SPolly: Speculative Optimizations in the Polyhedral Model

Johannes Doerfert
Saarbrücken Graduate School of Computer Science
Saarland University
Saarbrücken, Germany
doerfert@st.cs.uni-saarland.de

Clemens Hammacher
Saarbrücken Graduate School of Computer Science
Saarland University
Saarbrücken, Germany
hammacher@cs.uni-saarland.de

Kevin Streit
Saarbrücken Graduate School of Computer Science
Saarland University
Saarbrücken, Germany
streit@cs.uni-saarland.de

Sebastian Hack
Computer Science Department
Saarland University
Saarbrücken, Germany
hack@cs.uni-saarland.de

ABSTRACT

The polyhedral model is only applicable to code regions that form static control parts (SCoPs) or slight extensions thereof. To apply polyhedral techniques to a piece of code, the compiler usually checks, by static analysis, whether all SCoP conditions are fulfilled. However, in many codes, the compiler fails to verify that this is the case. In this paper we investigate the rejection causes as reported by *Polly*, the polyhedral optimizer of a state-of-the-art compiler. We show that many rejections follow from the conservative overapproximation of the employed static analyses. In *SPolly*, a speculative extension of Polly, we employ the knowledge of runtime features to supersede this overapproximation. All speculatively generated variants form valid SCoPs and are optimizable by the facilities of Polly. Our evaluation shows that SPolly is able to effectively widen the applicability of polyhedral optimization. On the SPEC 2000 suite, the number of optimizable code regions is increased by 131 percent. In 10 out of the 31 benchmarks of the PolyBench suite, SPolly achieves speedups of up to 11-fold as compared to plain Polly.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers, Optimization, Retargetable compilers, Run-time environments*

General Terms

Performance

Keywords

Adaptive optimization, polyhedral loop optimization, just-in-time compilation, speculative execution

1. INTRODUCTION

The polyhedral model uses polyhedra to abstract essential information of a static control program (SCoP, a restricted form of nested DO loops). By abstracting loop nests as polyhedra, many popular loop optimizations can be elegantly formulated in terms of linear transformations. However, a piece of code has to meet the SCoP criteria [5] (or slight extensions thereof) to be tractable by polyhedral techniques. Those conditions are usually met by many numerical kernels from linear algebra, image processing, signal processing, etc. However, in more general benchmarks such as the SPEC 2000 benchmark suite, the SCoP conditions are often violated. One simple reason is, for example, that the compiler could not prove that all arrays used in the loop nest do not overlap or alias. Further reasons we encountered are: The implementation of functions called in the loop nest was not available because they are extern to the translation unit in question. However, at runtime, the functions turned out to be pure. Also, index expressions and/or loop bounds could not be classified as affine. Often, some parameter, which does not change in the loop nest, makes an expression non-affine. Hence, specializing the loop nest to the concrete value at runtime makes polyhedral techniques applicable.

In this paper, we want to advance the applicability of polyhedral optimizations by incorporating runtime information. For example, parameters that lead to non-affine expressions can be replaced by their concrete value. This way, using a just-in-time compiler, the program can prepare specially optimized versions for different parameter sets. In summary, this paper makes the following contributions:

- We investigate the applicability of Polly [8], a polyhedral framework for LLVM [10], on the SPEC 2000 benchmark suite and classify the causes that prevent the application of Polly to a candidate code region[1]. We conclude that up to 2.4 times more code regions in SPEC 2000 could be considered by Polly if runtime information is available.

- Based on those insights, we present *SPolly,* a prototypical extension of Polly to integrate runtime information. At the moment, SPolly handles two cases: First, it adds code to the program that checks whether all

[1]As code regions we consider all non-trivial regions according to Grosser et al. [8], i.e. single entry, single exit regions containing at least one loop.

referenced arrays of a loop nest do not overlap. Second, it uses profiling to record the values of parameters that lead to non-affine index expressions and/or loop bounds and generates specialized variants that can be selected during runtime. We demonstrate the benefit of these techniques on the PolyBench benchmark suite.

The rest of this paper is organized as follows: In Section 2 we present related work. Section 3 provides an evaluation of the most frequent rejection causes of polyhedral transformations; Section 4 explains how SPolly alleviates some of those causes. Finally Section 5 evaluates SPolly and Section 6 concludes.

2. RELATED WORK

The polyhedral literature is certainly too extensive to be discussed in full detail here. Furthermore, this paper is concerned with increasing the applicability of the polyhedral model by speculation and runtime analysis and not with inventing new polyhedral techniques. Therefore, we concentrate on the recent work which is directly related to ours.

In his thesis [7], Grosser describes a speedup of up to 8x for a matrix multiplication benchmark, achieved by his polyhedral optimizer Polly [8]. He also produced similar results for other benchmarks of the PolyBench [11] benchmark suite. Other publications [4, 1, 12] show similar results on PolyBench. PolyBench is certainly well suited to compare polyhedral techniques. However, it does not allow for assessing their width of applicability as we do in this paper.

Baghdadi et al. [1] reveal a huge potential for speculative loop optimizations as an extension to the formerly described techniques. They state that aggressive loop nest optimizations (including parallel execution) are profitable and possible, even though overestimated data and flow dependences would statically prevent them. Their manually crafted tests also show the impact of different kinds of conflict management. Overhead, and therefore speedup, differs from loop to loop, as the applicability of such conflict management systems does, but a trend was observable. The best conflict management system has to be determined per loop and per input, but all can provide speedup, even if they are not that well suited for the situation.

Bastoul et al. [2, 3] and Girbal et al. [6] also perform an evaluation of the applicability of the polyhedral model in several benchmarks. In contrast to this work however, he is more concerned with the structure and size of the SCoPs and not with the reasons of why other code regions could not be classified as SCoPs.

Jimborean et al. [9] employ the polyhedral model in the context of speculative parallelization. A static parallelization pattern is generated assuming linearity of loop bounds and memory accesses. Additionally, polyhedral transformations are proposed based on static dependence analysis. At runtime, the linearity of memory accesses and the applicability of the proposed candidates is assessed by collecting profiling information. In case of success, the statically generated pattern is instantiated with the gathered information and corresponding code is generated. As the performed transformations are speculative and not validated before execution of the loop, a speculation mechanism guards the execution. In case of detected conflicts, rollbacks are performed and execution switches to the safe, sequential version. The approach has some drawbacks: First of all, it relies on programmer annotations to identify parallelization candidates. Second, polyhedral optimization is limited to transformations that do not change the loop structure or reorder statements. The use of synthetic benchmarks in their evaluation does not allow judging in how far the approach can profitably extend the applicability of polyhedral optimization on widely used benchmarks such as the PolyBench suite.

3. MOTIVATION

In order to analyze which of the SCoP restrictions limit the applicability of the polyhedral model most, we conducted an evaluation on nine programs from the SPEC 2000 benchmark suite[2]. To this end we instrumented Polly to output all rejection causes that prevent a candidate code region from being considered a polyhedron[3]. In case of multiple rejection causes for one region, we record all of them.

Figure 1 shows the results of this evaluation. For each rejection cause in Polly, we report three numbers:

- The number of regions where the violation of condition i is a cause for not considering (Column A).
- The number of regions where the violation of condition i is the *only* rejection cause (Column B).
- The number of regions gained when ignoring all conditions 0 to i (Column C).

Over the nine tested programs, 1862 candidate regions are tested, out of which 1587 are rejected by Polly. The remaining 275 regions (14.8%) are considered legal SCoPs.

i	Rejection cause	A	B	C
0	Non-affine expressions	1230	84	84
1	Aliasing	1093	207	510
2	Non-affine loop bounds	840	6	660
3	Function call	532	72	928
4	Non-canonical indvars	384	0	1174
5	Complex CFG	253	31	1387
6	Unsigned comparison	199	0	1586
7	Others	1	0	1587

Figure 1: SCoP rejection causes on the SPEC 2000 benchmark suite

The rejection causes are ordered by the number of regions that they affect (column A). In the following, we will examine the conditions constituting the rejection causes in further detail and discuss if and how SPolly can alleviate their impact.

Non-affine expressions As explained in more detail in Section 4.2, all expressions used for memory accesses or predicates of conditional branches have to be affine expressions with respect to parameters and induction variables. If this is not the case, the code cannot be represented as polyhedron, thus preventing corresponding optimizations. This happens for example

[2]As SPolly's runtime environment is based on the Sambamba framework [13] we selected the programs that were contained in the Sambamba test suite: ammp, art, bzip2, crafty, equake, gzip, mcf, mesa, and twolf

[3]All programs were compiled with
`-sink -indvars -mem2reg -polly-prepare`

when a programmer chose to represent a 2-dimensional array by a 1-dimensional, flattened, one, translating the index (i, j) to $i * N + j$. In case i and j are iteration variables, and N is a parameter, this expression is not affine but quadratic. If however N can be detected as quasi-constant via profiling, it can speculatively be replaced by that constant to circumvent the non-affinity.

Aliasing Possible aliasing causes a region to be rejected whenever the base address of two memory accesses cannot be proven to be disjoint. In particular, this is the case when pointers originate in parameter values instead of global variables or stack-allocated memory, since the default alias analysis used by Polly only works intra-procedural. In most cases however, two arrays passed as parameters are disjoint; speculatively assuming non-aliasing may thus be profitable.

Non-affine loop bounds This constraint requires all loop bounds to be affine expressions with respect to the parameters and surrounding iteration variables. In our experiments we frequently observed that although bounds have not been affine at compile-time, they often turn out to remain constant, and thus affine, during all executions. This is for example the case if a loop iterates N^2 times, where N is a parameter. It is obvious that in these cases a specialized variant can be generated where the loop bounds are considered constant. This not only makes the loop representable as a polyhedron, but also enables better optimizations in the *isl*.

Function call Another major reason for rejecting a code region is contained function calls. In general, computing memory dependences through calls is a hard task; for external functions or indirect calls it is often impossible. Additionally, external functions may not terminate or have observable effects like text output or aborting the application. Polly rejects each region containing at least one call to a function that cannot be proven to be pure. In practice though, parallelization prohibiting side-effects—for example exceptions, or error reporting—might manifest only infrequently. In such case, speculatively ignoring the calls and specializing a region accordingly can make it amenable to polyhedral optimizations. To preserve the original semantics, the specialized code needs protection by a runtime speculation mechanism, e.g. in the form of transactional memory.

Non-canonical induction variables This constraint requires the induction variables to be in a canonical form, starting at zero and being incremented by one in each iteration. If LLVM's and Polly's preparation passes are unable to canonicalize all such variables, the code region is discarded.

Complex CFG This error is reported if complex terminators like switch instructions or are found, or the control flow has a "complex form" not representable via *while* and *if* constructs only.

Unsigned comparison During polyhedral optimizations, Polly may have to modify comparison operators or their operands, for example to alter the iteration space of a loop. Special care has to be taken to handle possible overflows correctly. To this end, this is only implemented for signed operations. Consequently, Polly rejects all code regions containing unsigned comparison.

Others This is a collection of minor technical limitations, for example rarely used LLVM instructions, which are not handled properly yet. The only occurrence in our tests is a cast from an integer to a pointer in the mcf program.

SPolly concentrates on the first three rejection causes. In our benchmarks, they make up for 42% of all SCoP rejections. Thus, assuming we could speculatively eliminate them in all cases, we could expect 660 new SCoPs to be detected, which is exactly 2.4 times the original number of SCoPs. The fourth cause, function calls, could be solved using runtime techniques as described, but in our experiments the introduced overhead of transactional execution did not pay off. Therefore we skipped that for now, and consider it future work to improve the performance of the transactional memory system. All other reported causes are either conceptual obstacles (induction variables, complex CFGs) that SPolly cannot remove or technicalities that will disappear when LLVM and Polly will mature further.

4. SPOLLY

SPolly extends the applicability of the loop nest optimizer and parallelizer Polly by deploying runtime information and speculatively specializing functions. It targets restrictions on loop nests arising as a consequence of overestimations in static analyses. By inferring common values for parameters, and providing additional conditions for memory accesses, it makes polyhedral optimizations applicable to more code locations, and allows for better optimization and code generation. Those specialized versions will coexist with the original sequential version and will be executed whenever the actual runtime values permit this. This section describes in detail how SPolly achieves this.

4.1 Possible Aliasing

Possible aliasing is the main rejection cause of Polly on the considered benchmarks of the SPEC 2000 suite and particularly well suited for speculation. An example for a loop with possibly aliasing accesses is given in Figure 2a. It shows two arrays accessed via pointers `A` and `B`. In case they point to addresses less than `N * sizeof(int)` bytes apart, parallel execution and other loop transformations possibly alter the semantics. Most alias analyses are conservative and assume aliasing if `A` and `B` could potentially point to the same allocated memory block. Using the latter definition, Polly would not optimize the presented loop without further information on `A` and `B`.

Polly offers the possibility to override conservative assumptions concerning possible aliasing. This can be done by either ignoring all aliasing, or by annotating individual code regions. Both these approaches require manual intervention and profound knowledge of the application to optimize. In contrast, SPolly is able to deal with possible aliases without any programmer-provided information. It does so by introducing alias checks preceding the subject loop nests to

```
void a(int *A, int *B) {
  int i;
  for (i=1; i<N; i++)
    A[i] = 3 * B[i];
}
```

(a) Loop with possible aliasing pointers

```
void b(int *A, int N) {
  int i, j;
  for (i = 1; i<N; i++)
    for (j = 0; j<N; j++)
      A[j*N+i] += A[j*N+i-1];
}
```

(b) Loop nest using non-affine expressions

Figure 2: Example loops rejected by Polly for different speculatively resolvable reasons

ensure the absence of conflicts between accesses to loop invariant base pointers. For the shown loop, those introduced checks are conclusive as the accessed range, relative to the base address, is known before entering the loop, thus non-aliasing of all accesses to the arrays can be checked a priori. This approach allows to optimize loops even if the used base pointers might point into the same allocated block, for example different parts of the same array. If the checks fail, the original, unmodified version is executed, otherwise the optimized version is chosen. An example that would not benefit from this approach because of possibly aliasing loop variant pointers is dynamic multidimensional arrays (arrays of pointers to arrays). Iterating over those will change the base pointer of the inner dimension for each execution of the outer loop, and checking that none of them alias is too expensive to be performed prior to execution of the loop.

4.2 Non-affine Parameters

Consider Figure 2b. C does not provide support for n-dimensional arrays of which the dimensions are not known statically. Hence, a common pattern is to implement n-dimensional arrays using a 1D array and performing the index arithmetic "manually". This pattern creates non-affine array subscripts on the 1D array. Using static analysis, one could infer that the 1D access is actually a 2D access; at least in this example. However, such an analysis is currently not implemented in Polly. Additionally, it is imaginable that the code is more complicated making the static analysis unsuccessful.

Thus, SPolly utilizes runtime information gained with a profiling version of the loop nest to identify reoccurring parameter values. For those values specialized loop versions are created with constant values plugged in for the problematic parameters. Dispatching code is inserted before the corresponding loop to decide at runtime whether a specialized version exists for the actual parameter values. If none is found, the original, less optimized version is used. In our example, by specialization the multiplications would become affine and therefore representable in the polyhedral model, and thus amenable for all polyhedral optimizations implemented in Polly.

Finally, we observed (see the next section) that replacing parameters with constants often leads to superior code because it enables more aggressive optimizations. Thus, even if a static analysis would analyze pseudo-1D array accesses and restate them into multidimensional accesses, executing specialized versions often leads to better performance.

5. EVALUATION

In order to evaluate the success of incorporating runtime information to speculatively optimize code regions, we will investigate two hypotheses:

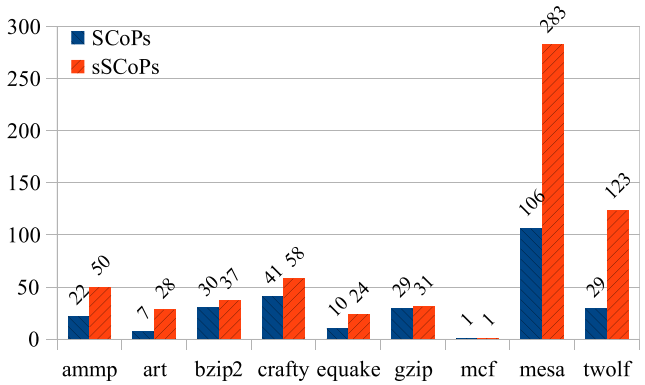

Figure 3: Quantitative analysis of the applicability of SPolly compared to Polly. Overall, SPolly provides 635 sSCoPs while Polly find 275 SCoPs.

1. SPolly is able to handle substantially more regions than Polly. In other words, the number of speculative SCoPs (*sSCoPs*) is substantially larger than the number of SCoPs.

2. The extended applicability of the polyhedral model results in improved runtime performance.

Hypothesis 1 is motivated from Section 3, where we showed that the constraints that SPolly tackles are the main reasons for rejecting a SCoP. Hypothesis 2 follows from the primary goal of polyhedral optimizations, which is reducing the program runtime.

5.1 Extended Applicability

In order to evaluate the successful elimination of SCoP rejection causes, we executed SPolly on the SPEC 2000 tests that we already used in Section 3. We then compared the number of sSCoPs detected by SPolly against the number of SCoPs detected by Polly. A graphical representation of this comparison is shown in Figure 3.

You can see that for all test cases except of "mcf", our speculative extensions provided an increased number of code regions amenable to polyhedral optimizations. The overall number of sSCoPs is 635, as compared to the 275 detected SCoPs by Polly. This is an **increase of 131 percent** (360 additional SCoPs).

Compared to the expectations in Section 3, where we identified an upper bound of 660 additional SCoPs, we reached a **success rate of 55 percent**. The remaining sSCoP candidates contain code where our heuristics decided that speculation is not profitable or impossible. This is the case if,

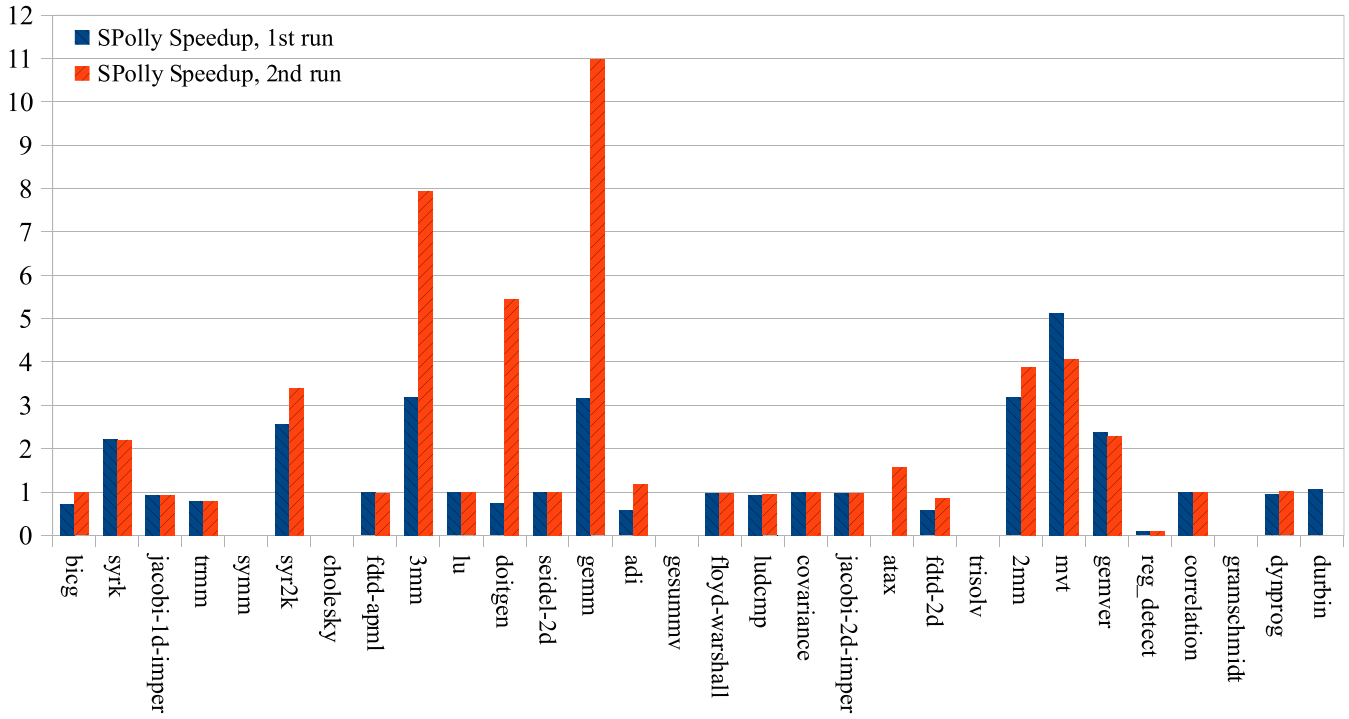

Figure 4: Speedups in execution time of SPolly-enabled programs on the PolyBench suite, normalized to the standard Clang-compiled version with all optimizations enabled

for example, an external function call is executed unconditionally, or indirect pointer loads are detected where aliasing cannot be checked a priori.

Overall, we conclude that SPolly in fact widens the applicability of polyhedral optimizations substantially.

5.2 Performance

After confirming that the number of SCoPs detected in SPEC 2000 is indeed increased, we investigated to which extent this actually improves the runtime of the programs. We observed that even though the additional SCoPs were valid and some of them were executed, the most interesting code regions were not optimized due to additional obstacles we could not eliminate. These are mainly external function calls and indirect memory accesses in the hot loops. So on these tests, we are unable to achieve significant speedups.

Thus we resort to the PolyBench suite [11], which has also been used by Grosser et al. [8, 7] before to evaluate the performance of Polly. We are using the current version 3.2, and the provided "large" data sets.

In order to compare the speedup in program execution achievable by applying polyhedral optimizations on statically validated SCoPs alone against that of speculatively created and optimized polyhedra, we run all tests

- on the standard toolchain of Clang and LLVM with all optimizations enabled
- using Polly after applying the standard preparation passes (see Section 3)
- using SPolly without any prior knowledge about the program
- using SPolly a second time, such that profiling information from the previous run is already available

All tests are conducted on an eight-core Intel Xeon machine running at 2.93 GHz with 24 GB of main memory.

Surprisingly, we are unable to reproduce earlier results of Polly on the PolyBench suite. Investigating the reasons why Polly rejects the SCoPs for the compute-intensive kernels of all of the benchmarks revealed, that in all cases this is due to aliasing problems. Since PolyBench 3.0, released in October 2011, the tests are using heap-allocated arrays, passed to the kernel via pointer arguments plus array sizes. Polly is unable to prove non-aliasing of those pointers, thus rejecting all possibly affected SCoP candidates. This is why we did not find any program for which Polly was able to improve the performance over a standard Clang-compiled program.

Figure 4 shows the speedups of the SPolly-enabled program runs, normalized to the runtime of the corresponding Clang-compiled program. Missing values in this bar chart indicate failures during optimization, which were mainly originating from the CLooG code generator, but also from creating the polytope in Polly. Nevertheless, for most of the programs we are able to report runtime results. It is not surprising that there are many programs where SPolly is not able to bring the kernel to a form amenable for polyhedral optimizations. Thus for these programs no speedup is achieved.

However, there are also different programs where SPolly indeed provides enormous speedups. For the first run, SPolly can make no use of any profiling data, so only those speculative transformations can be applied for which conclusive runtime checks can be synthesized. For these runs, the highest achieved speedup is 5.1-fold on the mvt program. In

the second run however, specialized versions can be created based on the profiles gathered in previous runs. This allows to replace loop bounds and parametric values by constant expressions, which not only makes those code regions representable as polyhedra, but also helps other transformations and the CLooG backend to create better code, e.g. by choosing the best tile size for loop tiling. Thus, the speedups achieved in the second runs are significantly higher in almost all cases, ranging to a maximum of 11-fold for the "gemm" program. In two cases (mvt and gemver), the speedup in the second run is slightly lower than in the first run. This is not due to overhead we introduce for checking whether a specialized code version can be chosen (this overhead was negligible for all our tests), but because the specialized variant actually executed slower than the original one. The problem of anticipating whether a code transformation, especially parallelization, will pay of at runtime is a well-known problem for which no general solution exists. Apart from that, it's not in the scope of this paper.

6. CONCLUSION

In this paper, we analyzed the most prominent causes that prohibit polyhedral optimizations for individual code regions. We observed that often promising regions are rejected by the polyhedral framework Polly because of overapproximation in the static analyses, and because of missing parameter values. Driven by this observation, we came up with runtime checks to dynamically switch to specialized variants of a code region, where the obstructive feature is removed. This approach is implemented in SPolly, a speculative extension to Polly. It enables the application of polyhedral optimizations to many more code locations, providing better runtime performance in those cases where the specialized variant could be used.

7. REFERENCES

[1] R. Baghdadi, A. Cohen, C. Bastoul, L.-N. Pouchet, and L. Rauchwerger. The potential of synergistic static, dynamic and speculative loop nest optimizations for automatic parallelization. In *Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures (PESPMA'10)*, June 2010.

[2] C. Bastoul. *Improving Data Locality in Static Control Programs*. PhD thesis, University Paris 6, Pierre et Marie Curie, France, 2004.

[3] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, LNCS, pages 23–30. Springer-Verlag, Oct. 2003.

[4] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, 2008.

[5] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[6] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006.

[7] T. Grosser. *Enabling Polyhedral Optimizations in LLVM*. Diploma thesis, University of Passau, Apr. 2011.

[8] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Grösslinger, and L.-N. Pouchet. Polly - Polyhedral Optimization in LLVM. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Apr. 2011.

[9] A. Jimborean, L. Mastrangelo, V. Loechner, and P. Clauss. VMAD: an Advanced Dynamic Program Analysis & Instrumentation Framework. In *CC - 21st International Conference on Compiler Construction*, pages 220–237, Mar. 2012.

[10] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Mar 2004.

[11] L.-N. Pouchet. Polybench, the Polyhedral Benchmark suite. `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`, 2010.

[12] B. Pradelle, A. Ketterlin, and P. Clauss. Polyhedral parallelization of binary code. *ACM Transactions on Architexture and Code Optimization*, 8(4):39:1–39:21, Jan. 2012.

[13] K. Streit, C. Hammacher, A. Zeller, and S. Hack. Sambamba: A runtime system for online adaptive parallelization. In *Proc. 21st International Conference on Compiler Construction (CC)*, pages 240–243, Mar. 2012.

APPENDIX

	clang	clang -O2	clang -O3	SPolly 1st run	SPolly 2nd run	Speedup *1st* run	Speedup *2nd* run
bicg	0.66	0.22	0.22	0.32	0.23	0.72	0.99
syrk	49.03	12.16	12.16	5.48	5.50	2.21	2.21
jacobi-1d-imper	0.95	0.23	0.23	0.25	0.25	0.93	0.93
trmm	24.87	6.44	6.44	8.08	8.09	0.80	0.80
symm	129.49	119.75	119.75				
syr2k	86.57	25.76	25.76	10.03	7.56	2.57	3.40
cholesky	6.71	1.68	1.68				
fdtd-apml	12.42	10.69	10.69	10.84	10.86	0.98	0.98
3mm	294.77	233.01	233.01	72.96	29.30	3.19	7.95
lu	19.18	4.53	4.53	4.58	4.59	0.99	0.99
doitgen	45.07	10.95	10.95	14.95	2.00	0.73	5.46
seidel-2d	1.30	1.12	1.12	1.12	1.12	1.00	1.00
gemm	107.93	77.48	77.48	24.49	7.05	3.16	10.98
adi	12.56	9.31	9.31	16.10	7.88	0.57	1.18
gesummv	0.64	0.24	0.24				
floyd-warshall	72.31	13.70	13.70	14.05	14.05	0.98	0.98
ludcmp	31.62	20.94	20.94	22.31	22.26	0.94	0.94
covariance	71.80	64.70	64.70	64.78	64.88	1.00	1.00
jacobi-2d-imper	1.23	0.48	0.48	0.49	0.49	0.97	0.97
atax	0.81	0.19	0.19		0.12		1.57
fdtd-2d	5.26	2.07	2.07	3.63	2.43	0.57	0.85
trisolv	0.20	0.05	0.05				
2mm	206.63	155.09	155.09	48.75	39.92	3.18	3.88
mvt	1.69	1.18	1.18	0.23	0.29	5.13	4.06
gemver	2.33	1.30	1.30	0.55	0.57	2.36	2.28
reg_detect	0.43	0.06	0.06	0.80	0.72	0.08	0.09
correlation	71.81	64.73	64.73	64.80	64.75	1.00	1.00
gramschmidt	159.34	164.49	164.49				
dynprog	167.63	65.87	65.87	69.83	65.07	0.94	1.01
durbin	2.23	2.07	2.07	1.96		1.06	

Figure 5: Runtime results on the PolyBench suite, comparing clang with different optimization levels and SPolly. Empty cells represent runs which could not be completed due to technical issues with the used frameworks.