binary code. High-level information is extracted by parsing the binary information. The extracted information is utilized to generate a C program which is parallelized by polyhedral parallelizer. Therefore, the C compiler re-introduces and re-compiles the original source semantics. Thus, this approach requires mainly high-level program re-generation in addition to re-compilation and re-execution.

Jimborean et al. [14] proposed a dynamic speculative polyhedral parallelization. The technique is based on compiler-generated skeletons which are applied at runtime on the original code via polyhedral transformations. These skeletons are produced at compile-time to be picked out and represented at runtime. This technique requests the computation of all loop bounds and memory access functions in order to affining the functions of the outer loop iterators.

Yukinori et al. [15] studied a system which monitors a binary code to check the data dependencies between memory references and dynamic loop- or call-contexts. Then, the analysis extracts the data-flow of the memory dynamically in order to re-execute the program with parallelization technique correctly.

Rus et al. [16] proposed a hybrid analysis which uses both static and dynamic analyses. The static analysis is used to verify memory reference properties. It could extract the independence conditions from the dependence main equations during compile-time. The independence conditions are evaluated at runtime to predicate the ability to parallelize the loop. The dependence equations are not checked whether they are

[1] An earlier stage work has been published in PLOS One journal with title: Binary-level data dependence analysis of hot execution regions using abstract interpretation at runtime

true or false because this part is not the main scope of their work. Therefore, the correctness of the system is not addressed in the dependence check.

Fonseca et al. [17] studied an automatic parallelization system. This system analyzed the memory access by understanding the dependencies between two program parts at compile time. These dependent program parts would read from and write to the same memory location. Therefore, these two program parts would not be parallelized. The approach identified the instructions which would be parallelized. The system also could extract some instructions' signatures from the program source code. These extracted signatures include the dependency and control flow information. Thus, this information would help the system to arrange the parallelized instructions into task-oriented structure. The main problem here is that the system requires the main source code of the program. Moreover, the analysis is performed at compile-time.

The AI is used to detect dependencies statically. Ricci et al. [18], proposed a static AI technique to analyze the loop using abstract domains at compile-time. The technique is implemented using the Program Analyzer Generator PAG [19] which includes set of codes to facilitate the application of the technique. Furthermore, Tzolovski et al. [20] have initially studied some properties of abstracting dependence such as the iteration data dependence graphs and dependence distance. However, the practical implementation details are missed in this study.

Unlike the previously mentioned works, our approach is both automatic and dynamic. Our system conducts AI analysis at runtime which in turn makes it able to provide accurate dependence detection. Moreover, our technique does not restrict the speculative system to re-compile or re-execute. Also, the framework is implemented using LLVM which can be used with various hardware architectures and front-end programming languages. We introduce dynamic dependence detection at runtime and accordingly suggest the parallelization style that should be followed. Therefore, we aim to apply the concept of parallelization using the collaboration of LLVM interpreter and JIT compiler. Furthermore, our dynamic dependence approach would be preferred than static method because of handling the pointer aliasing. The pointer aliasing occurs when there are two pointers containing two same values. Our technique would accurately detect the dependence in this case. However, there is no available values in static methods which complicates detecting dependence during compilation time.

III. Background

This paper aims to apply dynamic data dependence analysis using AI in order to carry out parallelization. Therefore, in this section we provide a brief illustration of parallelization, AI analysis technique, and LLVM compiler.

A. Parallelization

There are various parallelization techniques which are applicable with many compilers. These techniques are classified into two major categories. The first category is based on the scheme of inspector-executor which aims to extract loop with some directives. This extracted loop works as inspector to lead the executor of the original loop [21]. The second category is the speculative parallelization which executes the code in parallel. Moreover, at the same time, a reference monitors the data dependence in order to avoid possible violations. Generally, the data dependence analysis for speculative systems is studied over loop indices. These indices are used mainly with arrays. The violations may occur while accessing memory [22], [23].

The data dependence analysis should be found accurately, therefore the parallelization technique could have the information needed to prevent violation. The main dependence violation types are illustrated in [24] as follows:

- **Write-After-Read (WAR)** A write happens before an earlier read in the program order to the same memory location.
- **Read-After-Write (RAW)** a read happens before an earlier write in the program order to the same memory location.
- **Write-After-Write (WAW)** a write occurs before an earlier write in the program order to the same memory location.

SP proceeds in three main steps. The first step defines all the required memory operations regarding the speculative execution. These operations are extracted from the possible parallelized loops or code parts which are determined. The operations represent the main data used to compute the dependence using any data analysis technique. The second step feeds the parallelizer at runtime with the speculative current state. This state includes the speculative data extracted at first step to detect whether there is a violation or not. If the state does not contain dependence the data are committed. Third step tests whether there is a dependence or violation occurred. If so, the system has the ability to roll-back till last committed operations and resumes the program sequentially [25]. Apparently, the main motivation of this article is to study a new technique which provides the speculator with the required analysis dynamically within the same run. This analysis solves the problem of extracting dependence of the HL. Furthermore, the system would give the compiler the chance to continue execution with the parallel/serial exectution according to the analysis result at early iterations.

B. Abstract Interpretation Analysis

AI is a technique which is used to analyze the code statically. This technique depends basically on abstracting the semantics of each program. The abstraction would be applied on different abstract domains. There are main concepts related to AI, which could be defined as: [26], [5].

- **Concrete domain** D_c is the original object, the program point variables values that AI technique is applied on it.
- **Abstract domain** D_a is to replace the original objects, values of the variables in each program point (S), by their abstraction $\alpha(S)$. This abstraction would be computed according to the target of each technique. In our method, we used abstract interval as the main abstract domain.

- **Abstraction function** (α) maps the concrete object into its abstract interpretation.
- **Concretization function** (γ) is the inverse of the abstract function which maps an abstract domain to the concrete domain $S \subseteq \gamma\ (\alpha\ (S)\ )$.

In our approach, we define the abstract domain as the abstract interval computed from the variables values. Moreover, the abstract interval keeps the collective semantics of each program point at runtime.

C. LLVM Compilation Framework

LLVM is an open source compilation framework. LLVM includes high-quality components with interfaces to be appropriate the different purposes in wide range of architectures. LLVM includes transformation passes which are exploited to be applied on Intermediate Representation, IR, in different levels, for example Modules, Functions, BasicBlocks, etc, to perform some computations and tasks [27].

The IR is a well-defined representation for programs which is language independent, architecture independent, human-readable and easy to use. IR is used for analysis and optimization. Furthermore, this representation provides Static Single Assignment (SSA) which guarantees that each variable is assigned once. In LLVM, each variable is assigned to a typed register. The main benefit of SSA is the simplification of variables properties in different compiler optimization levels [28].

LLVM Transformation Pass is an important part of LLVM. It provides the compiler with different optimizations and transformations applied on code which enables the compiler to compute instrumentation results. Clearly, a transformation pass can mutate and modify IR code according to the pass functionality. Furthermore, the pass can extract information from IR to compute some specific details. Every transformation pass is implemented by overriding some methods included in LLVM. These methods are determined and implemented depending on the corresponding pass operation and the required changes [29].

IV. Dynamic Abstract Interpretation

Program semantics define the relation between input and output states for each statement/instruction. The state takes its values from a domain, known as the concrete domain. In AI, the state is mapped into an abstract state with the corresponding pre-defined abstract semantic functions.

At runtime, every IR instruction is mapped to an abstract equation I which includes the input abstract intervals, at right-hand side, and output abstract interval, at left-hand side. Moreover, the interpretation here defines the abstract semantic. The analysis is then carried out by iterating through I assignments, until reaching a fixpoint. The solution for each abstract equation indicates that all fixpoints are reached. It is worth noting that the obtained abstract state represents 'collective' trace semantics, which are all possible values at all program points for all possible executions of the program [5], [30].

The mapping is sound, such that ordering relations are maintained (Galois connection).

Thus, AI could be formally defined as a tuple $\langle D_a, D_c, \alpha, \gamma, I\rangle$. The symbol D_a would be known as a complete lattice with ordering $\leq$, join operations $\cup$, and intersection operations $\cap$. Moreover, this lattice includes a lattice bottom $\perp$ and top $\top$. Furthermore, the functions of abstraction, α and concretization, γ, define a connection called 'Galois' connection which formalizes the abstraction at each program point as follows:

$$\forall i \in D_c, i \leq \gamma(\alpha(i)) \tag{1}$$

and

$$\forall j \in D_a, \alpha(\gamma(j)) \leq j \tag{2}$$

Consider the following C++ program as an example to illustrate our approach:

```
// C++ Program Example Code
int a[500];
int i = 1;
bool rare = false;
...
while (i<200) {
  j = i + 200;
  while (j<500) {
    a[i] = 2*a[j];
    j++;
    if (rare) {
       a[i] = a[i-1];
    }
  }
  i++;
}
```

The example presents two nested loops which would contain a general case. Apparently, if our method could correctly handle the dependence problem for this loops, therefore the method could deal with different cases of HLs. The example has no loop carried dependence, except when rare branch is true (line 11). HLs extraction excludes 'rare' branch because it is not included in the main hot trace of the loop.

```
// C++ Code with AI Equations
int a[500];
1. a_a = a_a1 = [<address of a>,<address of a>]
int i = 1;
2. i = i2 = [1,1]
while (i<200) {
 3. i = (i ∩ [−∞,199]) ∪ i3
    j = i + 200;
 4. i = i4 ∪ i
    j = j4 = j4 ∪ (i + [200,200])
  while (j<500) {
  5. j = (j ∩ [−∞,199]) ∪ j5
    a[i] = 2*a[j];
  6. a_ai = a_ai6 = a_ai6 ∪ (a_a + i)
     a_aj = a_aj6 = a_aj6 ∪ (a_a + j)
    j++;
  7. j = j7 = j7 ∪ (j + [1,1])
    if (rare) {
       a[i] = a[i-1];
    }
  }
 8. j = j ∩ [500,∞] ∪ j8
  i++;
 9. i = i9 = i9 ∪ (i + [1,1])
}
10. i = i10 ∩ [200,∞] ∪ i10
```

The abstract semantics statement (equations) are defined at each control-flow edge between each two instructions. The program collective state is described with the variables at each program point (suffixed with a number, e.g. `a_a`), and the global variables. The initial interval for uninitialized variable is considered as $\emptyset$. Moreover, the initial intervals for array values would be [-∞, ∞]

Each equation updates the state at its corresponding program point by accumulating previous states and compute the new instruction state according to its semantic (please note that $\cup$ indicates union, and $\cap$ indicates intersection operations). Thus, `i++` translates into `i = i9 = i9 ∪ (i + [1,1])`, indicating that the global `i` is set to `i9` (the value of i at this program point); and both of them take the value of old `i9` joined with current value of i added to the interval $[1, 1]$. The abstract intervals of i and j present all possible value in the current program edge. The analysis is terminated upon reaching a fixpoint on all visited program points. Therefore, if rare is true, the analysis would terminate. However, guards are inserted into non analyzed equations such that the underlying speculator would recover the correct state.

Please note that the equations are monotonic, therefore the obtained intervals grow. The above analysis stops when reaching a fixpoint. However, the system employs the widening on the intervals in some cases to reach the fixpoint. For the above example, we notice that line 7 has the interval of j is $[201, 202]$, we extend j to be $[201, \infty]$. The analysis advances until reaching a fixpoint for the inner loop. The analysis then continues for the outer, similarly reaching a fixpoint. The intervals for `a_ai6=[A+1,A+199]` and `a_aj6=[A+201,A+499]` are showing that there is no intersection and therefore no loop carried dependence.

In Section V, we explain the main points of our proposed system. Therefore, we need to clarify the main targeted code part in our method, the hot loop.

Hot Loops (HLs) are the loops which contain strongly connected basic blocks. These blocks are repeatedly executed during runtime in the visited trace. We target the visited program points during the execution. We use a transformation pass to instrument the loop during early execution. Therefore, HLs are extracted as a preparatory step before the execution. After execution, every HL basic blocks are provided with special titles/names that are identified by our modified LLVM interpreter.

V. Proposed Dynamic AI System

Section IV has provided a conceptual view of our proposed method. In this section, we illustrate our system design and implementation details. Briefly, we could explain the main steps as follows:

1) The input source code is compiled by LLVM compilation framework to generate the corresponding IR.
2) The generated IR is inserted to an LLVM transformation pass to extract the HLs according to the number of execution for each basic block in IR to generate a new annotated IR with HLs.
3) The annotated IR is executed by LLVM interpreter till reaching entry basic block for HL.
4) Our approach in LLVM modified interpreter is applied to the current HL at runtime to analyze the main trace for this HL and construct the AI equations at each program point to compute abstract intervals.
5) The fixpoint is checked on the produced intervals of all passed program points after each iteration.
6) Once the fixpoint is reached in all visited instructions, the analysis stops to compute the intersection in the next iteration. This fixpoint is computed after visiting all points. This intersection is computed between all AI intervals of each program points' pairs at each HL.
7) The intersection computations' results are inserted into a map to set flags into the dependent instruction pairs. Also, this map is generated at early number of iterations for each HL to be ready to run using SP. The approach could utilize the intersection results to recommend the SP to parallelize the current HL or not. Thus, the system flags dependent instructions as well as not considered exit edges (for not normal loop exit edge).
8) The execution resumes and invokes SP which considers the dependence flags map and recommendations. The SP would resume execution of the rest of iterations of the current HL using JIT compiler.
9) Finally, the execution of the rest of code is resumed using LLVM interpreter till reaching a new HL.

These steps can be decomposed into two main subsystems. The first subsystem is dynamic data dependence from step 1–7 which is implemented by modifying the interpreter of LLVM compilation framework. The main acquired information during runtime is the variables abstract intervals in the early iterations. After reaching the fixpoint, the dependence check is applied using the intersection. The second subsystem is SP, step 8–9, which is proposed to be implemented by mingling the LLVM interpreter and JIT compiler. Our framework is depicted in Figure 1. For clarification, we use the example explained in Section IV. The speculative parallelizer would use this map directly for the same run to execute in parallel and detect violations.

A. Hot Loops (HLs) Extraction and Detection (1-3)

LLVM front-end receives the source code to compile and extract the corresponding IR code. Moreover, the IR is inserted to an LLVM transformation pass to instrument IR code. Then, the HLs are extracted during early run. The output of this preparatory step is a new modified IR with identified HLs. The new IR is executed using LLVM modified interpreter till reaching a HL to begin our analysis. The LLVM interpreter would detect the HL using the added annotations.

B. Dynamic AI on HLs (4)

LLVM original interpreter interprets IR instructions in the concrete domain. The interpreter reads the actual values for each instruction's variable. Also, each instruction's operation is processed according to its original functionality with these actual values. The abstract operations are adapted to be applied to the arguments on abstract domain. In our method, the LLVM

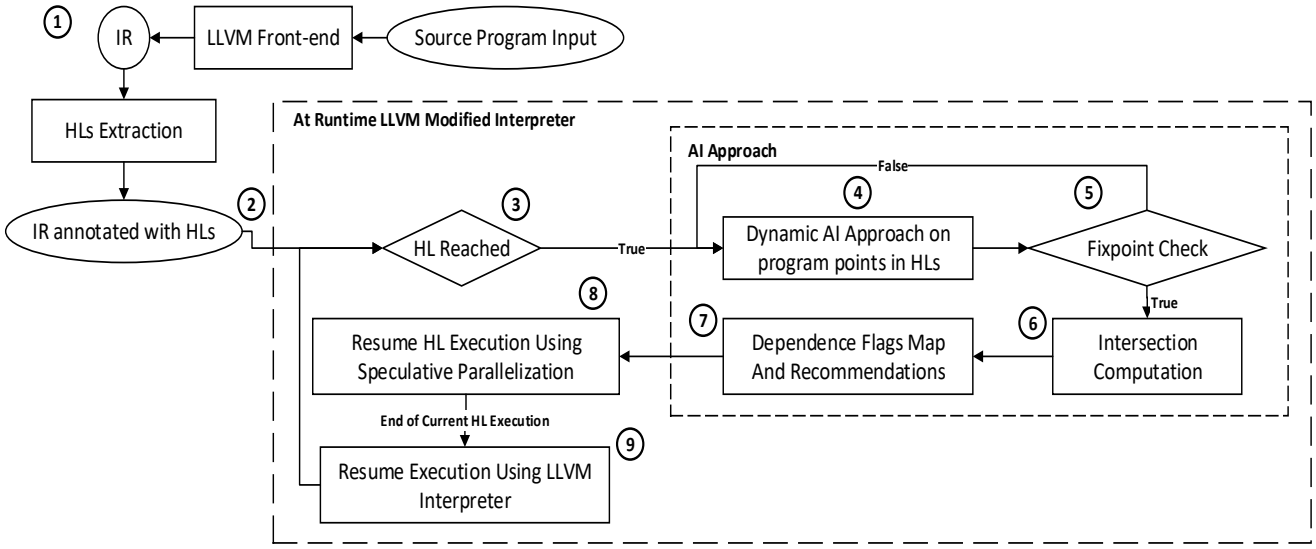

Fig. 1: A Flow Graph of Our System Operational Steps

interpreter has been extended to interpret the IR instructions of the hot trace of HLs on the abstract domain which is the abstract interval for each instruction's variable. The operations are processed with the new generated abstract intervals. The variables on either concrete or abstract domain may be memory addresses or any other type of values. Abstract domain and operations are computed during runtime in early iterations, thus the computations are correctly performed.

The functionality of operations would be briefly explained for LLVM IR different instructions and how our system interpret them using AI at runtime. For example, ***alloca*** instruction is used to define the variable. Each allocated variable is loaded, ***load***, to temporary variables to perform binary and other operations like ***add***, ***sub***, ***sext***, etc. Then, some of the results in temporary variables are stored, ***store***, back to original memory. Our approach applies the abstract interpretation over all LLVM IR operations. Each operation includes arguments to refer to the predecessors. Each predecessor may be a value, an address, or an operation of a predecessor instruction. After applying the abstract operations, some arguments are updated with the new abstract intervals whether these arguments are values, addresses or operations, specially ***alloca*** instruction. We can illustrate the abstract operation by the following equation example, `a[i] = 2*a[j]`, the part of reading `a[j]`, (line 11 in the code explained in Section IV):

```
Instruction1:
%17 = load i32, i32* %j, align 4
arg0:     %j = alloca i32, align 4
Read Interval [201, 499 ]
Address Interval [ 0x50d1420, 0x50d1420 ]
Instruction2:
%a = alloca [500 x i32], align 16
arg0: i32 1
At address 0x50d06e0 Array 500 elements
Array Index [0, 499 ]
Instruction3:
%19 = getelementptr inbounds [500 x i32],
[500 x i32]* %a, i64 0, i64 %18
arg0:    %a = alloca [500 x i32], align 16
arg1: i64 0
arg2:    %18 = sext i32 %17 to i64
Index Interval [ 201, 499 ]
Address Interval    [ 0x50d1fe4, 0x50d4540 ]
```

The first instruction presents the ***load*** of j variable which is the index in array a. Also, it is able to get the lower bound (LB) and upper bound (UB) for it during the execution. The computed interval for this instruction will be used in a successor ***sext*** instruction later. The second instruction allocates the array in the memory with its total number of elements. The third instruction is ***getelementptr*** or GEP that computes an array element address. This instruction has three main arguments. The first argument ***arg0*** is utilized to present the array base address and length. Moreover, the third argument ***arg2*** is used to specify the current index. From these two used arguments, our system could get the intervals of addresses and indices.

C. Fixpoint Check (5)

The proposed method abstracts index accesses as well as the corresponding memory addresses without considering the array content. Therefore, a read operation would return the interval $[-\infty, \infty]$. While iterating through the abstract assignments, the obtained intervals are widened till the fixpoint is reached. This could be achieved by setting a corresponding widening interval bound to an UB/LB.

We could briefly explain the widening step in our implementation by the following cases: First case is when the variable is the iterator of loop, so the LB and UB could be deduced from

the loop condition. Second case is for the regular variables used for different operations, the abstract interval is widened to $[-\infty, \infty]$. Third case exploits the result of the first case to deduce the addresses interval of the array, if the index is directly used as our used kernels. For array indices, the system obtains the LB and UB of each array access instruction. The index variables are applied to widening using their computed LB and UB determined from loop condition.

Some abstract operations are performed over the HL hot trace. These operations are converted from the concrete one by monitoring the inserted values in the first iterations to get the monotonicity of the intervals. The implementation is done in LLVM interpreter to compute the abstract domain, instruction by instruction. Then, this domain is applied to the binary operations using the abstract intervals instead of concrete values. The monitoring step is done during *load* and *store* in first number of iterations. The loop iterates over arrays, so we need to get the fixpoint. This fixpoint is found when the indices and all variables used in loop instructions have reached to the final widened intervals. These intervals of addresses, variables and indices values, are converted to abstract domain. Furthermore, the technique checks that these intervals are fixed after number of iterations. Most cases are converted into their final abstract state at the second iteration, so mostly the analysis converges at the next iteration. Thus, the analysis cuts off to compute dependence check, then continues the loop normally.

D. Intersection Computation (6)

Our system computes the dependence between the instructions pairs by intersecting the corresponding abstract memory addresses intervals. To illustrate, consider that the two instructions' abstract address intervals are $[I_l, I_u]$ and $[J_l, J_u]$. Equation 3 shows the corresponding intersection operation:

$$\text{Intersection} = [I_l, I_u] \cap [J_l, J_u] \tag{3}$$

The following part of generated output shows the intersection between the intervals from another for-loop example:

```
Instruction1:    %38 = load i32, i32* %37, align 4
 Read From [ 0x4407a20,  0x4409300 ]
Instruction2:    store i32 %39, i32* %42, align 4
 Write to : [ 0x4407a20,  0x4409300 ]
```

If all intersection operations result in $\emptyset$, the loop iterations are independent; otherwise, there is a dependency between one or more instructions' pairs, thus the loop could not be parallelized. The intersection result is stored in a map to instruct the speculation system that there is a dependence at the current program point. This map would assist the speculator to decide the parallelization ability for every analyzed program arc.

E. Dependence Flags Map and Recommendations (7)

The output is a map that consists of two different instructions and a flag, with values *true* for dependent pairs, and *false* for independent pairs. Also, our system could send a flag for parallelization recommendation *1* for parallel possibility and *0* for sequential execution.

F. Resuming Current HL Execution in SP (8–9)

Typical execution environments, such as HotSpot for the Java bytecode [31], rely on adaptive compilation. An interpreter first runs the bytecode without any startup delay. While executing, the interpreter collects information about the frequency of functions and various regions, such as loops. When a function is deemed critical enough, based on a predefined recompilation policy, the system decides to invoke a JIT compiler to produce native code. Low optimization levels guarantee short compilation time. Again, the code is monitored using our interpreter. When a second threshold is reached, the JIT compiler is invoked again, at a higher optimization level. The process repeats until the most aggressive optimizations are applied. By doing so, the systems only spends compilation time on the critical regions, and optimization time is recouped.

Our approach could detect from step 7 whether the current HL has the ability to be parallelized or not. As shown in Figure 2, by detecting during interpretation that a loop is parallel and no dependent instruction pairs at the same HL, our system has the ability to immediately apply parallelization (a typically aggressive optimization), hereby skipping the intermediate optimization levels. Parallelization is applied to the running loop, for the remaining iterations.

Figure 3 illustrates how JIT execution would deal with the dynamic AI analysis output. The speculation/parallelization subsystem shown in Figure 3 illustrates generally the process sequence after dependence check in order to execute the remaining code in current analyzed HL in parallel or serial. We propose to apply the SP technique of Yusuf et al. [32]. This SP technique exploits the on-stack replacement which deals with the dependence according to our approach. After dependence extraction, the SP technique would fork new process to enter the speculative state and kill the violated process. A serial program version is executed as a process simultaneously with the parallel execution. The serial process is suspended at specific checkpoints. These checkpoints are used to detect if any violation occurred during runtime to abort parallel execution and resume with the serial execution. If there is no detected problems, the checkpoint commits the acquired work in parallel execution.

The SP technique of Yusuf et al. [32] would be suggested to be applied at the highest level of optimization. This optimization jump would be useful for executing the program in parallel whenever our method generate its recommendations for current HL. The running HL analysis would be correct for the same analyzed trace. In rare cases, if the trace would change, the SP would take the decision to resume at parallel or serial. Moreover, it would be able to roll-back for any occurred violation. Therefore, the interpreter jumps to the highest level of optimization to resume the remaining iterations using SP in IR-level. After the current HL is interpreted and executed whatever in parallel or serial, the interpretation is resumed in LLVM interpreter till reaching a new HL.

We concentrate mainly on the detection of dependent pairs, therefore the last part is not the main interest of this paper. Moreover, the experiments in Section VI discusses the results applied by our method from step 1 to 7. Therefore, our

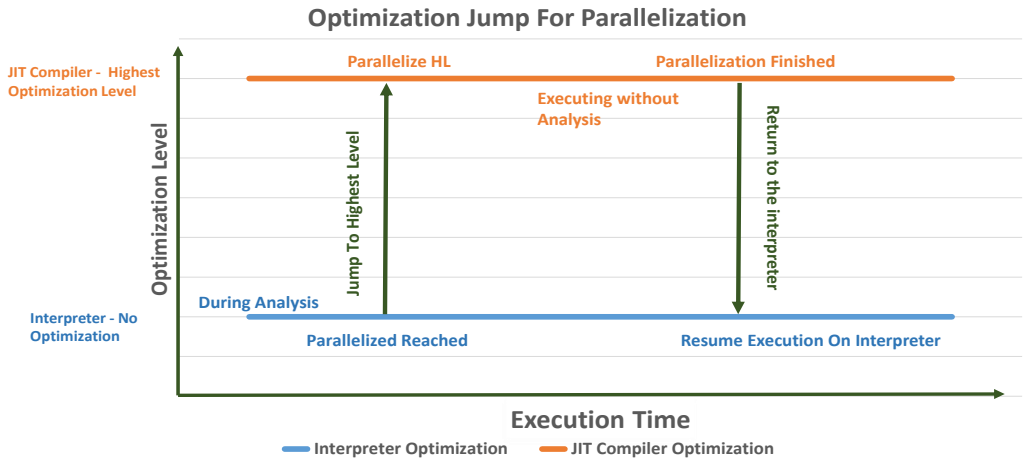

Fig. 2: Parallel HL detected from our analysis and recommendation, go immediately to highest optimization level to execute the HL in SP

dynamic dependence detection technique is examined using metrics of correctness and overhead. The proposed step 8 and 9 would receive the dependence flags map of dependent arcs and recommendation to execute the current HL in parallel or serial.

VI. Experimental Results

In this section, we study the main results generated by our proposed AI method shown in Figure 1. We used Intel Core i7-2670QM CPU 2.20 GHz x8. Moreover, the machine runs Ubuntu 14.04 LTS 64-bit Linux operating system. The approach is implemented on LLVM version 3.9.0.

We study our technique on a number of kernels of the Polyhedral Benchmark suite, Polybench [33]. The Polybench kernels are applied as a single file to compute the kernel instrumentation. Also, each kernel has loop bounds which are parametric in order to be applied with general-purpose implementation. The excluded kernels contain instructions that are not applicable with the original LLVM interpreter. The technique is applied in all hot loops in each kernel in the main two functions (init and kernel).

We have compared our approach with the well-known traditional static AI method [5]. The traditional technique of static AI is used to compute the abstract intervals during compile-time. We track the three main parts of each instruction at every program point, the operation, the arguments and type of these arguments. The operation is classified as read, write or neither. The arguments are checked whether their values are available at compile time or not. If the values of the operators are immediate, they will be used as abstract intervals with specified UB and LB. On the other hand, if the operators values are related to addresses that are not available at compile-time, the abstract intervals would be widened to [$-\infty$, ∞].

Table I explains the metric of correctness applied on our dependence technique. The first column refers to the kernel name. The second column represents the number of the extracted HLs in each kernel. Third column refers to the type of HL, nested or not. The fourth column is the true positives for our approach which indicates the dependent pairs which are actually dependent. The fifth column lists the false positives of our approach which are the dependent pairs which may not be actually dependent. If the numbers in the fourth and fifth columns equals 0, therefore there is no detected dependence. The sixth column adds the values of true and false positives. False positives' results are issued because of the IR trait of SSA. Sometimes, IR load and store operations are applied on induction variables in the same basic block which contains an operation. This operation would use the loaded value in an operation to be stored later in the same memory location. This load/store case will issue resolvable dependence. Therefore, our system tries to detect some of these code parts to be neglected during the dependence checking. There are some code parts which are not totally ignored. Hence, these non-ignored load/store cases cause resolvable dependence, false positives.

Example of false positives:

```
%12 = load i32, i32* %i, align 4
...
store i32 %12, i32* %i, align 4
```

There is another case might be extracted as false positives where the abstract intervals may intersect, in abstract domain, even the concrete values do not actually intersect, in concrete

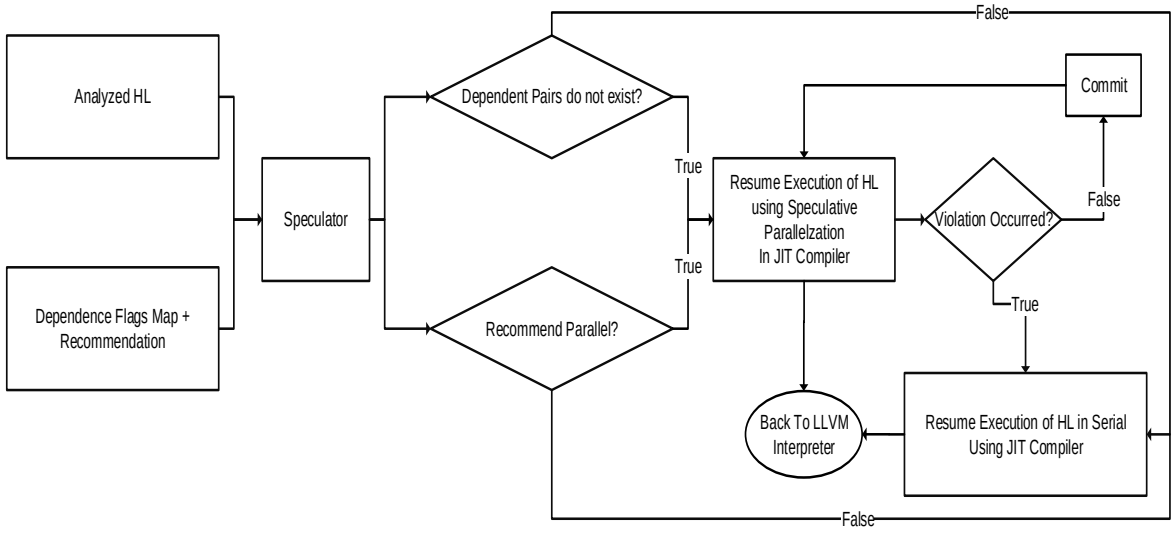

Fig. 3: Speculative Parallelization Subsystem Design

TABLE I: Extracted Dependent Pairs Results in Selected Kernels of Polybench

Kernel	No of HLs	Loop Type	True Positives (Our Approach)	False Positives (Our Approach)	Total Positives (Our Approach)	True Positives (Static AI)	False Positives (Static AI)	Total Positives (Static AI)
mvt	3	2-nested	0	0	0	0	3	3
		2-nested	2	0	2	2	2	4
		2-nested	2	0	2	2	3	5
bicg	2	2-nested	0	0	0	0	2	2
		2-nested	4	0	4	4	3	7
atax	3	2-nested	0	0	0	0	2	2
		for loop	0	0	0	0	0	0
		2-nested	4	1	5	4	2	6
gemver	4	2-nested	0	0	0	0	2	2
		2-nested	2	0	2	2	2	4
		2-nested	2	0	2	2	3	5
		2-nested	2	0	2	2	0	2
trmm	2	2-nested	0	0	0	0	2	2
		3-nested	4	2	6	4	2	6
gesummv	2	2-nested	0	0	0	0	2	2
		2-nested	4	2	6	4	3	7
syrk	4	2-nested	0	0	0	0	1	1
		2-nested	0	0	0	0	1	1
		2-nested	1	1	2	1	1	2
		3-nested	1	1	2	1	2	3
Accumulative Sum	-	-	28	7	35	28	38	66

domain. However, this case does not occur while we apply our method.

The remainder of Table I provides the static approach correctness results. The seventh and eighth columns are true and false positives which are related to the static method. The false positives for static method present that there are increased number of detected dependent pairs which are not actually dependent. These increased numbers explain the static approach main problem. This problem would be clear where there is no any dependence and the static method would result false dependence in the loop. The last column refers to the sum of true and false positives for the static approach.

For our approach, most of dependence pairs occur in the inner loop of the nested loops. Also, most of 2-nested for-loops in kernel function in each kernel program have dependency in inner loops, as in kernels of *mvt*, *bicg*, *atax*, *gemver*, *gesummv* and *syrk*. Moreover, the 3-nested loops include dependence in most inner loop, such as *syrk* and *trmm*. The true positives present the correctness that the extracted dependent pairs are actually dependent. Furthermore, the false positives explain that there are some extracted pairs that may not be dependent because of the IR instructions issues. The results present

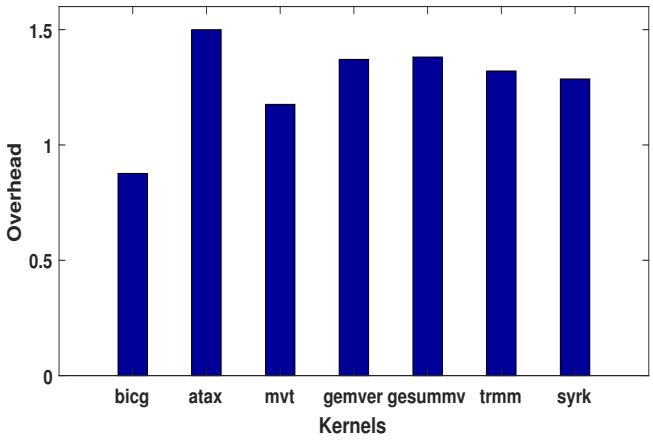

Fig. 4: The Overhead In Selected Kernels

that correct dependent pairs are detected. However, the false positives contain low number 0 in most kernels. The kernels *trmm* and *gesummv* actually include dependence. However, they also show maximum number, 2, in false positives which does not affect the accuracy. Also, the false negatives result the actual dependent pairs which are not extracted using our method. Regarding false negative, our analysis results always contain error-free code. During the execution, the dependent pairs of visited program points are detected. However, our approach may miss some opportunities. The last row represents the accumulative sum of the values. The accumulative sum of false positives of our approach is 7. In other hand, the value of static approach accumulative sum is 38. Thereby, the correctness for the dynamic system is higher than the static approach.

The missing opportunities mean that there are missing program points. These missing program points have never been passed, executed, during the current run of the hot trace in each hot loop. Finally, the non-mentioned for-loops are actually not included in hot loops. Thus, they are out of our concern.

We applied a simple static AI version by setting the initial values of uninitialized variables to [-∞, ∞]. The results of loops will be dependent in most cases which are actually incorrect.

Figure 4 presents the overhead in terms of the difference in execution time on the selected kernels running by original LLVM interpreter in comparison with our modified version. The overhead is computed using the following equation:

$$\text{Overhead} = (\text{Time}_m - \text{Time}_o)/\text{Time}_o \quad (4)$$

Overhead: refers to the main metric of execution time for our approach.

Time_m: refers to the execution time using our modified LLVM interpreter.

Time_o: refers to the execution time using original LLVM interpreter.

The overhead has occurred because of the computations done in the first number of iterations and conditions checking. These computations generate abstract intervals of the original concrete values to detect the dependence in all successor iterations. The resulting overhead is related to the number of HLs in the kernel as well as the computations and abstract operations applied in each HL. In our experiments, every kernel may contain two to four HLs, for-loop, 2-nested loop, 3-nested loop. These loops cause overhead increase because of each loop type criteria. Some of Polybench kernels are excluded because there are several IR operations which are not implemented in our LLVM modified interpreter. Moreover, the overhead would be diminished by the SP technique which would be applied in the same execution. The SP will be able to speedup the execution. The parallelization is able to decrease the programs execution time.

Our paper has presented a new automatic method which could be a strong dynamic support for SP systems. After number of iterations, the system would receive a dependence flags map for all instructions' pairs. Subsequently, this map would help our approach to recommend to the SP whether a HL is available to be parallelized or not. Thus, a correct decision to resume the execution in parallel or serial would be taken at the same run by the SP. Also, if any violations happen, according to the non-analysed instructions, the SP system would solve these violations using roll-back. Our approach is implemented using LLVM with its various features. The output results are accurate and the overhead is within the reasonable margins.

VII. Conclusions and Future Work

This paper investigated how to manage systems to detect data dependence at IR-level at runtime. The proposed analysis would be utilized in order to execute speculative parallelized system efficiently without re-compilation or re-execution. The proposed approach detects data dependence during program interpretation without requiring a separate analysis pass. The interpreter relies on conducting data dependence analysis on HLs through using AI. Our system applied the analysis in the LLVM interpreter and conducted a preliminary performance study on a set of kernels from the Polybench benchmark. The overhead range is from 0.88 to 1.49. Moreover, the results show accurate dependence analysis. Based on the analysis provided by our approach, we suggested how to manage the parallelization technique at the same run by jumping to the highest level of optimization to eliminate the overhead with more speedup. Our future work will consider implementing the speculative subsystem, where no further analysis is required as it is already conducted during interpretation; upon detecting no dependence, the interpreter can trigger immediately a high code generation pass, and skip intermediate passes. Moreover, future work would consider testing the system on full applications with irregular loops with the existence of complex control-flow structures generating multiple traces.

References

[1] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *ACM SIGPLAN Notices*, vol. 43, no. 6. ACM, 2008, pp. 101–113.

[2] M. Samadi, A. Hormati, J. Lee, and S. Mahlke, "Paragon: collaborative speculative loop execution on gpu and cpu," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*. ACM, 2012, pp. 64–73.

[3] G. Brat, J. A. Navas, N. Shi, and A. Venet, "Ikos: A framework for static analysis based on abstract interpretation," in *International Conference on Software Engineering and Formal Methods*. Springer, 2014, pp. 271–277.

[4] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai, "A cost-driven compilation framework for speculative parallelization of sequential programs," *ACM SIGPLAN Notices*, vol. 39, no. 6, pp. 71–81, 2004.

[5] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1977, pp. 238–252.

[6] P. Cousot, "Abstract interpretation based formal methods and future challenges," in *Informatics*. Springer, 2001, pp. 138–156.

[7] S. P. Midkiff, "Automatic parallelization: an overview of fundamental compiler techniques," *Synthesis Lectures on Computer Architecture*, vol. 7, no. 1, pp. 1–169, 2012.

[8] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[9] A. Bhattacharyya and J. N. Amaral, "Automatic speculative parallelization of loops using polyhedral dependence analysis," in *Proceedings of the First International Workshop on Code OptimiSation for MultI and many Cores*. ACM, 2013, p. 1.

[10] R. Rugina and M. Rinard, "Automatic parallelization of divide and conquer algorithms," in *ACM SIGPLAN Notices*, vol. 34, no. 8. ACM, 1999, pp. 72–83.

[11] M. Gupta, S. Mukhopadhyay, and N. Sinha, "Automatic parallelization of recursive procedures," *International Journal of Parallel Programming*, vol. 28, no. 6, pp. 537–562, 2000.

[12] S. Kobeissi and P. Clauss, "The polyhedral model beyond loops recursion optimization and parallelization through polyhedral modeling," in *IMPACT 2019, 9th International Workshop on Polyhedral Compilation Techniques, In conjunction with HiPEAC 2019*, 2019.

[13] B. Pradelle, A. Ketterlin, and P. Clauss, "Polyhedral parallelization of binary code," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 39, 2012.

[14] A. Jimborean, P. Clauss, J.-F. Dollinger, V. Loechner, and J. M. Martinez Caamaño, "Dynamic and speculative polyhedral parallelization using compiler-generated skeletons," *International Journal of Parallel Programming*, vol. 42, no. 4, pp. 529–545, Aug 2014. [Online]. Available: https://doi.org/10.1007/s10766-013-0259-4

[15] Y. Sato, Y. Inoguchi, and T. Nakamura, "Whole program data dependence profiling to unveil parallel regions in the dynamic execution," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2012, pp. 69–80.

[16] S. Rus and L. Rauchwerger, "Hybrid dependence analysis for automatic parallelization," *Technical Report, TR05-013*, 2005.

[17] A. Fonseca, B. Cabral, J. Rafael, and I. Correia, "Automatic parallelization: Executing sequential programs on a task-based parallel runtime," *International Journal of Parallel Programming*, vol. 44, no. 6, pp. 1337–1358, 2016.

[18] L. Ricci, "Automatic loop parallelization: an abstract interpretation approach," in *Parallel Computing in Electrical Engineering, 2002. PARELEC'02. Proceedings. International Conference on*. IEEE, 2002, pp. 112–118.

[19] F. Martin, "Pag–an efficient program analyzer generator," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 1, pp. 46–67, 1998.

[20] S. Tzolovski, "Data dependences as abstract interpretations," in *International Static Analysis Symposium*. Springer, 1997, pp. 366–366.

[21] S. Prema and R. Jehadeesan, "Analysis of parallelization techniques and tools," *International Journal of Information and Computation Technology*, no. 3, pp. 471–478, 2013.

[22] M. Cintra and D. R. Llanos, "Toward efficient and robust software speculative parallelization on multiprocessors," *ACM SIGPLAN Notices*, vol. 38, no. 10, pp. 13–24, 2003.

[23] P. Marcuello and A. González, "Clustered speculative multithreaded processors," in *Proceedings of the 13th international conference on Supercomputing*. Citeseer, 1999, pp. 365–372.

[24] M. Cintra, J. F. Martínez, and J. Torrellas, *Architectural support for scalable speculative parallelization in shared-memory multiprocessors*. ACM, 2000, vol. 28, no. 2.

[25] C.-L. Ooi, S. W. Kim, I. Park, R. Eigenmann, B. Falsafi, and T. Vijaykumar, "Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor," in *Proceedings of the 15th international conference on Supercomputing*. ACM, 2001, pp. 368–380.

[26] P. Cousot and R. Cousot, "Basic concepts of abstract interpretation," in *Building the Information Society*. Springer, 2004, pp. 359–366.

[27] C. Lattner, "Introduction to the llvm compiler infrastructure," in *Itanium conference and expo*, 2006.

[28] C. Lattner and V. Adve, "The LLVM Compiler Framework and Infrastructure Tutorial," in *LCPC'04 Mini Workshop on Compiler Research Infrastructures*, West Lafayette, Indiana, Sep 2004.

[29] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formalizing the llvm intermediate representation for verified program transformations," in *Acm sigplan notices*, vol. 47, no. 1. ACM, 2012, pp. 427–440.

[30] A. Cortesi, "Widening operators for abstract interpretation," in *Software Engineering and Formal Methods, 2008. SEFM'08. Sixth IEEE International Conference on*. IEEE, 2008, pp. 31–40.

[31] M. Paleczny, C. Vick, and C. Click, "The Java HotSpot server compiler," in *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium*, vol. 1, no. S 1, 2001.

[32] M. Yusuf, A. El-Mahdy, and E. Rohou, "Runtime, speculative on-stack parallelization of for-loops in binary programs," *IEEE Letters of the Computer Society*, 2018.

[33] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.