International Journal of Parallel Programming manuscript No.
(will be inserted by the editor)

PolyJIT: Polyhedral Optimization Just in Time

Andreas Simbürger · Sven Apel · Armin Größlinger · Christian Lengauer

the date of receipt and acceptance should be inserted later

Abstract While polyhedral optimization appeared in mainstream compilers during the past decade, its profitability in scenarios outside its classic domain of linear-algebra programs has remained in question. Recent implementations, such as the LLVM plugin Polly, produce promising speedups, but the restriction to affine loop programs with control flow known at compile time continues to be a limiting factor. PolyJIT combines polyhedral optimization with multi-versioning at run time, at which one has access to knowledge enabling polyhedral optimization, which is not available at compile time. By means of a fully-fledged implementation of a light-weight just-in-time (JIT) compiler and a series of experiments on a selection of real-world and benchmark programs, we demonstrate that the consideration of run-time knowledge helps in tackling compile-time violations of affinity and, consequently, offers new opportunities of optimization at run time.

Keywords JIT compilation · loop parallelization · polyhedron model

Acknowledgements All four authors received finanicial support by the Deutsche Forschungsgemeinschaft (DFG). The respective projects are PolyJIT (LE 912/14), SafeSPL (AP 206/4) and SafeSPL++ (AP 206/6).

Andreas Simbürger
University of Passau
E-mail: andreas.simbuerger@uni-passau.de

Sven Apel
University of Passau
E-mail: sven.apel@uni-passau.de

Armin Größlinger
University of Passau
E-mail: armin.groesslinger@uni-passau.de

Christian Lengauer
University of Passau
E-mail: christian.lengauer@uni-passau.de

1 Introduction

Automatic code optimization is becoming an increasingly challenging task. The variety and complexity of optimization techniques have increased with the introduction of hyperthreading, SIMD extensions, multicore processors, general-purpose computing on graphics hardware, low-power computing, etc. Programmers and compiler implementers are being confronted with the architectural variety of platforms and the resulting challenges of performance portability. Just setting a few compiler optimization flags no longer assures high performance.

To tackle this problem, a wide variety of approaches to the *automatic* optimization of program code has been proposed [2]. One model-based approach is *polyhedral optimization*. The *polyhedron model* [12], in which the steps of a loop nest are being spread across a polyhedral space, serves to optimize loop programs automatically (e.g., via parallelization or data localization) by applying algebraic transformations. Its main benefit is that all loop transformations can be discovered via integer linear programming at a cost that is independent of the problem size. Among the many implementations of polyhedral optimizers similar to PLuTo [4,5], there are two projects that target mainstream compilers: Polly [14,16] and Graphite [34].

Full automation comes at the price of limitations on the structure of the programs that can be optimized. In particular, to be analyzable at compile time, loop bounds and memory-access functions are limited to affine linear expressions. For example, array access functions like `A[3*i+1]` are allowed, whereas access functions like `A[i*i]` are disallowed. In the past, a number of extensions of the polyhedron model have been proposed to overcome affine linearity in certain cases, but the limitations are still severe [17]. Another promising approach is to apply polyhedral optimization not only at compile time but also at run time. At run time, more is known about the actual structure of the loops, which opens up new opportunities for optimization.

While promising, it is still unclear whether leveraging run-time information for polyhedral optimization actually pays off. We report on an empirical study of the usefulness of just-in-time polyhedral optimization of real-world applications and benchmark programs. Based on results of previous empirical work on the potential of polyhedral optimization [26], we have been developing PolyJIT, an extension to the LLVM compiler framework, which implements a light-weight, platform-independent JIT compiler. PolyJIT overcomes classic compile-time limitations of polyhedral optimization by delaying parts of the optimization process to run time. However, JIT optimization comes at the price of increased program execution time. While classic JIT compilers (e.g., the Java HotSpot VM [24]) operate on an intermediate representation of the whole program code, PolyJIT strives to minimize the run-time overhead by concentrating only on an intermediate representation of loops that are likely to profit from polyhedral optimization at run time.

Our empirical study of the merits of PolyJIT relies on 53 programs amenable to polyhedral optimization covering a variety of 12 application

domains, including multimedia, compression, or compilation as well as several well-known community benchmark suites. We found that PolyJIT is able to optimize the performance of 29 programs, with run-time speedups of up to 57.69, when considering individual code regions, and up to 13.03, when considering entire programs. Where polyhedral optimization is less applicable, PolyJIT is able to avoid slowing down 38 programs although, as of yet, it has not been able to provide a beneficial polyhedral transformation.

In summary, our contribution is three-fold:

- PolyJIT, a light-weight polyhedral JIT compiler that leverages run-time information to optimize preselected loop nests at run time,
- an empirical study of the state and benefit of polyhedral just-in-time compilation on real-world programs and benchmarks,
- an analysis of PolyJIT's overhead and its influence on the total benefit of polyhedral optimization at run time.

PolyJIT, our analysis engine, all sample programs, and all results are available at the project's Web site: https://www.infosun.fim.uni-passau.de/cl/PolyJIT/IJPP

2 The Polyhedron Model

The polyhedron model represents programs—in particular, loops—in an algebraic form as polyhedra, to make them amenable to algebraic transformations. A major use case of the model and the corresponding transformations is to parallelize loops; others are cache-locality and memory-usage optimization (see Section 5). The main advantage of the polyhedron model is that loop transformations can be applied fully automatically and independently of the problem size. The price of full automation and feasible complexity is that not all kinds of programs can be processed.

The polyhedral optimization of a program consists of two steps: (1) detect the loops of a program that can be represented in the model, called Static Control Part (SCoP) [3], and (2) apply the actual transformations to optimize the program (loop parallelization, etc.). In the following subsections, we introduce the basic model and its limitations. A comprehensive survey of the polyhedron model can be found elsewhere [12].

2.1 Basic Model

In the polyhedron model, a loop program consists of a number of statements. Each statement has an associated iteration domain (which is defined by the loops surrounding the statement) and a schedule (which determines the execution order of the statement instances). Consider Figure 1, which gives an example of a SCoP, to cover briefly the key concepts of the polyhedron model.

Each program statement S comes with its own iteration domain D_S, which is a subset of $\mathbb{Z}^n$. Each point $\mathbf{i} \in D_S$ in this domain represents a statement instance $(S; \mathbf{i})$; that is, statement S is executed once for every such $\mathbf{i}$. Our

```
for (int i=0; i<=n; ++i) {
    for (int j=i; j<=n; ++j) {
        if (i >= n-j) {
S:          A[i+n][j+i] = B[n+2*i-1][j];
T:          B[i+n][j-i] = A[n-2*i+1][j];
        }
    }
}
```

Listing 1: A static control part (SCoP)

example has two statements: S and T (Lines 4 and 5). They are surrounded by two loops and an `if` statement and, therefore, share the iteration domain

$$D_S = D_T = \{[i,j] : 0 \leq i \leq n \wedge \ i \leq j \leq n \ \wedge \ n - j \leq i\},$$

where n denotes a parameter that is constant during the execution of S and T but unknown at compile time. We call this a *structure parameter*. Note that the control flow is known at compile time because the predicate and the loop bounds are affine expressions.

Each statement may contain memory *accesses* to arrays (the aggregate data structure on which the polyhedron model concentrates); these are represented by relations between the domain of the statement and the indices of the array cells accessed. In our example, both statements perform a read access and, subsequently, a write access. The read and write accesses are summarized in the following relations R and W, respectively:

$$R = \{S[i,j] \mapsto B[n + 2 \cdot i - 1, j]; T[i,j] \mapsto A[n - 2 \cdot i + 1, j]\}$$
$$W = \{S[i,j] \mapsto A[i + n, j + i]; T[i,j] \mapsto B[i + n, j - i]\}$$

R and W map iteration (i,j) of S and T to the accessed elements of A and B, respectively.

Transformations of the program must not violate the partial order imposed by the program's semantics. Therefore, the most important computational task, when using the polyhedron model for program transformations, is to compute the dependences between statement instances. Statement instances $(T; \mathbf{j})$ and $(S; \mathbf{i})$ are members of the dependence relation iff there are accesses of the same memory cell in S and T (for the given values of $\mathbf{i}$ and $\mathbf{j}$).

Determining dependences is undecidable in the general case, because this would require the solution of arbitrary systems of equations derived from the accesses (cf. the unsolvability of Hilbert's 10th problem [9]). However, dependences can be computed when iteration domains and accesses are defined by affine expressions (i.e., all constraints can be written in the form $M\mathbf{i} \geq \mathbf{b}$ for $M \in \mathbb{Z}^{k \times n}$, $\mathbf{b} \in \mathbb{Z}^k$).[1] Note that some dimensions of $\mathbf{i}$ can depend on structure parameters, which allows parameters to occur additively (*weak* parametrization; Listing 2a) in all constraints, schedules, and memory accesses, but not multiplicatively (*strong* parametrization; Listing 2b).

[1] Geometrically, these objects are ($\mathbb{Z}$-)polyhedra.

2.2 Limitations

The restriction to affine expressions implies that non-affine conditions and recursive control flow cannot benefit from polyhedral optimization. Other consequences of the restriction are that the only aggregate data structure allowed is the array (scalars can be represented as zero-dimensional arrays) and the only statement type allowed in the loop body is the assignment. Calls of functions with side-effects inside a loop body are not supported by the basic model, because the memory access behavior and the control flow are hidden inside the body of the function called.

Parameters at run time. The basic polyhedron model requires all loop bounds and memory accesses to be linearly affine (Listing 2a, weak parametrization). Non-linearity introduced by parameters (strong parametrization) cannot be handled in the basic model.

```
for (int i=0; i<=n; i++) {
    A[i+n] = ...;
    ... = A[i-1+n];
}
```

(a) linear memory access (weak parametrization)

```
for (int i=0; i<=n; i++) {
    A[m*i+n] = ...;
    ... = A[m*(i-1)+n];
}
```

(b) non-linear memory access (strong parametrization)

Listing 2: Linear vs. non-linear memory access. The expression $i - n + 1$ can be handled in the basic polyhedron model. The expression $m \cdot (i - 1) + n$ cannot be handled in the basic polyhedron model, due to the multiplicative parameter m.

In Listing 2b, parameter `m`, although loop-invariant, is multiplied with the value of iteration variable `i`, forming a non-linear expression. The value of parameter `m` is known at run time, though. By substituting the value for the parameter name, the loop nest complies with the polyhedron model.

Run-time information about parameters is not limited to constant parameter values. If structure parameter `m` adopts a limited number of values, one can provide a specialized loop code for each value. In the worst case, polyhedral analysis and optimization have to be performed every time the loop nest is reached by the control flow of the program.

Aliasing known at run time. As soon as we consider input languages that support pointers, we have to deal with the possibility of aliasing. Contemporary alias analysis can provide only a conservative approximation, leading to a so-called *may-alias*, that is, an alias whose existence must be assumed but is uncertain. There are two alternative ways of dealing with a may-alias: (1) one postulates the corresponding dependence at compile time or (2) one tests for the alias at run time and respects its dependence conditionally.

In PolyJIT, we take advantage of the fact that the actual aliases are revealed at run time, so we can optimize SCoPs that give rise to aliasing behavior unknown at compile time.

3 PolyJIT

This section introduces our polyhedral loop parallelizer and JIT compiler PolyJIT. PolyJIT is implemented on top of the LLVM compiler infrastructure [22] and its compile-time polyhedral optimizer Polly [16].

3.1 Goals

The focus of PolyJIT is on the automatic loop parallelization of general-purpose programs—general-purpose, because they need not necessarily belong to the classic application domains of the polyhedron model, such as linear algebra kernels, which spend most of their execution time in a single code region. Such a region is called *hot*.

Since the workload of general-purpose programs may possess a considerable number of distinct hot regions, PolyJIT needs to be able to provide efficient dynamic coverage of SCoPs that contribute comparitively little to the total execution time.

Traditional JIT compilers (e.g., the Java Hotspot VM or the DalvikVM [1]) use an interpreter for the whole program to provide quick startup and an iterative optimization of the code that is deemed hot. Polyhedral transformations incur a considerably higher compilation overhead at run time. Therefore, when we trigger JIT compilation, we want to provide the optimized performance at once, without the intermediate stage provided by an interpreter. To avoid that the potential speedup is eaten up by the effort of its generation, PolyJIT must be able to (1) minimize its run-time overhead and (2) maximize its effective code coverage.

3.2 PolyJIT Phases

PolyJIT's two-phase design is similar to that of other frameworks that optimize kernel functions at run time (e.g., OpenCL [30] or CUDA [8]). The following algorithms are not restricted to LLVM but, in their explanation, we follow LLVM terminology.

The *compilation phase* detects all loops that are amenable to run-time optimization and prepares the binary for JIT support. The *execution phase* picks up the information collected in the compilation phase and invokes the JIT compilation of the optimized dynamic SCoPs during program execution.

3.3 Compilation Phase

The compilation phase prepares the binary for a JIT execution with PolyJIT. The steps performed are listed in Algorithm 1.

Input: a module M
Output: a set of modules, each of which carries an extracted SCoP function.
Variables: a set $params \in uses(scops(M))$,
a set $candidates \in scops(M) \times uses(scops(M))$,
a set $worklist \subseteq regions(M)$

1: $modules \leftarrow \emptyset$
2: **for each** $func \in M$ **do**
3: $candidates \leftarrow \emptyset$
4: $worklist \leftarrow \{topLevelRegion(func)\}$
5: **while** $worklist \neq \emptyset$ **do**
6: $region \leftarrow pop(worklist)$
7: **if** $\neg isValid(region)$ **then**
8: $worklist \leftarrow worklist \cup children(region)$
9: **else**
10: $candidates \leftarrow candidates \cup \{(region, uses(region))\}$
11: **for each** $(region, params) \in candidates$ **do**
12: $newFunc \leftarrow$ ExtractFunc$(region, params)$
13: $newModule \leftarrow$ GeneratePrototype$(newFunc)$
14: $prototype \leftarrow$ SerializePrototype$(newModule)$
15: $modules \leftarrow modules \cup$ InjectPolyJIT$(M, prototype, func)$
16: **return** $modules$

Algorithm 1: PrepareDynamicSCoPs(*Module*)

The preparation of dynamic SCoPs is divided into two parts. First, we identify all suitable code regions via a dynamic SCoP detection. Second, each detected region is encapsulated as a separate module, which serves as a prototype for multi-versioning, and stored in the final binary in serialized form.

Dynamic SCoP detection The detection of dynamic SCoPs is performed on each function of the program. Each function may contain an arbitrary number of SCoP candidates. A SCoP represents a region of LLVM IR code. A (simple) region is a group of basic blocks that share a single-entry edge and a single-exit edge in the function's control-flow graph [35]. The detection of dynamic SCoPs performs a recursive-descent traversal of LLVM's region tree, similarly to Polly's detection procedure. Algorithm 1 shows an iterative version of the region tree walk (Lines 3–10). However, the requirement of linear affinity of array access functions, conditions, and loop bounds must be relaxed to admit expressions of the form $n \cdot i + c$, where n and c are structure parameters and i is a loop

variable (cf. Section 2). Accesses to multi-dimensional arrays are represented as linearized pointer accesses in LLVM IR (e.g., `int A[n][m]; A[i][j];` becomes the non-affine access `int *A; *(A + n*i + j)`). All information about the multi-dimensional properties of array A is lost. Therefore, PolyJIT ignores Polly's optimistic delinearization of multi-dimensional accesses and subjects them instead to a dedicated treatment at run time.

Structure parameters that appear as a factor in a non-affine expression must be replaced with their run-time values later on. The dynamic SCoP detection collects all of these parameters and returns them together with a valid dynamic SCoP candidate.

Input: a region R, a set $params \in uses(R)$
Output: a function that contains R
Variables: a set $usedInside \in uses(R)$, a set $usedOutside \in uses(R)$

1: $(usedInside, usedOutside) \leftarrow inouts(R)$
2: $funcArguments \leftarrow usedInside \cup usedOutside$
3: **for each** $argument \in funcArguments$ **do**
4: **for each** $p \in params$ **do**
5: **if** $p \in exprTree(argument)$ **then** $mark(argument)$
6: $func \leftarrow emptyFunction(funcArguments)$
7: **return** $extract(R, func)$

Algorithm 2: ExtractFunction($R, params$)

SCoP-to-function extraction. A successfully detected dynamic SCoP candidate must be encapsulated as a separate function, as specified by Algorithm 2, before it can be optimized by PolyJIT at run time. The dynamic SCoP detection provides a `Region` and a set of `params` (Algorithm 1, Line 11). The result of the ExtractFunction is the newly encapsulated function that contains the dynamic SCoP.

The code region of a dynamic SCoP candidate can make use of values that are defined or used outside. The extraction collects these values and makes them arguments of the new function. All values are passed by reference to the function to preserve the semantics of the original code region. The function arguments are annotated as to whether or not they are suitable for multi-versioning at run time (Algorithm 2, Line 5). This includes all structural parameters, not only those that are required for a dynamic SCoP to become a standard affine one.

It is possible that structural parameters are used inside the SCoP in different contexts (e.g., as 64-bit or 32-bit integer). In such cases, cast instructions convert between the representations. These can be hoisted out of the SCoP and, therefore, would appear as two separate structural parameters in the

extracted function signature. PolyJIT tries to avoid this and enlarges the code region to the original definition of the structural parameter.

Input: a function F,
Output: a prototype module that contains a clone of F

1: $module \leftarrow getModule(F)$
2: $newModule \leftarrow copyEmpty(module(F))$
3: $newFunc \leftarrow createEmpty(newModule, type(F))$
4: **for each** $globalValue \in globals(module) \cap uses(F)$ **do**
5: $\quad globals(newModule) \leftarrow weakLinkage(copy(globalValue))$
6: **return** $cloneFunctionInto(F, newFunc)$

Algorithm 3: GeneratePrototype(F)

Prototype extraction. The extraction of functions for the main module is specified by Algorithm 3. During the execution of the main program, PolyJIT reconstructs the function from its serialized image and uses it as prototype for new function variants. To this end, the extracted function must be converted to a separate LLVM IR module, which can be read from memory later during the main program's execution. The function-to-module conversion requires the extracted function to be cloned into an empty LLVM IR module. However, in the presence of accesses to global values (i.e., global variable uses and calls of other functions), we must clone the global variable definitions, as well, and change the linkage type, such that the linker guarantees that there is only one copy of the symbol at run time (Line 4–5). After successful extraction, we can serialize the new module and store it as a global variable in the final binary.

PolyJIT *callback injection.* The final step of PolyJIT's compilation phase provides a stub that redirects calls of the extracted function to PolyJIT's run-time library. The prototype extraction generates a call of the extracted function in place of the original dynamic SCoP.

This function call must be changed to a call to PolyJIT's run-time library. Our generated stub has the same function signature as the extracted function. Listing 3 shows a code example of a function stub. The original `ScopFunction` takes five parameters and has the parameters `n` and `k` marked for multi-versioning with `__polyjit`. We store references to all parameter values in an array (`params`) on the stub function's stack and invoke PolyJIT's run-time environment. Other than the original function's parameter values, function stub redirection enables passing context information (e.g., call frequencies or execution times into PolyJIT's run-time environment without additional overhead).

```
void ScopFunction(int __polyjit n, int A[2048],
                  int __polyjit k, float alpha,
                  int B[2048]) {
    void *params[5];
    params[0] = &n;
    params[1] = A;
    params[2] = &k;
    params[3] = &alpha;
    params[4] = B;
    pjit_main(&ScopFunctionPrototype, 5, params);
}
```

Listing 3: Example of a function stub for PolyJIT in C-like syntax.

3.4 Execution Phase

During the execution phase, PolyJIT will be activated as soon as one of the JIT callbacks (injected in the compilation phase by Algorithm 3) gets issued. Once activated, PolyJIT will execute its run-time phase as specified by Algorithm 4. First, it deserializes the appropriate dynamic SCoP into memory (Lines 1–2). From there, the input parameters of the SCoP will be analyzed. Since the LLVM IR representation of the requested dynamic SCoP is available at run time, the given input parameter values can be matched with the corresponding formal parameters in LLVM IR, as specified by Algorithm 1.

The input parameter values are used for multi-versioning of the prototype functions. Each input parameter value is matched with a function parameter marked by PolyJIT at compile time (cf. Algorithm 2). PolyJIT holds a cache of all generated function variants. A hash calculated from the pointer to the prototype and the input parameter values used for multi-versioning serves as key for each variant.

Next, PolyJIT generates a new version of the prototype and substitutes all uses of the marked function arguments with a constant for each parameter value (Lines 4–5). This effectively transforms all non-affine expressions detected as strong parametrization by PolyJIT to affine ones. The new version is then optimized using the default compile-time polyhedral optimization pipeline provided by Polly. However, as an extension, Polly can be controlled by PolyJIT for each variant generated (i.e., Polly's configuration parameters can be tuned by PolyJIT). For example, we added a simple heuristics that is able to derive an appropriate tile size for the outermost parallel loop dimension, if any, in the variant generation step. After optimization, the variant is stored with its associated hash value in PolyJIT's function cache, such that a simple hash calculation and cache lookup will find it when it is called repeatedly.

Multi-versioning. The process of variant generation that is described in the following paragraph is an instance of function multi-versioning, a technique used by compiler optimizations and sample-based profilers [21]. Arbitrary pieces of code are cloned and modified to match different objectives (e.g., additional measurement instrumentation or different version for each accelerator card on

Input: a key ID, a list $Args$ of arguments
Output: a function to be executed on $Args$
Variables: a relation $prototypes \subseteq IDs \times Modules$, a relation $variants \subseteq IDs \times Modules$

1: $K \leftarrow key(ID, Args)$
2: $protoypeFunc \leftarrow select(ID, prototypes)$
3: $variantFunc \leftarrow copy(key, prototypeFunc)$
4: **for each** $p \in marked(params(prototypeFunc))$ **do**
5: $\quad uses(p, variantFunc) \leftarrow select(p, Args)$
6: $variants \leftarrow variants \cup (K, variantFunc)$
7: **if** $useable(prototypeFunc)$ **then**
8: $\quad apply(variantFunc, Args)$
9: **else**
10: $\quad apply(prototypeFunc, Args)$

Algorithm 4: EnterPolyJIT(ID, $Args$, $Stats$)

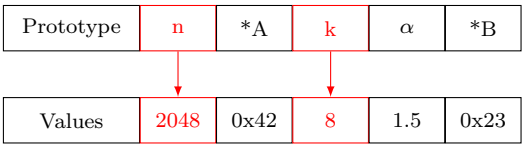

Figure 1: Select parameters for multi-versioning at run time.

the host). Different versions are meant to be activated conditionally. In general, multiple versions are generated at compile time and one is selected at run time or ahead of run time. Beside PolyJIT, a further example is Polly, which performs multi-versioning inside its code generation. In front of the original code region, Polly forms a new branch in the control flow and generates the code for the fully optimized SCoP into it, while the original code remains on the second branch. While Polly analyzes a SCoP, it may be required to make assumptions under which the optimized version of the SCoP may be run (e.g., assumptions about the bounds of structure parameters). These assumptions require a run-time check that forms the branch condition for the generated versions.

Variant generation. PolyJIT uses multi-versioning to spawn a new variant from a single prototype function. Function arguments that are suitable for multi-versioning have been marked by the detection process before (cf. Algorithm 1). Uses of these marked function arguments are replaced with the actual parameter values, as shown in Figure 1.

The new version of the prototype function, with all parameter values substituted, is then handed over to the polyhedral optimization pipeline driven by Polly. This pipeline includes all transformations available at compile time,

including the generation of alias checks that might be necessary to prove alias freedom at run time. Multi-versioning of all required parameters often leads to constant loop bounds in the outermost dimension of all transformed SCoP candidates. Therefore, we can determine the number of iterations of each loop dimension as a single scalar value and use it to guide Polly's first-level tiling. In PolyJIT's case, tiling serves to introduce task parallelism with OpenMP at the outermost dimension wherever suitable. The number of threads available for a task-parallel execution is known at run time. PolyJIT uses this knowledge to drive the tiling with simple heuristics that generate one tile for each thread available to the run-time environment. Inside each tile, a second round of tiling serves to increase data locality. Wherever suitable, SIMD vectorization is applied at the innermost level, driven by Polly's own cost model.

4 Evaluation

Typically, JIT compilers work with an intermediate representation of the input program, which is being profiled at run time to identify hot paths in the control-flow graph for further optimization. In contrast, PolyJIT avoids most costly run-time profiling because it can preselect all potentially hot code regions at compile time. In addition, the JIT approach permits PolyJIT to consider more liberal code formats than the polyhedron model can handle, such that the exploitation of later run-time knowledge will reduce the SCoPs back to a treatable format. Hence, we evaluate PolyJIT's capabilities on the execution of both the entire program and its individual SCoPs.

4.1 Research Questions

As a first research question, we would like to learn about PolyJIT's influence on the execution times of individual SCoPs:

***RQ*$_1$** Is PolyJIT able to reduce the execution time of individual SCoPs significantly compared to the best optimization that a traditional compiler can provide with or without polyhedral optimization enabled?

The ability to reduce the execution time of individual SCoPs is only half the story, though. A local speedup might be annihilated by the overhead incurred by the optimization necessary to achieve that speedup. Therefore, our second research question is:

***RQ*$_2$** Is PolyJIT able to reduce the overall execution time of entire programs significantly?

PolyJIT performs multi-versioning of structural parameters at run time. This empowers it to handle all array accesses covered by Polly's optimistic delinearization (Section 5)—and even some more. However, it requires run-time code generation. Thus, our third research question is:

***RQ*$_3$** Is multi-versioning at run time able to overcome the overhead introduced by run-time code generation and polyhedral optimization?

As noted before, PolyJIT can cover more memory accesses than optimistic delinearization at the price of run-time code generation. In cases that are covered by both approaches, we are interested in the differences in achievable performance. Therefore, our fourth research question is:

***RQ*$_4$** Does run-time value specialization result in more efficient code compared to using only delinearization of multi-dimensional array accesses?

4.2 Operationalization

Our four research questions require different measures. All configurations except the baseline ones are able to extract the following measures.

4.2.1 Measures

Definition 1 (Speedup & Slowdown) The *speedup* S_c of a program is the execution time T_0 of the program running in the baseline configuration, divided by the execution time T_c of the program running in configuration c. For values of T_c greater than T_0, we use its negative reciprocal, called *slowdown*:

$$S_c := \begin{cases} T_0/T_c & : T_0 \geq T_c \\ -T_c/T_0 & : \text{otherwise} \end{cases}$$

Note that, with our definition, we generate negative speedup values in the case of a slowdown (e.g., $T_A = 2, T_0 = 1 \rightarrow S_A = -2$). In addition to the classic speedup, we determine the fraction of the sequential run time that we can obtain with a polyhedral transformation [26]:

Definition 2 (Execution SCoP coverage) Let S be the set of SCoPs of a program. Let $t : S \rightarrow \mathbb{R}$ be a function that returns the accumulated run time of a SCoP in the run(s) of the program. Let T be the accumulated run time of the program for arbitrary sets of input values. The *execution SCoP coverage* (EC) of the program is:

$$EC := \frac{1}{T} \cdot \sum_{s \in S} t(s)$$

The execution SCoP coverage allows us to estimate the potential benefit of an optimization by determining the fraction of the sequential program's run time that is spent inside SCoPs. It reveals whether our transformations are able to hit the hot spot(s) of the program. The higher the SCoP coverage, the larger the potential effect of polyhedral optimization on a specific program run.

For an analysis of PolyJIT's run-time overhead, we use a third measure:

Definition 3 (Overhead Coverage) The *overhead coverage* (OC) of a program running in configuration c is the time O_c spent inside the JIT's internal execution phase in relation to the total execution time T_c:

$$OC_c := \frac{O_c}{T_c}$$

Furthermore, we count the number of variants (V) of versions that we were able to generate, the number of cache hits (C) that we were able to achieve, and the number of blocked (B) prototypes. A prototype is blocked from further optimization by PolyJIT if Polly is not able to generate optimized code at run time after run-time optimizations applied by PolyJIT.

For the presentation of descriptive statistics we list **n** as the number of observations, **Min** as the minimum, $\widetilde{\mathbf{x}}$ as the median, $\bar{x}_{\text{trim}}$ as the trimmed average with a trim factor of 5%, **Max** as the maximum, **IQR** as the interquartile range (i.e., the delta between the 1st and the 3rd quartile), and **s** as the standard deviation.

4.2.2 Configurations

The following configurations yield sets of measures that we employ to answer our research questions. The numbers and types of available measures depend on the configuration. All projects are compiled with the default options, supplied by the individual project. Deviations are documented in the source code of Benchbuild, a tool for large-scale empirical evaluation [28]. Benchbuild covers a wide range of different application domains including compression, databases, multimedia and verification.

We encode the properties and enabled features of all available configurations in their names. The two *static* (*stat*) configurations do not perform any optimization at run time and serve mainly as a baseline for our measurements. All other configurations are considered to be *dynamic* (*dyn*). Furthermore, we distinguish the configurations by their use of the following features.

- Polly (*polly*) enables optimization and code generation by Polly. All dynamic configurations have Polly enabled.
- Delinearization (*delin*) enables the recovery of multi-dimensional array accesses from their linearized representation in LLVM-IR.
- Specialization (*spec*) enables the run-time multi-versioning that replaces parameters with their known values.
- Adaptation ($^+$) enables run-time tuning of Polly's tiling configuration based on knowledge about the input program and the target machine. This feature uses a simple heuristic to optimize tile sizes of the outermost dimensions to adjust the workload size parallel threads and serves as a proof-of-concept that we can perform adaptive optimization of SCoPs.

Table 1 shows a feature matrix of all available configurations. All compile statically with the optimization level –O3 using clang. In all cases but one, Polly's parallel code generation is enabled and all other options remain on their default values.

	PolyJIT	Polly	Delinearization	Specialization	Adaptation
stat					
stat_polly		✓(1)	✓		
dyn_spec_delin	✓	✓(2)	✓	✓	
dyn_delin	✓	✓(2)	✓		
dyn	✓	✓(2)			
dyn_spec	✓	✓(2)		✓	
dyn_spec_delin$^+$	✓	✓(2)	✓	✓	✓

Table 1: All available configurations and their enabled features used for the evaluation of PolyJIT. Feature Polly is enabled at compile time when marked with (1) and enabled at run time when marked with (2). Polly's configuration options are left on their defaults, except for parallel code generation and *dyn_spec_delin*$^+$.

4.2.3 Measurement Strategies

The answers to our four research questions require different configurations of PolyJIT. Next, we introduce the configurations necessary and the measurement strategies applied to answer each question.

***RQ*$_1$** We are interested in the local speedup of each region, so we compare the execution time of each region in the following two configurations: the total execution time T_{dyn_delin} of configuration *dyn_delin* and the execution time $T_{dyn\_spec\_delin}$ of *dyn_spec_delin*. SCoPs are optimized with the default –O3 pipeline of LLVM after the polyhedral code generation has been completed.

***RQ*$_2$** We are also interested in the overall effect of PolyJIT. To this end, we compare the total execution time of *stat*, *stat_polly*, and *dyn_delin* to *dyn_spec_delin*. The comparison of PolyJIT with two versions of Polly, the compile-time version and the run-time version, exposes the effects of the overhead and changes due to a slightly different optimization pipeline and code structure. For example, in the run-time version of Polly, we cannot rely on the positive effects of inlining when working with extracted function prototypes.

***RQ*$_3$** While the previous two questions are directly related to the overall performance of PolyJIT, we are also interested in the costs incurred when applying run-time optimization to individual SCoPs. Beside execution times for regions and entire programs, we also collect detailed information about the execution SCoP coverage, the number of variants, the number of cache hits, and the number of blocked function prototypes. These measures influence directly the positive impact PolyJIT can have on a given program. All measures for this research question are taken from PolyJIT's default configuration *dyn_spec_delin*.

***RQ*$_4$** One source of non-affine expressions in a SCoP candidate is the linearization of multi-dimensional array accesses caused by LLVM's internal representation of arrays with pointers. It is possible to reclaim a suitable—but, in general, not identical—multi-dimensional representation for this kind of array access [27]. This reconstruction is often called "optimistic", because it is still necessary to verify the correctness of the reconstruction at run time. We claim that run-time value specialization subsumes optimistic delinearization. In general, we can compare the execution times of *dyn_spec* to *dyn_spec_delin*. Since PolyJIT hands off all optimized SCoPs to Polly at run time, there should be no difference in the code observed after run-time value specialization and, therefore, no difference in the total execution time of the program, because all memory accesses are represented in linear form to Polly. Deviations can occur when not all parameters are selected for specialization at compile time. However, these should not occur in *dyn_spec_delin*, because Polly can attempt to delinearize the missing parametrized accesses. Additionally, we compare the execution times of *dyn_spec_delin*, *dyn_spec*, and *dyn_spec_delin*$^+$ to *dyn*. This gives an indication of how the run-time specialization influences the total execution time when isolated from the efffects of delinearization. While both features can coexist, as in *dyn_spec_delin*, the absence of structural parameters in the polyhedral representation opens up more opportunities for run-time adaptive optimization, as in *dyn_spec_delin*$^+$.

4.3 Subject Systems and Benchmarks

We base our evaluation on a large variety of 334 real-world programs and community benchmarks, some of which have been used in previous work on polyhedral optimization [26]. The configurations for PolyJIT's evaluation are implemented as parts of a reusable experiment inside Benchbuild, which facilitates reproducability. We selected 334 programs included in Benchbuild that contained, at least, one SCoP.

Table 2 (page 32) lists all programs in our collection on which PolyJIT was able to act, that is, that contain at least one dynamic SCoP.

The column "Domain" of Table 2 lists the program groups, each of which is an application domain or benchmark. Below, we list all available application domains available for testing in Benchbuild. Note that Table 2 only lists the first six of these groups, because (1) for some, the filters applied to the dataset remove all candidates from the other three, (2) for some, PolyJIT is not triggered at run time, and (3) for some, PolyJIT and/or Polly fail to complete the run-time test successfully.

- *Benchbuild*: Applications that belong to Benchbuild's initial set of programs used in [26]. This includes programs from the (sub-)domains *Compilation*, *Encryption*, and *Scientific* (e.g., bzip2, SevenZ, xz, x264, FFmpeg, lapack, and lulesh)
- *BOTS*: The Barcelona OpenMP Task Suite, a set of benchmarks prepared to be executed with OpenMP enabled [11].

- *LNT (MSB)*: A set of multi-source benchmark programs included in LNT.
- *Polybench*: Benchmark codes found in the POLYBENCH benchmark suite. These benchmarks are specifically crafted to benefit from a polyhedral optimization. In the context of POLYJIT, they are especially useful for tests of changes to tuning parameters of the polyhedral optimizer POLLY.
- *SPEC*: Benchmark codes found in the SPEC CPU2006 benchmark suite (e.g., MCF, MILC, PERLBENCH). Typically, these are larger applications with exhaustive program inputs for each benchmark.
- *LNT (SSB)*: A set of single-source benchmark programs included in the LLVM nightly test suite (LNT).
- *LNT (MSA)*: A set of single-source application programs included in LNT.
- *Rodinia*: A set of work-intensive programs intended for benchmarking of heterogeneous compute clusters [7].
- *Scimark*: A set of benchmark programs with scientific background [25].

The rows of Table 2 are sorted by domain and, in each domain, by execution SCoP coverage (EC) in descending order. Elements closer to the top of their respective section in the table indicate that a successful polyhedral optimization has a higher impact on the total execution time of the program. Note that not all domains available in BENCHBUILD are displayed in Table 2, because of the filters applied to the measurement results. Most commonly, we were able to identify SCoPs that require POLYJIT at compile time, but the available run-time tests for these programs failed to activate POLYJIT at least once.

4.4 Measurement Setup

We conducted all experiments on a dual socket system with 2x Intel Xeon E5-2650v2 (8 physical cores, 2.60 GHz, 16 threads) and 128 Gb RAM (64 Gb RAM per socket). During all measurements, we were the exclusive user of the machine, and each test was restricted to 8 cores. The system's capability to boost processor speed, depending on system load (Intel Turbo Boost) has been turned off.

We ran each configuration as a separate experiment. All configurations that require POLYJIT's run-time library to be enabled collect all timing information as part of the integrated regionwise instrumentation interface. The configurations *stat* and *stat_polly* are not able to provide regionwise information and, therefore, only measure the total execution time of each project.

Due to space limitations, we present only a selection of evaluation results here. A comprehensive collection can be found at the POLYJIT Web site.

4.5 Results

Before we answer our research questions in detail, we want to provide a descriptive view on our subject programs. We are interested in the number of SCoPs that POLYJIT's detection algorithm can detect and if it can find more

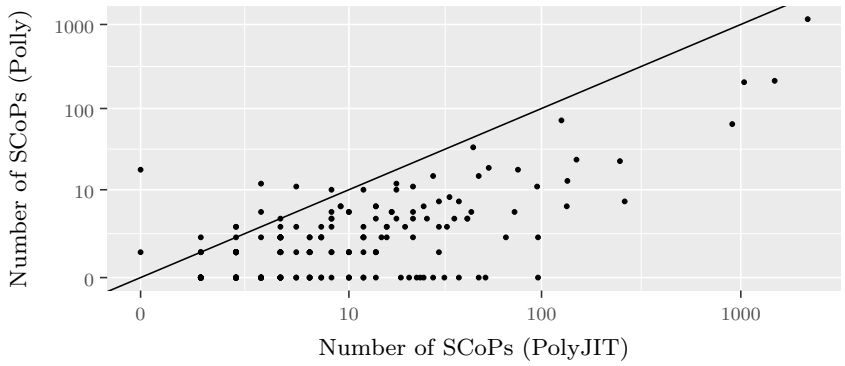

Figure 2: Number of SCoPs detected by Polly compared to number of SCoPs detected by PolyJIT for 206 of 334 programs.

SCoPs than Polly's implementation, which can indicate that we have more opportunities for polyhedral optimization at run time.

The experiment ran on 334 software systems of Benchbuild and returned valid results for 206. Figure 2 shows a scatter plot that compares the number of SCoPs detected by PolyJIT to those detected by Polly. Despite differences in the detection algorithm, we can clearly see that there is a significant increase in the number of detectable SCoPs for PolyJIT.

PolyJIT's SCoP detection algorithm is a modified version of Polly's that, in addition to the recognition of strongly parametrized non-affine expressions, does not take delinearizable multi-dimensional array accesses into account. Therefore we cannot decide, wheter some SCoPs that are detected by PolyJIT may also be detected by Polly as part of delinearization. However, both algorithms share the decision logic of profitable SCoPs and the expansion logic to find a SCoP of maximal size. Additionally, we need to consider that the raw number of SCoPs is not a sufficient metric for quantifying the effectiveness of the detection process in general. A lower number in detected SCoPs can be the result of a larger one being detected instead multiple smaller ones. We always prefer the larger SCoP, because of a greater chance for uncovering parallelism. However, the two detection algorithms for Polly and PolyJIT are similar enough to compare the raw numbers. Especially on larger programs, such as SevenZ, or FFmpeg, we found a substantial increase in the number of SCoPs detected by PolyJIT.

Let us now answer the research questions in the order in which Section 4.1 states them.

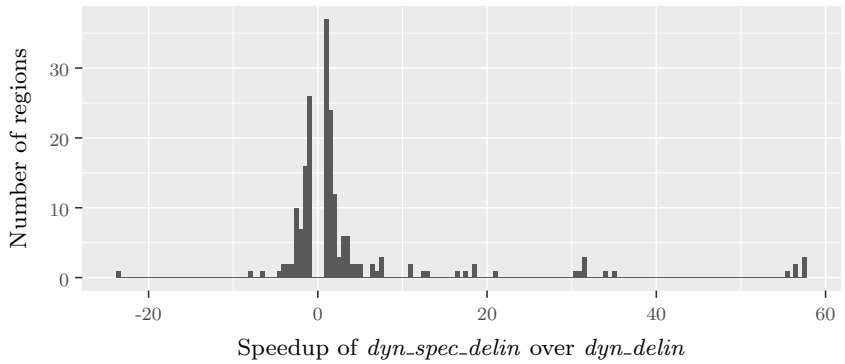

Figure 3: Histogram of all regionwise measurements, excluding the band between speedup values of –1.1 and 1.1. We used a binwidth of 0.5.

Variable	**n**	**Min**	$\widetilde{\mathbf{x}}$	$\bar{x}_{\text{trim}}$	**Max**	**IQR**	**s**
$T_{dyn\_spec\_delin}$ [µs]	704	1.0	$5.0{\cdot}10^{4}$	$6.2{\cdot}10^{5}$	$2.5{\cdot}10^{8}$	$3.6{\cdot}10^{5}$	$2.9{\cdot}10^{7}$
T_{stat_polly} [µs]	704	1.0	$5.0{\cdot}10^{4}$	$3.2{\cdot}10^{5}$	$1.6{\cdot}10^{8}$	$3.6{\cdot}10^{5}$	$1.1{\cdot}10^{7}$
S_P	704	$-2.4{\cdot}10^{1}$	1.0	$3 \cdot 10^{-1}$	$5.8{\cdot}10^{1}$	2.1	6.5

Table 3: Descriptive statistics for all regionwise measurements.

***RQ*$_1$** Figure 3 shows the descriptive statistics of all regionwise measurements for each variable that compare *dyn_delin* to *dyn_spec_delin*.

We did not filter the regionwise measurements in any way, because all require PolyJIT to engage at least once. This leaves 135 programs with 704 regions in total. We provide the complete table of regionwise results on the project's Web site.

At first glance, Table 3 suggests that the code generated by PolyJIT does not provide any significant speedup over Polly across all regions. We achieve a trimmed mean of 0.29, over Polly with an inter-quartile range (IQR) of 2.06, indicating that the majority of measurement results is packed tightly around the median. Furthermore, the standard deviation of 6.54 also indicates that the data spread is relatively high. Excluding all measurements between speedup values of –1.1 and 1.1, we found 69 regions that show speedup values below –1.1 and 122 regions that show speedup values above 1.1.

Figure 3 shows the distribution of speedup values of *dyn_spec_delin* over the baseline *dyn_delin*. While a few regions achieve speedup factors of up to 57.69, most SCoPs do not get beyond a speedup of 1 over Polly. 67 regions show speedup values between 1.1 and 2, indicating that the benefit over Polly is caused by data locality optimizations or the ability to generate simpler code due to parameter value substitution. The regions that achieve a positive speedup are typically found in domains that are considered optimal candidates for polyhedral optimization (Scientific, Polybench, and Multimedia).

Note, though, that across all SCoPs that exhibit a slowdown, the majority (45 out of 69) show only a slowdown of less than –2. The occurrence of slowdowns suggests that there is potential for improvement in the selection and postprocessing of polyhedral optimizations applied to SCoPs.

In summary, we have shown that PolyJIT is able to improve the performance of individual SCoPs at run time albeit, in classic applications domains, more than in others.

Var.	Levels	n	Min	$\widetilde{\mathbf{x}}$	$\bar{x}_{\text{trim}}$	Max	IQR	s
S_{O3}	*dyn_spec_delin*	53	0.2	0.9	2.1	13.0	2.3	2.8
	dyn_spec	53	0.2	1.0	1.6	8.7	0.7	2.0
	dyn_delin	53	0.1	0.9	1.7	12.2	1.4	2.4
	dyn	53	0.1	0.9	1.0	6.7	0.2	1.1
	dyn_spec_delin$^+$	53	0.2	1.0	2.1	13.0	2.5	2.7
	stat_polly	53	0.8	1.0	1.6	8.4	0.7	1.7
	stat	53	1.0	1.0	1.0	1.0	0.0	0.0
	all	371	0.1	1.0	1.5	13.0	0.6	2.1
S_P	*dyn_spec_delin*	53	0.1	0.9	1.2	6.7	0.4	1.4
	dyn_spec	53	0.1	0.9	1.0	6.7	0.5	1.4
	dyn_delin	53	0.1	0.9	1.0	6.7	0.5	1.4
	dyn	53	0.1	0.8	0.8	6.7	0.6	1.2
	dyn_spec_delin$^+$	53	0.1	0.9	1.2	6.8	0.4	1.4
	stat_polly	53	1.0	1.0	1.0	1.0	0.0	0.0
	stat	53	0.1	1.0	0.8	1.2	0.4	0.3
	all	371	0.1	1.0	0.9	6.8	0.4	1.2

Table 4: Descriptive statistics for all measurement configurations.

***RQ*$_2$** From 334 projects in our test corpus, we excluded all that did not trigger PolyJIT at least once. This results in the 53 programs listed in Table 2. Table 4 provides descriptive statistics for speedup values over the baselines *stat* and *stat_polly* for all these programs. In contrast to Table 2, the speedup values are calculated using the standard formula T_1/T_p, because we need a continuous value range for descriptive statistics calculation.

Figures 4 and 5 show the speedup values over baseline *stat* and *stat_polly* of all projects included in Table 2 in the form of violin plots [18] per configuration. The violins between configurations share the same maximum width. The difference between dynamic configurations is quite small, which is caused by the two complementary ways of dealing with non-affinity. Notice the clear distinction when we compare *dyn* with baseline *stat* to *dyn_spec_delin*: here, neither multi-versioning nor Polly's own delinearization are enabled. Additionally, while Polly achieves speedups without projects causing a high slowdown, it does not achieve as many and high speedups as the dynamic configurations using PolyJIT.

We filtered out all programs with an execution time lower than 1s in configuration *stat* (e.g., small benchmark programs or programs with small problem sizes as input that lead to short execution times). The remaining set

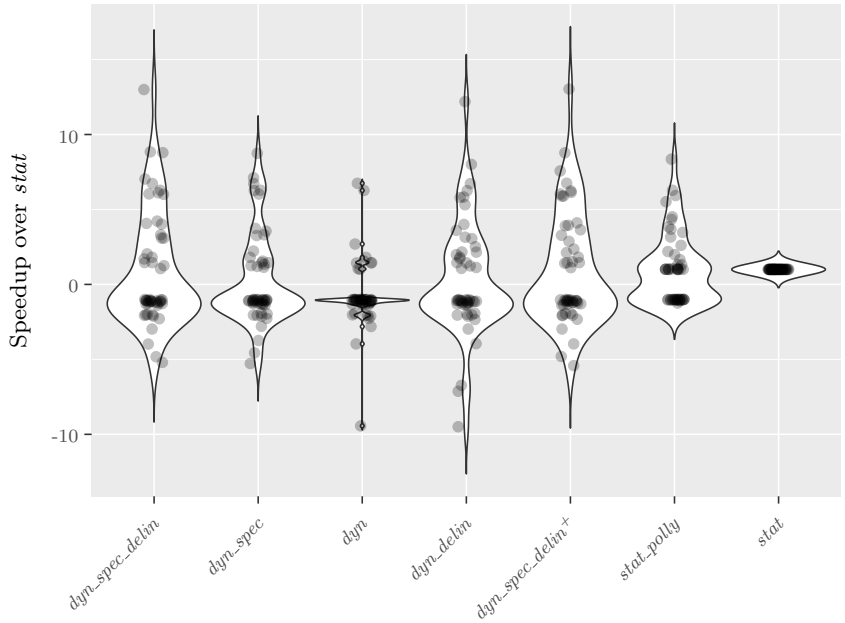

Figure 4: Violin plots of the speedup density of all projects for each configuration available. The baseline for this set of plots is *stat*.

of programs contains 53 elements. At first glance, Table 4 shows median values between 0.9 and 1.0, which suggests that none of the configurations are able to improve the overall performance of the complete set on average over *stat* or *stat_polly*. This indicates that we have a large number of projects in the test corpus that are not amenable for polyhedral optimization at all, even though the SCoP selection suggests a profitable optimization. However, we are still able to discuss the benefits of polyhedral optimization on single programs instead of considering the general applicability. A closer look shows that *stat_polly* achieves the highest minimum speedup over *stat* (0.8). The worst performance is delivered by *dyn*, which is expected because, in terms of optimization opportunities, it is the most constrained configuration of PolyJIT. However, the trimmed mean suggests that *dyn_spec_delin* performs better, on average, achieving a maximum speedup of 0.19 over *stat* and 0.15 over *stat_polly*. This higher average speedup comes at the cost of a higher spread in speedup values across the complete set of test programs, with a standard deviation of 2.06 for *dyn_spec_delin* compared to the standard deviation of 1.67 for *stat_polly*.

A performance slightly worse than Polly on a large test suite is not surprising because our test suite includes projects in which we cannot beneficially improve all SCoPs, but we still pay for the added overhead of JIT compilation. This is evidenced by Table 2, which contains a more detailed view on the 53

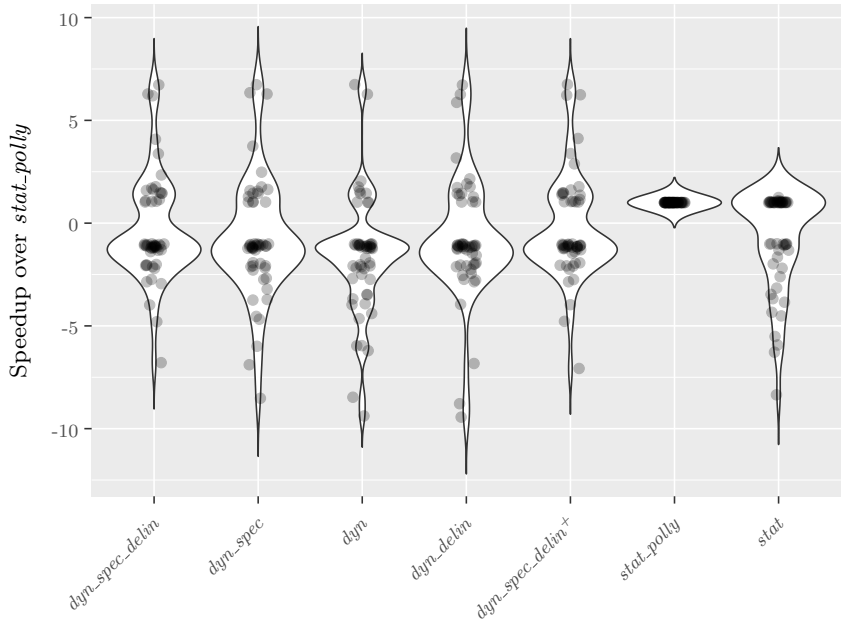

Figure 5: Violin plots of the speedup density of all projects for each configuration available. The baseline for this set of plots is *stat_polly*.

programs that were optimizable by PolyJIT. Projects in which the number of variants is equal or almost equal to the number of blocked functions are not filtered from the results because, at compile time, we assumed these functions to be profitable for PolyJIT. Either our generated code after multi-versioning is still not suitable for polyhedral optimization or Polly deemed the optimization opportunities not profitable and rejected optimization of the SCoP.

It is remarkable that we observe positive effects when the number of variants is equal to the number of blocked functions (e.g., 473.astar, fft, flops, IndirectAddressing, Reductions). These cannot be caused directly by the optimization pipeline of PolyJIT, because all variants use the compiled version of the prototype function. In these cases, we would expect a slowdown caused by the added overhead for one-time compilation inside PolyJIT, which is the case for other projects, e.g., bullet, chomp, or floorplan. Overall, we achieve speedup values between –9.49 and 13.03 for *stat* and values between –9.44 and 6.75 for *stat_polly*.

In summary, we have shown that PolyJIT is able to translate the speedup achieved on SCoPs to a speedup of complete programs in many cases. However, many programs remain on whose performance PolyJIT has a negative impact.

Var.	n	Min	$\tilde{x}$	$\bar{x}_{\text{trim}}$	Max	IQR	s
variants	53	1.0	2.0	5.9	301.0	7.0	41.2
blocked	53	0.0	2.0	4.4	98.0	6.0	14.3
cache hits	53	0.0	1.0	8489.1	920944.0	2.0	144394.0
execution SCoP coverage	53	8.6	53.7	53.7	99.2	55.7	30.7
overhead coverage	53	0.2	4.9	7.4	30.5	11.0	8.1

Table 5: Descriptive statistics for all measurement variables.

***RQ*$_3$** Table 4 shows descriptive statistics for the experiment variables V, B, C, EC and OC for the configuration *dyn_spec_delin*. Our measurements reveal that, accross 53 projects, we have to generate 5.88 variants, on average, with a median of 2. As the number of variants required determines the amount of compilation we have to perform during run time, it is beneficial for PolyJIT if the number of variants does not become too large. This is the case for our dataset with a variant count of 1 to 301. At the same time, the number of blocked variants needs to be significantly lower than the number of variants, because a blocked variant means that we spent compilation time without obtaining an optimized function. On our dataset, we find that we block between 0 and 98 functions with an average of 4. This number is close to the average number of variants. We evaluate this more thoroughly in Section 4.6. While PolyJIT spends time compiling function variants during the program's execution (synchronously as well as asynchronously), it can profit from function variant requests to already compiled ones. On average, we can resolve 8489 function variant requests from the cache, with a range from 0 to 920 944 cache hits. Given the low median of 1, we have to assume that one half of our input programs only require 0 to 1 cache hits, as is confirmed by Table 2. This suggests that we frequently generate functions that are called only once.

The compilation overhead generated by PolyJIT is quantified by variable OC. For OC, we achieve values between 0.24% and 30.48%, with an average of 7% of the program's execution time spent compiling new function variants. This overhead in compilation time must be compensated for in terms of optimizable execution time, captured by variable EC. Ranging from 8.64 to 99.19, with an average EC value of 54, our data suggests that PolyJIT's SCoP detection provides enough opportunities to compensate for the compilation overhead. More precisely, a quick T-Test of the hypothesis H_0: $\mu_{\text{EC}} \geq \mu_{\text{OC}}$ reveals that the overhead is significantly smaller than the execution coverage ($\text{p}=1.89 \cdot 10^{-15}$).

In summary, we have shown that PolyJIT's execution time overhead can be considered negligible, given a sufficiently large value of EC.

***RQ*$_4$** Table 4 shows how run-time multi-versioning performs when compared to classic delinearization. As a first step, we compare *dyn_spec* to *stat_polly*, that is, we perform full run-time multi-versioning but no delinearization of memory accesses inside Polly, and compare it to Polly at compile time. Both configurations behave similarly in terms of speedup over *stat*, on average, with

an average speedup of 1.58 (median: 0.96) for *dyn_spec* and 1.64 (median: 1) for *stat_polly*. This is expected because disallowing delinearization for Polly should be irrelevant in the presence of run-time value specialization by PolyJIT. All other statistics point in the same direction, except the lower minimum speedup of 0.19 compared to 0.8. We suspect that this accounts for compilation overhead caused by PolyJIT, which results in a slower possible speedup as discussed in the previous paragraph.

Figure 6 (page 33) shows a comparison of speedup values for the two baseline configurations *stat* (horizontal axis) and *stat_polly* (vertical axis). The white area marks zones in which the introduced speedup measure is not continuous. Ideally, all values should reside in the first quadrant for *dyn_spec_delin*, because then we would achieve higher speedups over both baseline configurations. The second quadrant means that we are slower than *stat* but faster than *stat_polly*. This can be the case if polyhedral transformations found by Polly cause a slowdown and the optimizations of PolyJIT cannot compensate fully for it. The third quadrant collects cases in which PolyJIT fails to improve the program and causes slowdown compared to both baseline configurations. The fourth quadrant houses all measurements in which PolyJIT manages to improve over *stat*, but to the same or a lesser degree than *stat_polly*. At first glance, Figure 6 shows that *dyn_spec_delin* behaves similarly to *dyn_spec*, except for a few programs in the third quadrant for the latter configuration. *dyn* performs worst, which is expected because there is nothing that removes non-affine accesses in the SCoP candidates that are presented to Polly. As an aside, *dyn_delin* also looks similar to *dyn_spec_delin* because, on our test set, there are many programs that belong to the classic polyhedral domain, for example, Polybench. This leads to delinearization behaving almost identically to run-time value specialization on these benchmark programs.

In summary, we have shown that parameter value specialization behaves similarly but not identically to optimistic delinearization. Additionally, in many cases, parameter value specialization is able to improve the execution time even further, because it is able to simplify the generated code after optimization.

4.6 Discussion

Our experiments have demonstrated that PolyJIT is able to handle a large number of SCoPs (704), distributed over a wide variety of programs from different domains. 122 out of 704 can be optimized at run time and achieve a reduction in execution time of up to 57.69. Especially larger speedups can be gained in programming domains that are known to be amenable for polyhedral optimization, such as Polybench or the domain "Scientific". However, over all 69 cases, we found that the slowdown stays mostly below -2. Here, future advances in polyhedral scheduling or postprocessing have a chance of generating new opportunities. Additionally, it is necessary to impose a cost model on PolyJIT since short-running regions (e.g., below 1ms) cause more overhead than possible speedup, even if invoked multiple times.

```
<...>
create_matrix(float **mp, int size) {
    float *m;
    float coe[2*size-1];
    <...>
    m = (float*) malloc(sizeof(float)*size*size);
    <...>
    for (int i=0; i < size; i++) {
        for (int j=0; j < size; j++) {
            m[i*size+j] = coe[size-1-i+j];
        }
    }
}
```

(a) Source code taken from the OpenMP version of the LUD benchmark included in Rodinia version 3.1, file name: `common/common.c`

```
#pragma omp parallel for
for (int c0 = 0; c0 <= 249; c0 += 1) {
    for (int c1 = 0; c1 <= 249; c1 += 1) {
        // 1st level tiling - Points
        for (int c2 = 0; c2 <= 31; c2 += 1) {
            #pragma simd
            for (int c3 = 0; c3 <= 31; c3 += 1) {
                Stmt6(32 * c0 + c2, 32 * c1 + c3);
            }
        }
    }
}
Stmt10();
```

(b) AST generated by Polly after PolyJIT's parameter value substitution for `size = 32`

Listing 4: Small real-world example SCoP found by PolyJIT that required run-time support to be detected by Polly. While Polly should be able to find a multi-dimensional representation of array `m` in the form `m[size][size]`, it rejects the candidate with the diagnostic message: "The array subscript of "UNKNOWN" is not affine".

A small example SCoP found in our set of subject programs and optimized by PolyJIT is shown in Listing 4. Here, the source code contains a multi-dimensional array implemented by means of linearized memory accesses, which result in non-affine expressions in the array subscripts (shown in Listing 4a). Polly fails to delinearize the memory access in this case and rejects the SCoP because of the non-affine memory accesses occuring inside the loop body. PolyJIT performs multi-versioning based on the parameter `size` at run time and generates the tiled and vectorized code shown in Listing 4b. More examples can be found on the project's Web site.

An inspection of PolyJIT's effect on the overall execution time yields mixed results. While we gain overall performance for Polybench and LNT, we often pay for the added compilation overhead in cases that cannot achieve a speedup over existing performance of either *stat* or *stat_polly*. On average, we achieve a speedup of 2.07 and 1.16, respectively. Results from other domains

are included on the project's Web site, but are excluded here due to our filter rules for measurement results. Typically, the EC values of the projects need to be high enough for a polyhedral transformation to have an effect on the generated code, which is not the case for many programs from other domains.

Since a program of PolyBench typically contains a single loop kernel that produces a large number of iterations, the decrease in execution time is able to outweigh the overhead of code generation completely. Outside PolyBench, we experience a much higher frequency of SCoP invocations (8489.1, on average) with a moderate number of different parameter values (5.88, on average). In this setting, PolyJIT needs to hide the overhead caused by code generation as much as possible. We have achieved this by means of asynchronous function-variant generation, which makes us benefit from the optimized code every time the program requests a variant after code generation is completed. Our experiments also established that the overhead of PolyJIT's parameter redirection in addition to its run-time compilation is negligible (OC 7%). In its default configuration, PolyJIT strives to hide this overhead partly by asynchronous code generation in combination with a function variant cache. When considering the large numbers of cache hits (8489.1, on average), this proves to be a good way to reduce the perceived overhead for the end user.

When we compare PolyJIT's parameter-value specialization at run time to delinearization at compile time, we find that they both behave similarly when applied in isolation. This follows naturally, because all non-affine memory accesses that were created by linearization are becoming affine as soon as we specialize for all loop-invariant structural parameter values. However, our results show that the speedup improves, on average, from 1.71 and 1.58 to 2.07. This shows that parameter value specialization offers a potential for further optimization on top of the elimination of non-affine access functions.

So, overall, we conclude that it is beneficial to apply polyhedral optimization at run time on programs of both the domain of polyhedral applications as well as real-world applications. We showed that the SCoPs encountered outside of the classic polyhedral programs require more effort on the part of the JIT code generation, because SCoP candidates tend to be called with a higher frequency and a higher number of different parameter values. For use in practice, our research prototype PolyJIT still has to cope with a large amount of overhead during code generation, which limits it to SCoPs with a sufficient execution time. We discuss possible solutions that reduce the overhead further in Section 6.

4.7 Deviations & Threats to Validity

We experienced measurement bias in a few cases, caused by interference of the host system with the execution of the binaries, which resulted in fluctuating execution times between different configurations. While this does not affect our conclusions, as we discuss in Section 4.6, we tried to counter this effect by pinning each thread to its own core on the host system.

Construct validity We measure execution time per dynamic SCoP. This requires the placement of timing calls around each SCoP, which has an influence on the overhead generated by PolyJIT and, therefore, reduces its perceived benefit the more frequently its run-time environment is re-entered. We calibrated the time necessary for a single timing measurement and counted the number of timing events generated. The overhead introduced was linear in the number of timing calls. All measurements were conducted on hardware with a running operating system. Even though we control the affinity mask and processor frequency settings, there is still a chance to suffer from interference by tasks that run on the same system. Therefore, for a SCoP to be considered a good region, we require it to be at least 10% faster than the baseline.

Internal validity Our experiments rely on the quality of our input data for run-time measurements. We addressed this threat to validity by choosing the developers' own benchmark sets or by using the known default benchmark of the corresponding domain, such as FATE [10] for audio-video codecs. We assume that the developers' own test cases cover the important code paths and that the default benchmarks of a domain cover the most common use cases.

External validity As with any other comparable empirical study, the selection of subject programs threatens the generalizability of our results. We controlled this threat sufficiently by selecting a large and diverse number of subject systems from different areas.

5 Related Work

Grosser et al.'s work on optimistic delinearization of linearized memory access functions [15] provides a compile-time alternative to multi-versioning at run time. It recovers the multi-dimensional nature of linearized memory accesses with a possible representation of them. Parametric terms that occur in a linearized access can encode the size information for each original array dimension. Since this reconstruction is optimistic, it must be guarded by a run-time check that verifies any assumptions on array sizes taken to avoid illegal memory accesses. Multi-versioning at run time in the form used by PolyJIT can deal with the same cases as the delinearization without the need for a run-time check.

Sambamba [31] is a JIT compiler like PolyJIT that performs run-time adaptive parallelization based on program dependence graphs. In contrast to PolyJIT's focus on polyhedral optimization, it exposes general task parallelism via speculative execution and privatization, with special attention paid to the computational pattern of reduction. Both PolyJIT and Sambamba share the preparation of serialized micro-kernels at compile time for use at run time.

In contrast to PolyJIT's approach to detecting dynamic regions, the speculative loop parallelizing framework Apollo [6] uses statement-based parameterized code snippets, so-called *code bones*, to prepare dynamic SCoPs for speculative parallelization at run time. Apollo's run-time system orchestrates

the execution of all code bones. At compile time, multiple versions are generated for each code bone. One assumes the role of a verification code snippet, whereas the others provide different templated optimizations. At run time, each loop nest is executed in a rolling-window fashion, which offers the opportunity to monitor the running code bone and employ adaptations, if necessary.

VMAD [20] is the predecessor of the Apollo framework. It relies on an earlier variant of Apollo's code bones, the *code skeletons*. These are selected, templated skeletons that support only a limited set of polyhedral transformations that already have to be known at compile time. Furthermore, no transformations that alter the number of statements (e.g., loop fission) are supported.

Alenov et al.'s work on enabling SIMD intrinsics for managed run-time systems [29] proposes a more manual approach by unlocking the performance of SIMD instructions on high-level languages such as Java for the developer. While PolyJIT focusses on uncovering the potential to generate parallel code fully automatically at run time, their work enables the developer to make semi-automatic use of a high-level language in combination with the low-level capabilities of the processor's vector instructions using Scala's lightweight modular staging (LMS) framework.

As PolyJIT strives to optimize the run-time performance of programs, we also depend on capabilities for post-processing SCoPs that have been optimized according to the model's objective function. This typically involves tiling to improve data-locality of the transformed SCoPs. There is a large body of work on a variety of tiling strategies for a variety of architectures, including diamond tiling [32], hexagonal tiling [13], and traditional rectangular tiling [19], [36]. However, these typically focus on properties such as minimization of inter-tile communication, or maximal distribution of tiles on the available processors. The target-dependent part of optimal sizes is not considered. Commonly, one resorts to manual tuning by an expert or auto-tuning, neither of which is suitable for the low-latency environments that PolyJIT targets. However, there are more promising approaches that try to dynamically select corrrect tile sizes [33], or even propose model based tile-size selection [37,23]. A prototype implementation of [23] inside PolyJIT did not yield stable results yet.

6 Conclusions

The polyhedron model is a well-studied approach to automatic program optimization, typically, used in the domains of highly regular high-performance and linear-algebra programs. With its inclusion in modern compiler frameworks, such as LLVM and GCC, there is the desire for applying polyhedral optimization techniques to—more complex—general-purpose programs. The irregularity of general-purpose programs often impairs a polyhedral optimization at compile time, though. We propose PolyJIT, a light-weight JIT compiler that moves classic polyhedral optimization techniques from compile time to run time. In a diverse corpus of 53 real-world programs and community benchmarks, we achieved speedups of up to 13.03. More precisely, with the use of

multi-versioning and run-time knowledge, we were able to improve the performance of 122 SCoPs with speedups between –23.67 and 57.69. Not surprisingly, classic polyhedral programs showed large speedups in our experiments, even in the presence of PolyJIT's run-time overhead. This is mainly due to task parallelism and data-locality optimization, in conjunction with sufficient execution time spent inside SCoPs. While programs beyond the polyhedral domain scaled less well for task parallelism, data-locality optimization still yielded considerable speedups.

Overall, PolyJIT's results demonstrate that automatic program optimization based on the polyhedral model can yield promising results when applied to general-purpose programs.

References

1. Android Developers: Art and Dalvik (2016). https://source.android.com/devices/tech/dalvik/
2. Banerjee, U.: Loop nest parallelization. In: D. Padua, et al. (eds.) Encyclopedia of Parallel Computing, vol. 2, pp. 1068–1079. Springer (2011)
3. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: Proc. 13th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT), pp. 7–16. IEEE Computer Society (2004)
4. Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In: Proc. 17th Int'l Conf. on Compiler Construction (CC). Springer (2008)
5. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral program optimization system. In: Proc. 29th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). ACM (2008)
6. Caamaño, J.M.M., Selva, M., Clauss, P., Baloian, A., Wolff, W.: Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones. Concurrency and Computation: Practice and Experience **29**(15), 4192:1–4192:16. Special Issue on Euro-Par 2016
7. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: Proc. IEEE Int'l Symp. on Workload Characterization (IISWC), pp. 44–54. IEEE Computer Society (2009)
8. Cook, S.: CUDA Programming: A Developer's Guide to Parallel Computing with GPUs. Morgan Kaufmann (2013)
9. Davis, M.: Hilbert's tenth problem is unsolvable. American Mathematical Monthly **80**(3), 233–269 (1973)
10. FFmpeg Developers: FFmpeg Automated Testing Environment (2016). https://www.ffmpeg.org/fate.html
11. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: Intl. Conf. on Parallel Processing (ICPP), pp. 124–131 (2009)
12. Feautrier, P., Lengauer, C.: Polyhedron model. In: D. Padua, et al. (eds.) Encyclopedia of Parallel Computing, vol. 4, pp. 1581–1592. Springer (2011)
13. Grosser, T., Cohen, A., Holewinski, J., Sadayappan, P., Verdoolaege, S.: Hybrid hexagonal/classical tiling for GPUs. In: Proc. 12th Int'l. Symp. on Code Generation and Optimization (CGO). ACM (2014). Article 66, 10 pp.
14. Grosser, T., Größlinger, A., Lengauer, C.: Polly – Performing polyhedral optimizations on a low-level intermediate representation. Parallel Processing Letters (PPL) **22**(4) (2012). 28 pp.

15. Grosser, T., Ramanujam, J., Pouchet, L.N., Sadayappan, P., Pop, S.: Optimistic delinearization of parametrically sized arrays. In: Proc. 29th ACM Int'l Conf. on Supercomputing (ICS), pp. 351–360. ACM (2015)
16. Grosser, T., Zheng, H., Alor, R., Simbürger, A., Größlinger, A., Pouchet, L.N.: Polly – Polyhedral optimization in LLVM. In: C. Alias, C. Bastoul (eds.) Proc. First Int'l Workshop on Polyhedral Compilation Techniques (IMPACT). INRIA Grenoble Rhône-Alpes (2011). 6 pages
17. Größlinger, A.: The challenges of non-linear parameters and variables in automatic loop parallelisation. Doctoral thesis, Department of Computer Science and Mathematics, University of Passau (2009)
18. Hintze, J.L., Nelson, R.D.: Violin plots: A box plot-density trace synergism. The American Statistician **52**(2), 181–184 (1998)
19. Irigoin, F.: Tiling. In: D. Padua, et al. (eds.) Encyclopedia of Parallel Computing, vol. 4, pp. 2041–2049. Springer (2011)
20. Jimborean, A.: Adapting the polytope model for dynamic and speculative parallelization. Doctoral thesis, Image Sciences, Computer Sciences and Remote Sensing Laboratory, University of Strasbourg (2012)
21. Jimborean, A., Loechner, V., Clauss, P.: Handling multi-versioning in LLVM: Code tracking and cloning. In: Proc. Int'l Workshop on Intermediate Representations (WIR). IEEE Computer Society (2011)
22. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. Second Int'l Symp. on Code Generation and Optimization (CGO), pp. 75–86. IEEE Computer Society (2004)
23. Mehta, S., Beeraka, G., Yew, P.: Tile size selection revisited. ACM Trans. on Architecture and Code Optimization (TACO) **10**(4), 35:1–35:27 (2013)
24. Paleczny, M., Vick, C., Click, C.: The Java Hotspot server compiler. In: Proc. 1st Symp. on Java Virtual Machine Research and Technology (JVM). USENIX Association (2001). 13 pages
25. Pozo, R., Miller, B.R.: SciMark2 (2017). http://math.nist.gov/scimark2
26. Simbürger, A., Apel, S., Größlinger, A., Lengauer, C.: The potential of polyhedral optimization: An empirical study. In: Proc. 28th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE), pp. 508–518. IEEE Computer Society (2013)
27. Simbürger, A., Größlinger, A.: On the variety of static control parts in real-world programs: from affine via multi-dimensional to polynomial and just-in-time. In: Proc. 4th Int'l Workshop on Polyhedral Compilation Techniques (IMPACT) (2014)
28. Simbürger, A., Sattler, F., Größlinger, A., Lengauer, C.: BenchBuild: A large-scale empirical-research toolkit. Tech. Rep. MIP-1602, Faculty of Computer Science and Mathematics, University of Passau (2016). 6 pages
29. Stojanov, A., Toskov, I., Rompf, T., Püschel, M.: SIMD intrinsics on managed language runtimes. In: Proc. 15th Int'l. Symp. on Code Generation and Optimization (CGO), pp. 2–15. ACM (2018)
30. Stone, J.E., Gohara, D., Shi, G.: OpenCL: A parallel programming standard for heterogeneous computing systems. IEEE Design and Test **12**(3), 66–73 (2010)
31. Streit, K., Hammacher, C., Zeller, A., Hack, S.: Sambamba: A runtime system for online adaptive parallelization. In: B. Franke (ed.) Proc. 21st Int'l Conf. on Compiler Construction (CC), pp. 240–243. Springer (2012)
32. Strzodka, R., Shaheen, M., Pajak, D., Seidel, H.P.: Cache accurate time skewing in iterative stencil computations. In: Proc. Int'l Conf. on Parallel Processing (ICPP), pp. 571–581. IEEE Computer Society (2011)
33. Tavarageri, S., Pouchet, L., Ramanujam, J., Rountev, A., Sadayappan, P.: Dynamic selection of tile sizes. In: Proc. 18th Int'l Conf. on High Performance Computing (HiPC), pp. 1–10 (2011)
34. Trifunovic, K., Cohen, A., Edelsohn, D., Li, F., Grosser, T., Jagasia, H., Ladelsky, R., Pop, S., Sjödin, J., Upadrasta, R.: GRAPHITE two years after: First lessons learned from real-world polyhedral compilation. In: Proc. Int'l Workshop on GCC Research Opportunities (GROW), pp. 1–13 (2010). http://ctuning.org/workshop-grow10
35. Vanhatalo, J., Völzer, H., Koehler, J.: The refined process structure tree. Data & Knowledge Engineering **68**(9), 793–818 (2009)

36. Xue, J.: Loop tiling for parallelism, vol. 575. Springer Science & Business Media (2012)
37. Yuki, T., Renganarayanan, L., Rajopadhye, S.V., Anderson, C., Eichenberger, A.E., O'Brien, K.: Automatic creation of tile size selection models. In: Proc. 8th Int'l Symp. on Code Generation and Optimization (CGO), pp. 190–199 (2010)

	S_{stat}	S_{stat_polly}	S_{dyn_delin}	EC [%]	OC [%]	variants	blocked	cache hits	$T_{stat}[s]$	$T_{stat\_polly}[s]$	$T_{dyn\_spec\_delin}[s]$
BENCHBUILD (COMPILATION, ENCRYPTION, SCIENTIFIC)											
scimark	−1	−1.0	1.0	43.3	8	4	4	0	31.5	31.4	31.6
libressl	−3.0	−2.9	1	24.1	26.0	10	10	0	5.2	5.4	15.4
lua	−2.0	−2.0	−1.0	11.9	13.1	4	4	0	19.9	20.0	40.5
BOTS											
fft	1.5	1.5	1.0	80.8	4.0	1	1	0	10.3	10.3	7.1
sparselu	−5.2	−6.8	1.3	69.5	26.5	301	98	921 000	12.5	9.6	65.2
alignment	−1.1	−1.1	1	36.4	3.7	4	2	2	23.2	23.2	26.0
floorplan	−2.0	−2.1	1.0	12.5	13.4	6	6	0	21.5	21.2	43.6
MSB (ASC_SEQUOIA, BULLET, COYOTEBENCH, TSVC)											
IRSmk	−1.0	−1.0	−1	98.2	1.5	1	1	0	3.8	3.8	3.8
Recurrences	−1.0	−1.0	1	96.9	1.6	6	6	0	8.3	8.3	8.6
StatementReordering	−1.1	−1.1	1	94.7	2.6	6	6	0	6.5	6.5	7.0
NodeSplitting	−1.1	−1.1	1	94.2	3.2	12	12	0	7.0	7.0	7.6
IndirectAddressing	6.7	6.7	1	90.7	5.3	2	2	0	6.9	6.9	1.0
Expansion	−1.1	−1.2	1	88.8	7.5	24	20	640	5.9	5.8	6.6
Equivalencing	−1.2	−1.2	1	85.1	8.3	20	20	0	3.3	3.4	4.1
LoopRestructuring	1.3	−1.3	1.1	80.0	18.8	18	6	22 400	8.5	5.1	6.8
LinearDependence	−1.3	−1.2	1.1	76.3	18.6	28	16	49 400	6.1	6.7	8.0
InductionVariable	−1.1	−1.2	−1.1	74.1	7.0	14	10	7 700	7.3	7.0	8.4
ControlFlow	−1.1	−1.2	−1.0	63.3	11.0	34	18	27 372	7.4	6.9	8.2
CrossingThresholds	−1.2	−1.1	−1	57.1	6.1	12	8	4 402	6.1	6.4	7.2
ControlLoops	−1.1	−1.1	−1.0	53.7	4.3	20	20	0	6.6	6.5	7.3
Symbolics	−1.3	−1.2	1.0	47.6	4.9	6	6	0	4.2	4.4	5.3
lpbench	−4.0	−4.0	−1.0	43.9	20.1	4	3	1	2.3	2.3	9.1
Reductions	6.3	6.3	1	42.4	3.2	4	4	0	10.9	10.9	1.7
GlobalDataFlow	−2.1	−2.1	−1.0	32.3	14.3	12	12	0	8.8	8.8	18.6
bullet	−2.0	−2.0	−1.0	13.8	14.6	22	22	0	5.7	5.7	11.4
POLYBENCH											
ludcmp	4.0	−1.1	1	99.2	0.8	2	0	2	306.0	67.8	76.2
nussinov	−1.3	−1.3	1	92.0	0.2	1	0	1	68.4	68.4	88.7
floyd-warshall	−1.1	−1.0	6.7	91.9	0.3	1	0	1	168.2	175.6	179.2
symm	−1.0	1.0	1	89.3	0.6	1	0	1	30.1	31.2	30.3
cholesky	6.0	−1.4	1.1	87.5	0.8	2	0	4 000	260.4	31.2	43.1
jacobi-2d	−1.1	−2.9	−1.2	81.1	0.6	1	0	1	35.6	13.7	40.2
heat-3d	1.7	1.7	−1.0	81.0	2.1	1	0	1	67.3	67.4	39.6
seidel-2d	4.1	4.1	1.9	74.2	0.6	1	0	1	248.3	249.0	61.0
lu	12.9	3.4	1.1	56.4	2.4	2	0	2	319.2	83.0	24.6
gemm	2.0	−1.1	2.4	32.5	2.8	1	0	1	13.2	6.0	6.5
gramschmidt	4.2	1.2	1.3	31.9	2.4	1	0	1	48.6	13.2	11.5
syr2k	6.0	−1.0	1.0	28.1	2.8	1	0	1	47.6	7.6	7.9
correlation	8.8	1.5	1.1	26.0	6.5	2	0	2	55.4	9.3	6.3
3mm	8.8	1.6	4.4	25.4	8.2	1	0	1	52.0	9.4	5.9
2mm	7.0	1.6	3.3	20.2	5.6	1	0	1	40.8	9.4	5.8
trmm	3.1	−1.1	−1.2	18.6	1.7	1	0	1	23.0	6.6	7.4
doitgen	1.1	1.1	−1.2	18.1	0.9	1	0	1	17.2	17.7	16.3
covariance	6.1	6.2	1.1	17.7	4.4	1	0	1	58.2	59.0	9.5
syrk	3.3	1.1	1.1	17.0	3.3	1	0	1	21.9	6.9	6.6
SPEC CPU2006											
470.lbm	−1.2	1.1	1.0	80.0	18.2	3	1	22	2.7	3.4	3.2
473.astar	1.5	1.5	1	28.4	0.4	1	1	0	9.0	9.0	6.2
SSB (MCGILL, SHOOTOUT, MISC)											
flops-2	−1.1	−1.1	−1	93.6	6.3	1	0	1	1.0	1.0	1.1
ary3	−1.4	−2.7	−1	57.7	5.9	2	1	1	1.2	0.6	1.6
oourafft	−4.8	−4.8	2.0	36	30.5	8	6	300 000	3.4	3.4	16.5
gramschmidt	3.1	2.3	1.2	20.0	20.9	1	0	1	3.1	2.4	1.0
flops	1.8	1.8	1	12.5	0.6	2	2	0	6.3	6.1	3.5
chomp	−2.1	−2.1	1.0	12.4	15.0	3	3	0	1.8	1.8	3.7
ReedSolomon	−2.3	−2.2	1.0	8.6	27.7	6	3	450 000	4.7	4.9	10.8

Table 2: Subject programs and benchmarks used in the evaluation of POLYJIT. A comprehensive introduction to all measures used in the table can be found in Section 4.2. S_{O3}, S_P, and S_{Jns} denote the speedup over the baseline configurations *stat*, *stat_polly*, and *dyn_delin*. EC describes the fraction of the total execution time spent inside SCoPs in %. overhead coverage (OC) describes the fraction of the total execution time spent compiling at run time in %. C count the number of times POLYJIT was able to serve a function request from its internal cache. Furthermore, we list the total execution times T_{stat}, and T_{stat_polly} of the two baseline configurations, and $T_{dyn\_spec\_delin}$ of the configuration *dyn_spec_delin* in seconds.

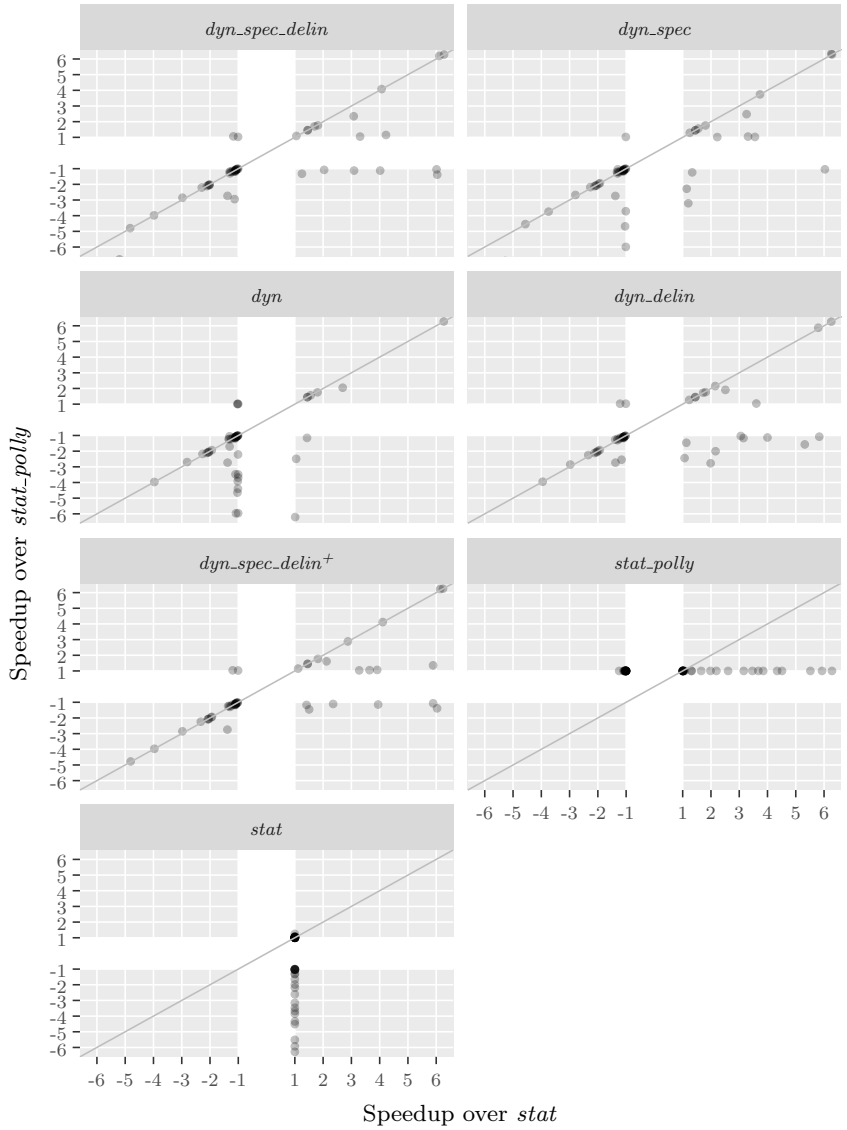

Figure 6: Faceted two-dimensional speedup plot. The horizontal axis shows the speedup of the given configuration over *stat*. The vertical axis shows the speedup of the given configuration over *stat_polly*. The dark grey area marks a zone of invalid values, because the speedup metric is not continuous in the range between –1 and 1.