ZS3: Marrying Static Analyzers and Constraint Solvers to Parallelize Loops in Managed Runtimes

Rishi Sharma*
EPFL
rishi-sharma@outlook.com

Shreyansh Kulshreshtha*
Publicis Sapient
shreyanshkuls@outlook.com

Manas Thakur
IIT Mandi
manas@iitmandi.ac.in

ABSTRACT

With the advent of multi-core systems, GPUs and FPGAs, loop parallelization has become a promising way to speed-up program execution. Correspondingly, researchers have developed techniques to parallelize loops that do not carry dependences across iterations, and/or call pure functions. However, in languages such as Java with managed runtimes, it is practically infeasible to perform complex dependence analysis during JIT compilation. In this paper, we propose ZS3, a first of its kind loop parallelizer for Java programs that marks parallelizable loops for heterogeneous architectures using TornadoVM (a Graal-based VM that supports insertion of `@Parallel` constructs for loop parallelization).

ZS3 statically performs dependence and purity analysis of Java programs in the Soot framework, to generate constraints under which a given loop can be parallelized. These constraints are fed to the Z3 theorem prover (which we have integrated with Soot) to annotate parallelizable loops with the `@Parallel` construct. We have also added runtime support in TornadoVM to use static analysis results for loop parallelization. Our evaluation over standard parallelization kernels shows that ZS3 correctly parallelizes 61.3% of manually parallelizable loops, with an efficient static analysis and a near-zero runtime overhead. ZS3 is not only the first tool that performs program-analysis based parallelization for a real-world JVM, but also the first to integrate Z3 with Soot for loop parallelization.

ACM Reference Format:
Rishi Sharma, Shreyansh Kulshreshtha, and Manas Thakur. 2022. ZS3: Marrying Static Analyzers and Constraint Solvers to Parallelize Loops in Managed Runtimes. In *Proceedings of CASCON'22*. ACM, New York, NY, USA, 8 pages.

1 INTRODUCTION

With the onset of multicore and heterogeneous systems over the last two decades, several programming languages were enriched with various ways to write concurrent programs that take advantage of the available hardware. Few languages provide built-in facilities to fork and launch multiple threads, whereas others support writing concurrent programs with extensions or libraries. At program level, writing complex computations invariably involves iterating over large data sets in loops. However, languages such as Java, though provide an built-in ability to write multithreaded programs, do not allow the programmer to directly mark loops for parallelization. One of the useful developments in this space has been the design of TornadoVM [9].

```
1 int elem_sq(int x) { return x*x; }
2 int array_sq(int[] x) {
3     for (int i = 0; i < x.length; i++) {
4         int t = elem_sq(x[i]);
5         x[i] = t; }
6     return x; }
```

Figure 1: A Java code snippet to demonstrate the analyses required for loop parallelization.

TornadoVM is a Java virtual machine (JVM) that extends OpenJDK and GraalVM [11] with a facility to parallelize `for` loops across heterogeneous architectures. It allows programmers to annotate `for` loops in Java source code with an `@Parallel` construct, preserves the annotations in Java bytecode, and then parallelizes the marked loops on the available hardware (CPUs, GPUs, FPGAs) in the virtual machine. This not only alleviates the need to write different parallelization back-ends for different architectural components, but also enriches Java with a facility to express parallelization opportunities at loop-level. However, identifying which loops can be parallelized is a non-trivial problem and involves sophisticated program analyses for even trivial programs (those involving accesses to arrays with indices being affine functions of loop variables).

As an example, consider the Java code snippet shown in Figure 1. In order to determine if the `for` loop in function `array_sq` can be parallelized without changing the semantics of the program, we need to (i) extract the array indices at statements 4 and 5; (ii) find out if the indices may access the same location in a conflicting manner across iterations; and (iii) check if the call to the function `elem_sq` may cause any side-effect. This translates to performing dependence analysis to find constraints under which a loop can be parallelized, solving the identified constraints (possibly using a constraint solver), pointer analysis to identify aliases and to resolve method calls, and an interprocedural purity analysis. However, performing multiple such precise interprocedural analyses in a JVM, where the program-analysis time directly affects the execution time of a program, is prohibitively expensive. In this paper, we present a tool named ZS3 that addresses all the challenges listed above: it performs precise analysis of Java bytecode and marks loops for parallelization, with negligible overhead during program execution.

ZS3 first performs dependence analysis over Java programs, in the Soot framework [23]. The dependence analysis generates constraints (over iteration variables) under which a given candidate loop can be parallelized. ZS3 then feeds these constraints to the

*Work performed during the author's affiliation with IIT Mandi, India.

CASCON'22, November 15 - 17 2022, Toronto, Canada

Z3 [8] theorem prover (which we have integrated with Soot), to determine if the constraints are satisfiable. In case the loop additionally makes some function calls, ZS3 also analyzes the called function(s) for purity. Finally, if there are no dependences found by Z3, and if the loop does not call any impure functions, then the given loop can be marked for parallelization. Inspired by the PYE framework [22] (which proposes ways to interface analyses across static and JIT compilers), ZS3 stores and conveys information about such parallelizable loops to our modified TornadoVM, which finally uses ZS3-results to parallelize the identified loops.

The key contribution of our approach is to enable complex dependence- and purity-analysis based loop parallelization for a VM, without spending much time in the VM. To evaluate the efficacy of this approach, we compared the precision of ZS3-identified parallelization opportunities with manually marked loops, for 14 benchmarks from a Java version of the PolyBench suite [18]. We found that ZS3 successfully marks 61.3% of parallelizable loops, and even identifies few loops that were not identified manually. We also measured the speed-ups achieved due to ZS3-identified parallel loops, and found that it is significant (1.77x, on average). In order to evaluate the trade-off between storing additional static-analysis results and performing expensive analyses in the VM, we computed the space overhead of our result files and the analysis time spent in Soot+Z3. The results show that ZS3 enables loop parallelization with a small storage overhead (10.2% of class files), negligible run-time overhead (2.85% of the execution-time of benchmarks), and that performing those analyses in the VM would have been prohibitively expensive (57.58% of the total execution-time itself).

To the best of our knowledge, ZS3 is not only the first tool that performs program-analysis based parallelization for a real-world JVM, but is also the first to integrate Soot with Z3 for loop parallelization. The purity analysis on its own is a Soot enhancement for recent Java versions and is usable as an add-on by other analyses and frameworks. Additionally, the idea of carrying static-analysis results to a Java runtime in itself is novel and presents interesting engineering challenges. This manuscript reflects that adapting existing techniques that involve multiple analyses and tools to a static+VM landscape is not easy. Hence, throughout the presentation we traverses through our exploration of the design space, and describes our solutions therein. Also, though the demonstrated target is TornadoVM, our techniques are general enough to be applied to any Java runtime that supports running parallel loops, and are adaptable to other similar languages such as C#.

2 OUR APPROACH: ZS3

Figure 2 shows how ZS3 identifies parallelizable loops and conveys this information to `TornadoVM`. Given Java bytecode, ZS3 first identifies canonical `for` loops (candidates for parallelization) and then performs dependence analysis over the loop bodies, along with purity analysis over called functions. The dependence constraints are fed to the Z3 theorem prover, which helps in classifying which loops can be parallelized without changing the underlying semantics. Finally, ZS3 generates annotations that can be supplied to the `TornadoVM` runtime and adds support to the VM (see Section 3) to parallelize loops identified by our static analysis during execution.

2.1 Identifying Canonical Loops

Java code is first compiled to Java bytecode which then runs on the JVM. Soot accepts this bytecode as input and converts it into Jimple. However, Java bytecode and Jimple do not have syntactic constructs like `for` loops. Figure 3 shows a simple `for` loop that doubles array elements in Jimple representation. As can be observed, identifying the loop, iteration variable, lower-bound, upper-bound, and increment poses a challenge in the Jimple representation.

To match the kinds of loops supported by TornadoVM, we first identify *canonical* `for` loops having the properties: (a) constant lower bound; (b) private iteration variable; (c) linear, positive and constant increment to the iteration variable; (d) condition of the form $i\{< \mid \leq \mid > \mid \geq\}u$, (e) as canonical loops, (f) iteration variable only modified in the update statement, and (g) single exit.

2.2 Variable Scoping

Local and non-local variables need to be handled separately for dependence analysis. In a parallel runtime, variables in the loop that are locally scoped are private and allocated on the stack of the running thread. On the other hand, non-local variables are shared across threads. Therefore, we must be able to differentiate local variables from their non-local counterparts. Java bytecode, however, does not contain any information about variable scopes.

Scoping information can be derived indirectly from the local variable table in the bytecode, if the source code is compiled with the `-g` flag. For each variable in the function, the local-variable-table entry in the class file contains the bytecode index of its initialization, its size, and the length in bytes for which it stays live. We directly use ASM (the front-end used by Soot) to read the class file and return the local-variable-table entry corresponding to each variable. Here, a variable is local to a loop if and only if its bytecode index and length are within the bounds of the bytecode indices of the initialization and update statements. Additionally, Jimple's temporary variables prefixed with a `$` are always considered local to the loop. For each canonical loop, we store the set of local variables in a list *localVars*.

2.3 Dependence Analysis: Scalars and Fields

Being able to discriminate between local and non-local variables in the loop, along with the fact that local variables can be accommodated on the thread's local stack on parallelization, helps parallelize more loops when compared to the conservative case. Figure 4a shows a loop writing to a local as well as a non-local variable. As the variable `c` at line 4 can be allocated on the local stack of the thread on parallelization, the iterations of the loop running in parallel will be writing to different memory locations. Therefore, this would not constitute a dependence. On the other hand, as the variable `a` is declared outside the loop, each iteration of the loop running in parallel would write to the same location. Therefore, this would indeed constitute a write-after-write (WAW) dependence. Assuming that we omit line 5 from the code, we can parallelize the loop because it would not contain any dependence. But even in this case, if we were not able to discriminate between the scopes of variables, we would not parallelize the loop. Also note that object fields can be considered special non-local variables.

To identify dependence for scalar variables and object fields, we need to check for writes to variables inside the loop. There would

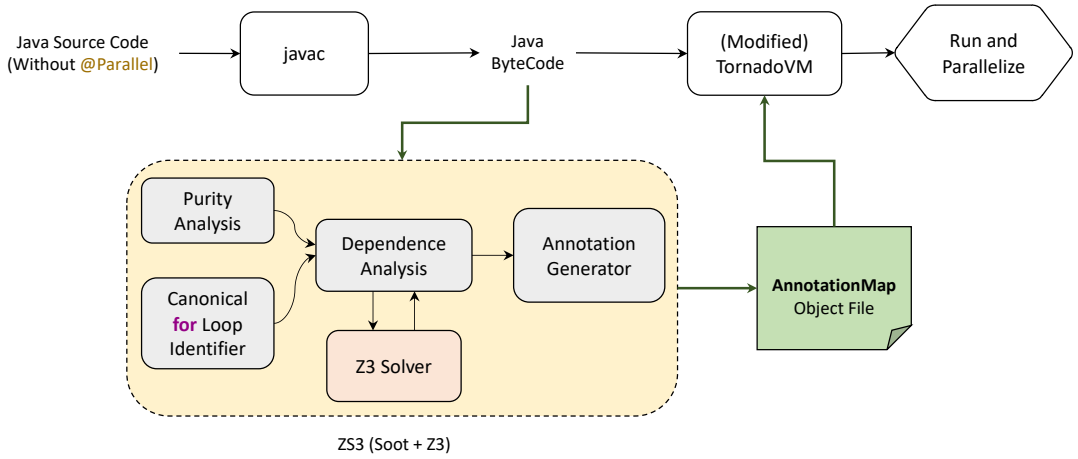

Figure 2: Architecture of ZS3.

```
1 label1: if l2 >= 5 goto label2;
2         $stack7 = $stack3[l2];
3         $stack5 = $stack3[l2];
4         $stack6 = 2.0F * $stack5;
5         $stack8 = $stack7 + $stack6;
6         $stack3[l2] = $stack8;
7         l2 = l2 + 1;
8         goto label1;
9 label2: return;
```

Figure 3: Jimple representation of a simple `for` loop.

be a WAW dependence every time a non-local variable or an object field is written-to inside the loop. Specifically, for each definition in the given loop, we check if a field reference is being written to. If yes, the loop is rejected for parallelization, else the variable being written to is looked up in *localVars*. If it is not found, ZS3 has identified a dependence and the loop is not parallelized. If no such dependence is detected after processing all the statements, the loop is passed on to the next module.

2.4 Dependence Analysis: Array References

In order to safely mark a loop as parallelizable, for each write to an element of an array, it is crucial to ensure that the same element is not accessed (either read or written) in any other iteration of the loop. Analyzing array references for dependence is difficult because the runtime values of the array indices are not available during static analysis. We propose to formulate this dependence analysis as a *satisfiability problem*. The Z3 Theorem Prover can solve the satisfiability problem if given a set of constraints, and returns a satisfying assignment, if it exists. Hence, we generate constraints based on program logic, loop iteration variable and array indices, and feed them to Z3. We pose the satisfiability problem as follows: "Is there a satisfying assignment for the program variables under the given constraints, such that the indices of the array references can be the same for two different iterations of the loop?" To make the array references easier to work with, we separate them into three sets, namely `arReads` (set of array-read references), `arWrites` (set of array-write references) and `arRefs` (`arWrites` ∪ `arReads`). The dependence analysis for arrays is then done in three phases: (i) alias analysis; (ii) constraint generation; and (iii) invoking Z3.

Only those references that point to the same array object can have a mutual dependence. Therefore, alias analysis is crucial for the precision of the dependence analysis. We generate constraints for pairs of elements from `arWrites` and `arRefs` if and only if their array objects alias. To identify aliases we use the *GeomPTA* and *Spark* pointer analyses provided by the Soot framework.

To identify if there is a dependence between a given pair of aliasing array references, the array reference indices over two different iterations must not be the same. However, this would be meaningless without capturing the program states in logic: each variable may be assigned results from complex computations. Considering all these factors, we now describe how do we generate the set of constraints; see Algorithm 1. Note that we limit the types of values on the right-hand side of the assignment statements that occur in the *def-use chain* of the indices to `IntConstant`, `Local`, `JAddExpr`, `JMulExpr`, and `JSubExpr`, denoting common integer-arithmetic expressions for forming indices in parallel programs. From now on, we use $op \in \{+, -, *\}$ to represent a supported operator.

The procedure LoopC generates the set of array dependence constraints for each pair of elements from `arWrites` and `arRefs` in a given loop l, and the set of iteration variables, lower bounds and upper bounds of all the loops of the function $iterVs$, lbs and ubs, respectively. The constraints are encoded such that if the set of constraints is *satisfiable*, there is a dependence. The procedure N returns a mapping from the set of program variables to a set of logical variables. If a variable y is non-local to the loop, there is an identity mapping. If the variable is local to the loop, y^i denotes the logical variable for the program variable y in the i^{th} iteration. The procedure StmtC recursively generates the constraints for the program logic, starting at a given statement. If the value of the variable comes from a parameter, we stop and return identity constraint *true*. If the variable is the iteration variable of another

```
1 public void f(int n) {
2     int a = 0;
3     for (int i=0; i<n; i++) {
4         int c = i;
5         a = i * 2; ...
6     } }
```

(a)

```
1 public void g(int ar[]) {
2     for (int i = 0; i < 10000; i++) {
3         int k1 = f1(i);
4         int k2 = f2(i,k1);
5         int k3 = f3(i,k2);
6         ar[k3] = k2; } }
```

(b)

```
1 void h(float[] output,int rows,int cols) {
2     for (int i = 0; i < rows; i++) {
3         for (int j = 0; j < cols; j++) {
4             output[i*rows+j] = 1.0/((i+1)+(j+1)-1);
5         }
6     } }
```

(c)

Figure 4: Examples. (a) A loop containing a write to a local variable `c` and to a non-local variable `a`. (b) A loop containing an array write. (c) Code for Hilbert computation showing a currently unparallelizable loop.

(nested) loop, we constrain it to its lower-bound and upper bound. Lines 13-14 suggest that if the value assigned is an integer constant, the variable should be constrained to that integer value. Finally, lines 15-18 recursively generate the constraints for the operands on the right hand side. Procedure DepC generates the constraints for a given pair of elements from the sets `arWrites` and `arRefs`. Lines 22-23 enforce separate iterations and that the indices of the references be equal for a dependence. This procedure is called from the procedure LoopC, which generates the entire constraint set by taking the disjunction of the dependence constraints for each pair of $w \in$ `arWrites` and $r \in$ `arRefs`. In the absence of the value of a variable during static analysis, the constraints would be weaker, and hence, easier to satisfy. Therefore, ZS3 takes a conservative approach in the absence of surety, thus maintaining soundness.

Figure 4b shows a simple loop given for dependence analysis. `k1`, `k2` and `k3` are locally-scoped, and `f1`, `f2` and `f3` can be one of the supported operators as mentioned above. Equation 1 shows the constraints generated for the array reference at line 6 with itself, for two separate iterations represented by the superscript:

$$\begin{aligned}(k3^u == f3(i^u, k2^u)) \wedge (k2^u == f2(i^u, k1^u)) \wedge \\ (k1^u == f1(i^u)) \wedge (i^u \geq 0) \wedge (i^u < 10000) \wedge \\ (k3^v == f3(i^v, k2^v)) \wedge (k2^v == f2(i^v, k1^v)) \wedge \\ (k1^v == f1(i^v)) \wedge (i^v \geq 0) \wedge (i^v < 10000) \wedge \\ (i^u \neq i^v) \wedge (k3^u == k3^v) \end{aligned} \quad (1)$$

The generated constraints are passed to the Z3 *Solver*. If the *Solver* returns *Status.UNSATISFIABLE*, the indices in the array references cannot be equal in different iterations, thus deeming that the loop is free of dependencies and can be parallelized. Otherwise, the *Solver* was able to satisfy the constraints and the loop contains some dependence; the loop is not parallelizable in such a case.

2.5 Including Calls to Pure Functions

While dependences caused by scalars, fields and array references inside the loop body are taken care of by the analyses presented in preceding sections, analyzing function invocations inside loops is not straightforward. Such an analysis requires checking the variables and references in the invoked functions for dependences in context of the loop, in some sense, extending the above-mentioned analyses to external functions. This problem can be solved by checking if the function has references to non-local objects, i.e. if the function is *pure* [3], since purity will ensure that there are no dependences among different iterations of the loop, thus enabling the parallelization of loops containing function calls.

Algorithm 1 Algorithm to generate constraints.

1: **procedure** N(y, i)
2: **if** $y \in localVars$ **then**
3: Return y^i ▷ y in iteration i
4: Return y
5: **procedure** StmtC(s, i, $iterVs$, lbs, ubs, l)
6: **if** s is $IdentityStmt$ **then** ▷ The values come from Parameters
7: Return $true$
8: **else if** $s: y \leftarrow (...)\ AND\ y \in iterVs$ **then** ▷ Other loop's $iter$
9: $lbCur \leftarrow N(y, i) \geq lbs[y]$
10: $ubCur \leftarrow N(y, i) < N(ubs[y], i)$
11: $cU \leftarrow \bigvee_{d \in Def(ubs[y], l.head)}$ StmtC(d, i, $iterVs$, lbs, ubs, l)
12: Return $lbCur\ AND\ ubCur\ AND\ cU$
13: **else if** $s: y \leftarrow k\ AND\ k$ is $IntConst$ **then**
14: Return $N(y, i) == k$
15: **else if** $s: y \leftarrow x_1\ op\ x_2\ AND\ x_1, x_2$ are scalars **then**
16: $cx_1 \leftarrow \bigvee_{d \in Def(x_1, s)}$ StmtC(d, i, lbs, ubs, l)
17: $cx_2 \leftarrow \bigvee_{d \in Def(x_2, s)}$ StmtC(d, i, lbs, ubs, l)
18: Return $N(y, i) == (N(x_1, i)\ op\ N(x_2, i)) \wedge cx_1 \wedge cx_2$
19: **procedure** DepC(w, r, l, $iterVs$, lbs, ubs)
20: $c_1 \leftarrow \bigvee_{d \in Def(w.index,\ w.stmt)}$ StmtC(d, $0, \emptyset$, $iterVs$, lbs, ubs, l)
21: $c_2 \leftarrow \bigvee_{d \in Def(r.index,\ r.stmt)}$ StmtC(d, $1, \emptyset$, $iterVs$, lbs, ubs, l)
22: $lc \leftarrow l.iter^0 != l.iter^1$ ▷ Different iterations of loop
23: $c_{dep} \leftarrow$ N($w.index$) == N($r.index$) ▷ The dependence
24: Return $c_1 \wedge c_2 \wedge lc \wedge c_{dep}$
25: **procedure** LoopC($arWrites$, $arRefs$, l, $iterVs$, lbs, ubs)
26: Return $\bigvee_{w \in arWrites} \bigvee_{r \in arRefs}$ DepC(w, r, l, $iterVs$, lbs, ubs)

Soot provides an built-in purity analysis, but in our preliminary testing we found it to be inadequate for recent Java versions. Hence we have written a new interprocedural purity analysis that uses Soot's points-to analysis [15] and call-graph construction modules. Our purity analysis module analyzes the methods called within canonical `for` loops for purity. It starts by marking methods that access static field references as impure. Calls to other methods are handled using the interprocedural part of the analysis, marking the caller impure if the callee is impure. For the parameters, any object reachable from the parameter should not be accessed inside the method, else the method would be deemed impure. We use the points-to analysis provided by Soot, which establishes a relationship between the variables and the objects they point to, to get all the objects that are transitively reachable from the objects pointed to by the parameters. We maintain a list that contains all the local variables that can point to an external object. This is done by iteratively updating the list by adding the local variables that can alias with ones already in the list, and then also adding the ones that store the fields of external objects. Whenever any object referenced by a variable from this list is read, it indicates

read-impurity, whereas if they are written then it indicates write-impurity. We store these results and use them to determine whether a function called from within a loop is pure or not. As an example, after extending the dependence analysis with purity analysis, ZS3 can successfully mark the `for` loop in Figure 1 as parallelizable.

3 RUNTIME SUPPORT

Performing the analyses shown in Section 2 inside `TornadoVM` during runtime would have simplified actually parallelizing the loop. However, performance concerns render such sophisticated analyses in Java virtual machines infeasible, which brings in the problem of communicating the analysis results from static analysis to `TornadoVM`. In this section, we describe how ZS3 conveys static analysis results to `TornadoVM`, along with the support added in `TornadoVM` to use the conveyed results for loop parallelization.

As `TornadoVM` requires the `@Parallel` annotation to be placed above parallelizable `for` loops in the Java source code and Soot works with Java class files, the straightforward solution would have been to insert these annotations in the bytecode itself. We refrained from using this approach because our analysis and parallelization are very sensitive to the bytecode indices and the local variable table generated after static compilation, and there is no one-to-one correspondence among the same between Java source code and bytecode. Even a slight change in the instructions while generating a new class file can be fatal to the program semantics.

A safer solution than the above for communicating the static analysis results of ZS3 to the `TornadoVM` runtime is to create a map $AnnotationMap : signature \rightarrow List(Annotations)$. Each annotation consists of the *start*, *length* and *slot* (in the stack frame) of the iteration variable of the parallelizable loop, looked up from the `LocalVariableTable` in the corresponding Java class file. *signature* denotes the bytecode signature of a Java method. This map is written to disk after ZS3 returns and supplied to the VM (see Figure 2). Following is an example *AnnotationMap* for the code in Figure 4b with adapted functions `f1`, `f2`, `f3`:

$$\{< DepTest : foo([I)V >: [start : 2, length : 35, slot : 1]\}$$

When TornadoVM encounters a method call, it reads the existing `@Parallel` annotations from the classfile and adds them to a method-local data structure. We modified TornadoVM to read, in addition to the annotations in the classfile, the *AnnotationMap* generated by our static analysis, and extend the data structure maintained by TornadoVM by looking up the signature of the called method in the map. This solution bypasses the need to change the classfile, and is still able to communicate the results to the runtime.

4 DISCUSSION

In this section, we highlight few subtle aspects of the design decisions made while implementing ZS3, along with a discussion on the reasoning and the alternatives.

1. Static initializers. ZS3 treats calls made to static initializers (`classinit` methods) as impure. We handle static initializers conservatively because it is difficult to predict when are they called during runtime (first reference to a class); marking them impure should not be an issue because in general they are used to assign values to static fields, which are essentially shared (global) variables anyway, thus leading to impurity for the enclosing loop.

2. Annotating class files with static-analysis results. As mentioned in Section 3, we have chosen to include static-analysis results in separate files (containing the *AnnotationMap*), for simplicity. In a production scenario, the results can be added as annotations in the class-files themselves, without loss of generality, depending on the correctness of the support to maintain local-variable offsets in Soot and ASM. We leave this as a future engineering exercise.

3. Verifying static-analysis results. For an approach that uses static-analysis results to perform VM-level optimizations without programmer intervention, one way to assess the precision and the usefulness would be to validate the results through a parallel-programming expert. In this paper, we validate the precision and correctness of ZS3-results by checking whether the loops identified as parallelizable are a subset of the manually annotated loops by TornadoVM designers, and the usefulness by measuring the speed-ups achieved with the parallelized loops (in Section 5). One could also export ZS3 as an IDE plugin that suggests parallelizable loops to programmers, who can then either accept or reject the suggestion; we mark this as an interesting future software-engineering exercise.

5 IMPLEMENTATION AND EVALUATION

We have implemented the four static components of ZS3 (highlighted in gray in Figure 2), in the Soot framework version 4.1.0, over its Jimple intermediate representation, with different modules implemented independently (thus being candidates for separately useful artefacts as well). The integration for constraint solving was done with Z3 theorem prover version 4.8.11. We have added the runtime-support code to TornadoVM version 0.11 installed along with JDK 11, and ran our experiments on an 8-core (two threads per core) 11th Gen Intel Core i5 machine with 8 GB of RAM, bundled with an Intel Iris Xe Graphics chip, running Arch Linux.

We have evaluated ZS3 on 14 benchmarks from the PolyBench suite [18], adapted to Java by the TornadoVM team [9] itself. The parallelizable loops in all these benchmarks are already annotated with `@Parallel` constructs, thus providing a baseline for evaluating the precision and correctness of the loops identified as parallelizable by ZS3. Additionally, we have also evaluated our techniques on a series of synthetic benchmarks, written specifically to test individual cases that ZS3 parallelizes, as well as to illustrate cases that pose challenges for further automatic parallelization. We plan to release our complete implementation, bundled with all the test-cases, to the community as open source. ZS3 can not only be used as a bundle tool to parallelize loops for TornadoVM, but its individual components (particularly the dependence analysis, the purity analysis, and the Soot-Z3 integration modules) can separately be used to develop various other Soot-based program analyses as well.

We now evaluate the impact of ZS3 in supporting loop parallelization for TornadoVM. In particular, the next four subsections respectively address the following four research questions:

- **RQ1.** How many of manually parallelized loops are marked as parallelizable by ZS3?
- **RQ2.** Are the overheads of static-analysis, those of storing the *AnnotationMap*, and the time spent in the VM significant?
- **RQ3.** How good are the speedups of ZS3-marked parallel loops?
- **RQ4.** What are the challenges yet to be handled by future static-analysis guided loop parallelizers?

Name	#loops	#annot	#iden	tAnalysis (s)	szClassFile (B)	szAnnotMap (B)	tParallel (ms)	tRead (ms)
Convolution2D	2	2	1	1	4527	391	1546.23	7.94
Euler	5	2	1	1	4767	497	191.03	8.22
FDTDSolver	6	6	2	4	7609	886	429.93	7.56
FlatMapExample	2	1	1	1	3815	389	658.05	20.72
GSeidel2D	2	2	0	1	4592	0	59.35	0
HilbertMatrix	2	2	1	1	3218	390	863.65	12.64
Jacobi1D	2	2	2	1	4785	739	418.06	7.06
Jacobi2D	4	4	4	1	5222	758	70.66	4.45
Mandelbrot	3	2	0	1	9186	0	13740.69	0
MatrixMul2D	3	2	2	1	5304	840	50.73	25.99
MatrixTranspose	2	2	2	1	4229	391	1031.58	25.52
Montecarlo	1	1	0	3	3615	0	418.34	0
Saxpy	1	1	1	1	3658	371	842.44	12.41
SGEMMFPGA	3	2	2	1	3835	388	1058.39	13.16
GeoMean	**2.30**	**1.96**	-	**1.22**	**4647.22**	-	**479.48**	-

Figure 5: Evaluation metrics. Out of the total number of loops (#loops) and the manually annotated `@Parallel` loops (#annot), ZS3 identified loops are shown in the #iden column. tAnalysis denotes the time taken by static analysis in seconds. szClassFile and szAnnotMap respectively denote the size of the benchmark classes and static-analysis results in bytes. tParallel and tRead respectively denote the total execution time and run-time overhead of our approach in milliseconds.

5.1 Precision and Correctness

Columns 2, 3 and 4 in Figure 5 show the number of source-code `for` loops, the number of loops manually annotated with `@Parallel` in the TornadoVM benchmarks, and the number of loops identified as parallelizable by ZS3, respectively. Each of the benchmarks contains at least one parallelizable loop, with the maximum number of parallelizable loops being 6 (for `FDTDSolver`). Out of the 38 loops across the 14 kernels, 31 of them were found to contain the `@Parallel` annotation, whereas ZS3 successfully identified 19 of them to be parallelizable. Thus, across all the benchmarks, ZS3 is able to successfully identify 61.3% of the manually parallelized loops. Recognizing that this identification is done statically without any manual intervention (and as shown later, with negligible runtime overhead), we believe that the precision is reasonable.

To validate the correctness of the loops identified by ZS3, we checked whether the identified loops are a subset of the loops manually annotated with `@Parallel`, and we found that this was indeed the case. On synthetic testcases designed to test individual features of our analysis, we also found few loops marked as parallelizable by ZS3 that were not so straightforward to be parallelized manually. This supports our hypothesis that a manual identification of parallelizable constructs is error-prone and imprecise, and consequently encourages the use of ZS3 for real-world programs and runtimes.

5.2 Time and Space Overheads

ZS3 makes use of sophisticated interprocedural points-to and call-graph construction analyses, computed dependence information within loops, purity information about functions, and invokes Z3 from within to check satisfiability of dependence constraints. Performing all of this during runtime (in a Java VM) would not only be a tremendous engineering effort, but might also be practically infeasible due to the associated analysis cost. On the other hand, though performing analyses statically takes care of the complexity and practicality to a great extent, our approach incurs additional overhead in terms of conveying the *AnnotationMap* to, and adding runtime support to read the same in, TornadoVM. We assess the scale and impact of these overheads next.

Column 5 in Figure 5 shows the total analysis time spent by ZS3, for all the benchmarks. This includes the time spent by Soot to construct control-flow- and call-graphs, as well as the time spent by Z3 in solving the ZS3-generated dependence constraints. We note that the time spent across different benchmarks varies between 1 and 4 seconds, which is of the same order for different benchmarks due to the similarity in their size, but we expect it to increase proportionally with the size of the benchmark. Nevertheless, the total analysis time is reasonable to be incurred statically (i.e. offline).

Columns 6 and 7 respectively show the size (in bytes) of the classfiles of various benchmark applications and the size of the ZS3-generated result files. We observe that the extra space incurred by our approach to convey precise static-analysis results to TornadoVM is very small (on an average 500 bytes), which is just 10.2% of the overall class-file size. This denotes that the results computed by our static analyses are small enough to be conveyed to the VM without much overhead. As explained in Section 3, we achieve this by storing the results in terms of mostly integer values (storing information in terms of bytecode indices in the class files).

Column 8 shows the total execution time (in milliseconds) of each (parallelized) program under consideration. Similarly, column 9 shows the time spent by our modified TornadoVM for reading the *AnnotationMap* and using the results to mark loops as parallelizable. As can be noted, the runtime overhead of our approach is just a few milliseconds, which is negligible compared to the amount of time that would have been needed for actually performing sophisticated program analyses in the JVM, as well as to the total execution time. Also note that ZS3 enables interprocedural-analysis based loop parallelization irrespective of the tiered VM component translating the program (interpreter or JIT compiler(s)), whereas even an imprecise version of such analyses could have been performed only if the given method was picked up by a JIT compiler.

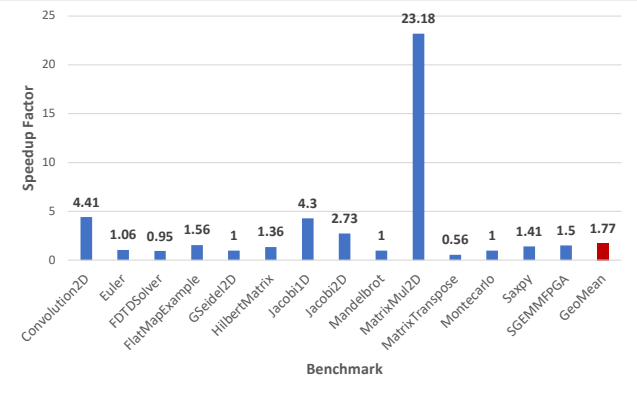

Figure 6: Speed-ups in TornadoVM with ZS3.

5.3 Achieved Speed-Ups

The previous sections assert the precision and efficiency of ZS3 for parallelizing loops. We now assess the impact of loop parallelization itself, by comparing the execution times of the sequential and the ZS3-parallelized versions of various benchmarks, on TornadoVM.

Figure 6 shows the speed-ups achieved by the parallel versions of the benchmarks under consideration. We note that the speed-ups go up to ~23x (for `MatrixMul2D`), and on an average stand at 1.77x. We also noted slowdowns on few of the benchmarks, specifically `FDTDSolver` and `MatrixTranspose`, and suggest two ways to address the same. Firstly, the speed-ups may vary if the programs are executed on larger datasets and/or on higher-end systems. Second and more importantly, the slow-downs indicate that not all of the parallelizable loops are good candidates for parallelization; overheads with respect to communication (particularly TornadoVM's target being heterogeneous systems consisting of GPUs and FPGAs that have high communication overheads, apart from CPUs) is an important factor. We have also observed that the imprecision affects mostly the outer loops (see Section 5.4), thereby increasing the overall overhead of parallelization when parallelizing inner loops. We envisage that future studies should adapt the approaches of Surendran et al. [20] to filter out loops that may not be good candidates for parallelization, for heterogeneous systems.

5.4 Challenges Towards Further Parallelization

The loops that are not parallelized by ZS3 but can be labelled as parallelizable manually may be non-exhaustively categorized into the following two categories.

1. Unknown upper bounds. Multiple loops in our evaluation set are not parallelizable because of the lack of the value of upper bound of the iteration variable during static analysis. As the upper bounds of loops are generally runtime values, the constraints given to the Z3 solver are weaker than they would be at run time. Figure 4c is an example of such a loop; as the values of *rows* and *cols* are missing during the analysis, the solver is able to find a dependence when $j >= rows$, for $i_0 = 0$ and $i_1 = 1$. Hence, the outer loop is not parallelized. In our evaluation, we found that about 83% of imprecision of ZS3 was due to unknown upper bounds. In future, we plan to explore how can we generate results in terms of conditions on loop bounds, and how can they be efficiently resolved during JIT compilation in a JVM, similar to the PYE framework [22]. Approaches such as the PYE framework [22] handle both these challenges by generating results conditional on the library methods, and the same can be incorporated, if deemed suitable for precision and scalability, by integrating ZS3 into the same.

2. Library function calls. Even though we account for purity of function calls during analysis in ZS3, library functions are marked (conservatively) as impure. This means not parallelizing all such loops which even contain functions like `sqrt`; ZS3 failed to parallelize `Montecarlo` because of this reason. We treated library functions as impure due to two reasons: First and more important, in a real-world scenario, the JDK installation on the target machine may be different from that available for static analysis, which may lead to invalidity of statically generated results. Second, including library calls in the analysis blows up the size of Soot's call graph (due to its imprecision), thus making the analysis unsuitable for general-purpose machines with moderate compute capabilities.

6 RELATED WORK

Automating loop parallelization for achieving performance on multi-core (and recently, heterogeneous) systems has been studied for long [2], and its optimality for even simple loops has been proved to be undecidable [10]. In this section, we primarily focus on related works that propose significant advancements in performing (loop) parallelization using dependence and/or purity analysis.

The precision of finding and solving dependence constraints directly affects the amount of parallelization, and hence several dependence analysis algorithms have been proposed. Partition-based merging implemented in Parascope [7] works by separating arrays into separable minimally coupled groups. Merging direction vectors [1] is a common dependence analysis algorithm used by tools such as Automatic Code Parallelizer [16]. Symbolic test and Banerjee-GCD test [4] are used to detect data dependence among array references by assuming the loop in normal form and the loop indices to be affine functions. TornadoVM in itself does not use dependence analyses, due to the overhead of performing these checks during program execution. Our dependence analysis is based on Z3, does not require indices to be affine (is able to solve non-linear arithmetic), and most interestingly is performed statically (that is, without incurring any analysis or satisfaction overheads during program execution in the JVM).

Constraint solvers such as Z3 have been extensively used for verification of programs. Bounded model checking [5] is a popular way of finding bugs in programs. Satisfiability modulo theory has been extended for verification of higher-order programs [6], and multi-threaded program verification [12]. Constraint satisfiability based techniques have also been used for quantification of information flow in imperative programs using a SAT-based QIF [14]. On the other hand, Pugh and Wonnacott [19] were among the first to propose the usage of constraint-solving for dependence analysis. Inspired by these prior works, in this work, we feed the constraints identified by our dependence analysis written in Soot to the Z3 solver, and marked loops for parallelization where the dependence constraints are satisfied. To the best of our knowledge, ours is the first approach that integrates Soot with Z3 for this purpose.

Purity analysis [3] identifies side-effect free functions, and is imperative for parallelization of otherwise dependence-free loops consisting of calls to functions that may be impure. Süß et al. propose a C extension [21] that marks pure function calls to support parallelization of polyhedral loops. ZS3 implements a stand-alone purity analysis component (which works for recent versions of Java, unlike prior implementations in Soot), and hence naturally supports programs containing function calls inside Java `for` loops.

There have been few works that discuss loop parallelization for managed runtimes. Zhao et al. [25] support our claim that performing analyses for parallelization during runtime is expensive, and use a conservative GCD test as compared to our constraint-solver based dependence analysis [24] for the Jikes RVM. Similarly, few prior works have proposed dependence analysis based loop parallelization with hybrid static+dynamic strategies. Oancea and Rauchwerger [17] use runtime information to improve the performance of static dependence analysis for FORTRAN. Recently, Jacob et al. [13] used staged dependence analysis while parallelizing Python loops on GPUs, to determine loop bounds and variable types that cannot be determined statically (Python being a dynamically typed language). Thakur and Nandivada [22], though for a different set of analyses, propose a static+JIT approach that statically encodes dependencies between Java application and libraries and resolves them during JIT compilation. Our approach facilitates loop parallelization on a dynamic Java runtime, by offloading complex program analyses to static time, and can be extended using such hybrid strategies to improve the precision further.

7 CONCLUSION

Loop parallelization, though one of the most promising ways to speed-up programs on multicore and heterogeneous systems, requires performing several expensive analyses for automation. For a language like Java, where most of the program analysis happens during just-in-time compilation in a VM (thus affecting the execution-time of programs directly), performing such analyses for loop parallelization not only presents several challenges in terms of integrating program analyses with constraint solvers, but may often also be prohibitively expensive. In this paper, we proposed an approach that solves this problem by performing the required analyses statically, and conveying the obtained results to a recent JVM that parallelizes loops for heterogeneous architectures. Our solution involved generating dependence constraints from Java bytecode, feeding them to a constraint solver, supporting calls to pure functions, generating results in a form that is valid in the Java runtime, and modifying the VM to support static-analysis guided parallelization. Our exposition describes the design decisions and implementation challenges along with our novel solutions in detail, and our tool ZS3 is composed of several modules that can additionally be used for performing more such analyses in future.

REFERENCES

[1] 1993. Automatic Program Parallelization. *Proceedings of the Institute of Radio Engineers* 81, 2 (Feb. 1993), 211–243. https://doi.org/10.1109/5.214548

[2] A. Aiken and A. Nicolau. 1988. Optimal Loop Parallelization. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '88)*. Association for Computing Machinery, New York, NY, USA, 308–317. https://doi.org/10.1145/53990.54021

[3] Ru Alcianu and Martin Rinard. 2004. A Combined Pointer and Purity Analysis for Java Programs. (06 2004).

[4] Utpal K. Banerjee. 1988. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, USA.

[5] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, W. Rance Cleaveland (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 193–207.

[6] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. 2013. Higher-order Program Verification as Satisfiability Modulo Theories with Algebraic Datatypes. *CoRR* abs/1306.5264 (2013). arXiv:1306.5264 http://arxiv.org/abs/1306.5264

[7] K.D. Cooper, M.W. Hall, R.T. Hood, K. Kennedy, K.S. McKinley, J.M. Mellor-Crummey, L. Torczon, and S.K. Warren. 1993. The ParaScope parallel programming environment. *Proc. IEEE* 81, 2 (1993), 244–263. https://doi.org/10.1109/5.214549

[8] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[9] Juan Fumero. 2020. TornadoVM: Accelerating Java with GPUs and FPGAs. (2020). https://www.infoq.com/articles/tornadovm-java-gpu-fpga/

[10] F. Gasperoni, U. Schwiegelshohn, and K. Ebcioğlu. 1989. On Optimal Loop Parallelization. In *Proceedings of the 22nd Annual Workshop on Microprogramming and Microarchitecture* (Dublin, Ireland) *(MICRO 22)*. Association for Computing Machinery, New York, NY, USA, 141–147. https://doi.org/10.1145/75362.75411

[11] Graal. 2019. OpenJDK Graal. http://openjdk.java.net/projects/graal/.

[12] Fei He, Zhihang Sun, and Hongyu Fan. 2021. Satisfiability modulo Ordering Consistency Theory for Multi-Threaded Program Verification. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, NY, USA, 1264–1279. https://doi.org/10.1145/3453483.3454108

[13] Dejice Jacob, Phil Trinder, and Jeremy Singer. 2019. Python Programmers Have GPUs Too: Automatic Python Loop Parallelization with Staged Dependence Analysis. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages* (Athens, Greece) *(DLS 2019)*. Association for Computing Machinery, New York, NY, USA, 42–54. https://doi.org/10.1145/3359619.3359743

[14] Vladimir Klebanov, Norbert Manthey, and Christian Muise. 2013. SAT-Based Analysis and Quantification of Information Flow in Programs. In *Quantitative Evaluation of Systems*, Kaustubh Joshi, Markus Siegle, Mariëlle Stoelinga, and Pedro R. D'Argenio (Eds.). Springer Berlin Heidelberg, Heidelberg, 177–192.

[15] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction* (Warsaw, Poland) *(CC'03)*. Springer-Verlag, Berlin, Heidelberg, 153–169. http://dl.acm.org/citation.cfm?id=1765931.1765948

[16] Manju Mathews and Jisha P Abraham. 2016. Automatic Code Parallelization with OpenMP task constructs. In *2016 International Conference on Information Science (ICIS)*. 233–238. https://doi.org/10.1109/INFOSCI.2016.7845333

[17] Cosmin E. Oancea and Lawrence Rauchwerger. 2012. Logical Inference Techniques for Loop Parallelization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. ACM, NY, USA, 509–520. https://doi.org/10.1145/2254064.2254124

[18] Louis-Noel Pouchet and Tomofumi Yuki. 2016. PolyBench. http://polybench.sourceforge.net/.

[19] William Pugh and David Wonnacott. 1998. Constraint-Based Array Dependence Analysis. *ACM Trans. Program. Lang. Syst.* 20, 3 (may 1998), 635–678. https://doi.org/10.1145/291889.291900

[20] Rishi Surendran et al. 2016. Automatic Parallelization of Pure Method Calls via Conditional Future Synthesis. *SIGPLAN Not.* 51, 10 (Oct. 2016), 19 pages. https://doi.org/10.1145/3022671.2984035

[21] Tim Süß, Lars Nagel, Marc-André Vef, André Brinkmann, Dustin Feld, and Thomas Soddemann. 2017. Pure Functions in C: A Small Keyword for Automatic Parallelization. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 552–556. https://doi.org/10.1109/CLUSTER.2017.32

[22] Manas Thakur and V. Krishna Nandivada. 2019. PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs. *ACM Trans. Program. Lang. Syst.* 41, 3, Article 16 (July 2019), 37 pages. https://doi.org/10.1145/3337794

[23] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press.

[24] J. Zhao, C. Kirkham, and I. Rogers. 2006. Lazy Interprocedural Analysis for Dynamic Loop Parallelization. In *Proceedings of the Workshop on New Horizons in Compilers (NHC '06)*.

[25] Jisheng Zhao, I. Rogers, C. Kirkham, and I. Watson. 2005. Loop Parallelisation for the Jikes RVM. In *Sixth International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT'05)*. 35–39. https://doi.org/10.1109/PDCAT.2005.164