Switchable Scheduling for Runtime Adaptation of Optimization

Lénaïc Bagnères[1] and Cédric Bastoul[2]

[1] University of Paris-Sud and Inria
lenaic.bagneres@inria.fr
[2] University of Strasbourg and Inria
cedric.bastoul@unistra.fr

Abstract. Parallel applications used to be executed alone until their termination on partitions of supercomputers: a very static environment for very static applications. The recent shift to multicore architectures for desktop and embedded systems as well as the emergence of cloud computing is raising the problem of the impact of the execution context on performance. The number of criteria to take into account for that purpose is significant: architecture, system, workload, dynamic parameters, etc. Finding the best optimization for every context at compile time is clearly out of reach. Dynamic optimization is the natural solution, but it is often costly in execution time and may offset the optimization it is enabling. In this paper, we present a static-dynamic compiler optimization technique that generates loop-based programs with dynamic auto-tuning capabilities with very low overhead. Our strategy introduces switchable scheduling, a family of program transformations that allows to switch between optimized versions while always processing useful computation. We present both the technique to generate self-adaptive programs based on switchable scheduling and experimental evidence of their ability to sustain high-performance in a dynamic environment.

1 Introduction

Static compilers are facing the challenge of generating efficient codes for increasingly dynamic execution environments. Two decades ago, optimizing compilation was referred as building "supercompilers for supercomputers" [20]. Compiler techniques had to optimize aggressively for complex parallel machines but in a very static context: usually one program with few dynamic parameters, one well defined architecture/system and one user. Iterative compilation and auto-tuning approaches have been developed on top of static compilation as efficient solutions to find the best optimization parameters and to adapt to various (but fixed) architectures and problem sizes [2, 19, 12]. The large adoption of multicore systems and the emergence of cloud computing brings new dynamic factors that are not captured by iterative compilation or auto-tuning, such as the existence of competing workloads or the possible migration of the process to another architecture. This situation raises the need for more dynamic optimization schemes.

Just in time compilation is a convenient solution to address dynamic execution environments. However, it requires very low algorithmic complexity of the underlying techniques to avoid to offset the optimization it is enabling. Current state-of-the-art static automatic optimization and parallelization techniques rely on an algebraic representation of programs that allows precise analyses as well as very aggressive program transformations to optimize codes, known as the *polyhedral model* [7, 3, 11]. Unfortunately, most polyhedral-based techniques show exponential complexity [17]. Hence, they are challenging to include in a dynamic compilation framework, except when a runtime analysis allows to use this model while it was not possible at static compile time [9]. Our proposal is a mixed static-dynamic technique, which benefits from the power of polyhedral frameworks at static compile time, while being able to change the optimization decision at runtime during the computation itself.

The potential benefit of such a technique is significant because the dynamic nature of the execution environment comes from several factors that directly impact performance. First of all, a compiled program may be run on different architectures with different features such as various cache memories or number of cores, which have dramatic impact on the best optimization choice. A decision at the early stage of the execution is not enough: virtual machines and cloud computing technology allow the architecture to change during execution. Next, the application may depend on dynamic parameters such as problem size (e.g., array size). Hence the best optimization is likely to be different depending on those parameters that will be known only at runtime. Finally, the operating system and the system workload are also paramount because processes may affect each other, e.g., through cache pollution or by stressing the system scheduler.

Our approach is to design at compile-time programs that can adapt at runtime to the execution context. The originality of our solution is to rely on *switchable scheduling*, a selected set of program restructuring which allows to swap between program versions at some meeting points without any rollback. A first step selects pertinent switchable versions according to their performance behavior on some execution contexts. The second step builds a self-adaptive program including selected versions. Then at runtime the program keeps choosing the best version thanks to a low overhead sampling and profiling of the versions, ensuring during the process that every computation contributes to the final result. We performed an experimental study on dozens of execution contexts and demonstrate superior adaptability of our generated codes with respect to state-of-the-art static optimization technique.

2 Background

The application domain of our technique is loop-based kernels with affine control and memory accesses, i.e., such that loop bounds, conditions and array subscripts are affine forms of outer loop counters and constant parameters. This class of computational kernels is known as SCoPs for Static Control Parts. SCoPs can be modeled using an algebraic representation called the *polyhedral model*. Because of the restriction on the input program form, each dynamic *instance* of a given

SCoP statement can be modeled as an integer point in a union of polyhedra called the *iteration domain* of that statement. For example, let us consider the input code in Figure 1(a). Figure 1(b) shows the iteration domain of the statement `S(i,j)`. Each loop enclosing the statement in the code corresponds to a dimension of the domain. Several compilers have the ability to raise SCoPs to a polyhedral form such as *GNU GCC*[3] and *LLVM*[4].

Once a SCoP is raised to the polyhedral model, an optimizer can compute a *scheduling* by means of *scheduling relations* that express logical execution dates for all statement instances, e.g., to achieve data locality or to expose parallelism while satisfying data dependences. In the following, we will only consider scheduling that does not alter the original program semantics. Figures 1(c1) and 1(c2) show two different possible scheduling relations. They map original *input* dimensions, which express original statement instances, to target *output* dimensions, which express their new order. Scheduling relations are expressive enough to encode a complex composition of program transformations (including, e.g., loop interchange, fusion, fission, skewing, tiling etc.) [8]. Those in Figures 1(c1) and 1(c2) correspond respectively to the identity transformation and to the reversal of the inner loop. Many efficient scheduling algorithms have been designed, notably the Pluto algorithm for automatic optimization and parallelization [3] and the Letsee technique based on iterative optimization [11].

Finally a code generator for scanning polyhedra such as CLooG [1] can produce a syntactic program that implements the new scheduling from the iteration domains and the scheduling relations. Figures 1(d1) and 1(d2) present the programs generated back from the corresponding polyhedral representations after the code generation step. The complete Figure 1 summarizes the usual workflow of a polyhedral framework with two different scheduling relations that result in two *versions* of the input program. Most previous works aim at finding only *one* good version. Our work improves this scheme with dynamic capablities, to be able to chose the right version for the right execution context.

3 Switchable Scheduling

In a polyhedral compilation framework, a program version is generated from the input program information and a scheduling. The scheduling is in turn expressed as a list of scheduling relations, one for each statement. In this work, we focus on particular sets of scheduling called *switchable scheduling*. Two scheduling are switchable if and only if there exist meeting points in the corresponding generated versions such that it is possible to continue the execution from any of these versions at those meeting points without affecting the program result. Translated to the polyhedral model terminology, it means that there must exist a couple of logical dates called *switching dates*, one for each scheduling, such that the sets of instances that have been scheduled prior to these dates in each version is the same, regardless of their respective order. To simplify their computation,

[3] http://gcc.gnu.org/wiki/Graphite
[4] http://polly.llvm.org

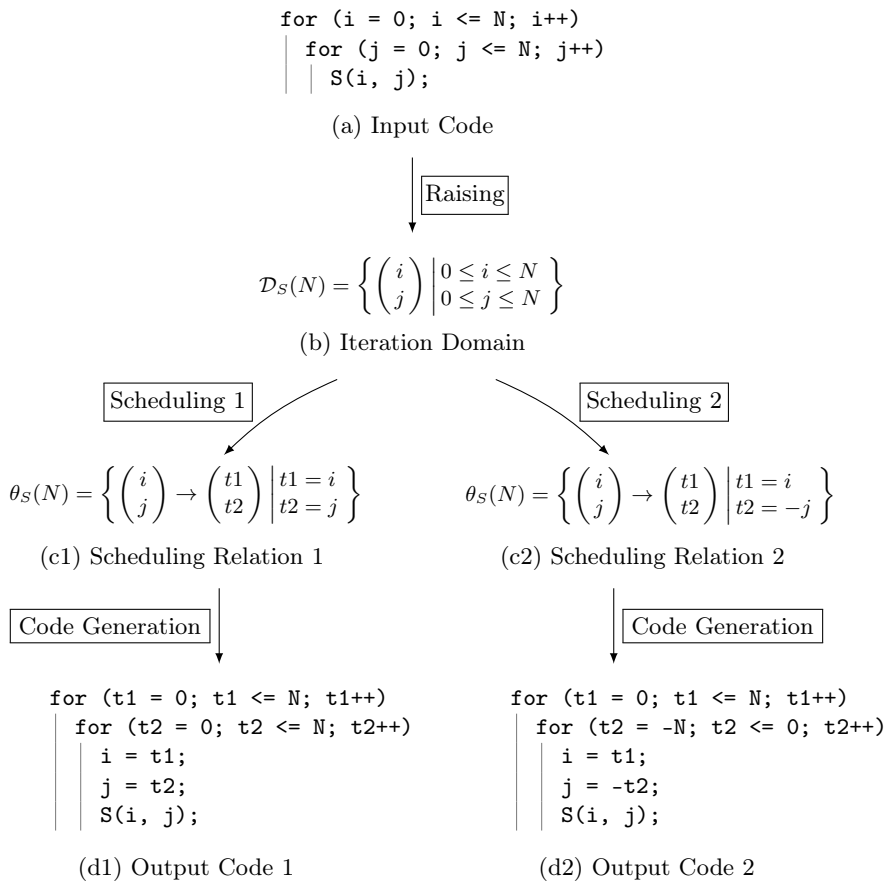


Fig. 1: Polyhedral Transformation Workflow For Two Example Versions

and without loss of generality, we require that switching dates correspond to existing instance schedules. The set of switching dates for a scheduling θ to a scheduling θ' is called its *switching domain to* θ'.

Property 1. To a given switching date in a scheduling there may exist only a unique corresponding switching date in another scheduling.

Explanation. Each instance of the original program has a unique image in the target program. Hence, given a set of already executed instances before a meeting point in a version, the corresponding meeting point in another version, if it exists, is the unique instance that will be executed directly after that set. □

Property 2. If the outermost dimensions of two scheduling are mapping input dimensions in the same order, then the first instance scheduled at any value of these outermost dimension belongs to the switching domain of the corresponding scheduling to the other scheduling.

Explanation. Logical dates are multidimensional like clocks: the first dimension may correspond to days (most significant) then the next one to hours (less significant), then the next one to minutes and so on. To each value of the outermost scheduling dimensions corresponds a set of scheduled instances. If the execution order of such sets is the same in any version, then at the beginning of each set it is possible to switch between versions, regardless of the scheduling order inside the set, i.e., of less significant scheduling dimensions. □

From these two properties we derive a practical technique to build a multi-version code. First for each version we compute a switching domain, as detailed in Section 3.1. Next we generate the code itself, inserting switching statements for each integer point of the switching domains, as explained in Section 3.2. Switching statements themselves rely on a low overhead runtime system described in Section 3.3.

3.1 Switching Domain Computation

We derive from Property 2 that a (subset of) the switching domain is the set of output vectors such that:

1. The outermost "common" output dimensions are expressed in the same way for every scheduling (this ensures that all versions are executing equivalent subsets of instances in the same order regardless of the order inside those subsets). This condition may be relaxed when information about the scheduling semantics is available. The most important case we are supporting is strip-mining and, by extension, tiling, with a restriction on possible tile sizes. Tile sizes are chosen to be a multiple of the smallest tile size. Hence, we know statically that, e.g., an iteration at a given dimension in one version corresponds to n iterations of the same dimension in another version. We derive from this a simple affine constraint on the existence of meeting points.
2. The remaining output dimensions are set to the lexicographic minimum of the possible values (to ensure the logical date of the switching statement is at most the same as the first instance scheduled inside the subset). Moreover, we add another output dimension set to 0 to ensure the switching statement is executed before the first instance of the subset.

Switching domains are easy to compute from the scheduling using the PIP tool [6] to compute the lexicographic minimum of the innermost output dimensions. Figures 2(d1) and 2(d2) show the switching domains corresponding to the scheduling in Figures 1(c1) and 1(c2): the first dimension has the same expression in both scheduling and has the same range, the second one is set to the minimum value for each version, and a new one has been added and set to 0.

The code generation step detailed in Section 3.2 uses switching domains to insert "switching statements" in the final code: to each integer point in this domain will correspond an execution of the switching statement. It is not desirable to execute the switching statement at each meeting point because of the overhead it may introduce. Switching domains can be easily restricted to fit the

need. A first solution is to intersect it with a convenient lattice. In this way, switching statements will be executed at constant intervals along scheduling dimensions. A second solution with the same effect is to apply a special strip-mine onto some scheduling dimensions. In this case, selected scheduling dimensions are decoupled into three dimensions in the switching domains and the scheduling relations. The outer dimension iterates over strips, the middle one is set to 0 for the switching domain and to 1 for all the scheduling relations, and the inner one is set to 0 for the switching domain and iterates over integer points inside strips for the scheduling relations. This does not affect the order of the instances, but it inserts a switching date before each strip. While the first solution is simpler, the second one allows to consider switching along parallel dimensions: the dimension over strips has to be sequential, but the one over points inside strips may be parallel.

3.2 Multi-Version Code generation

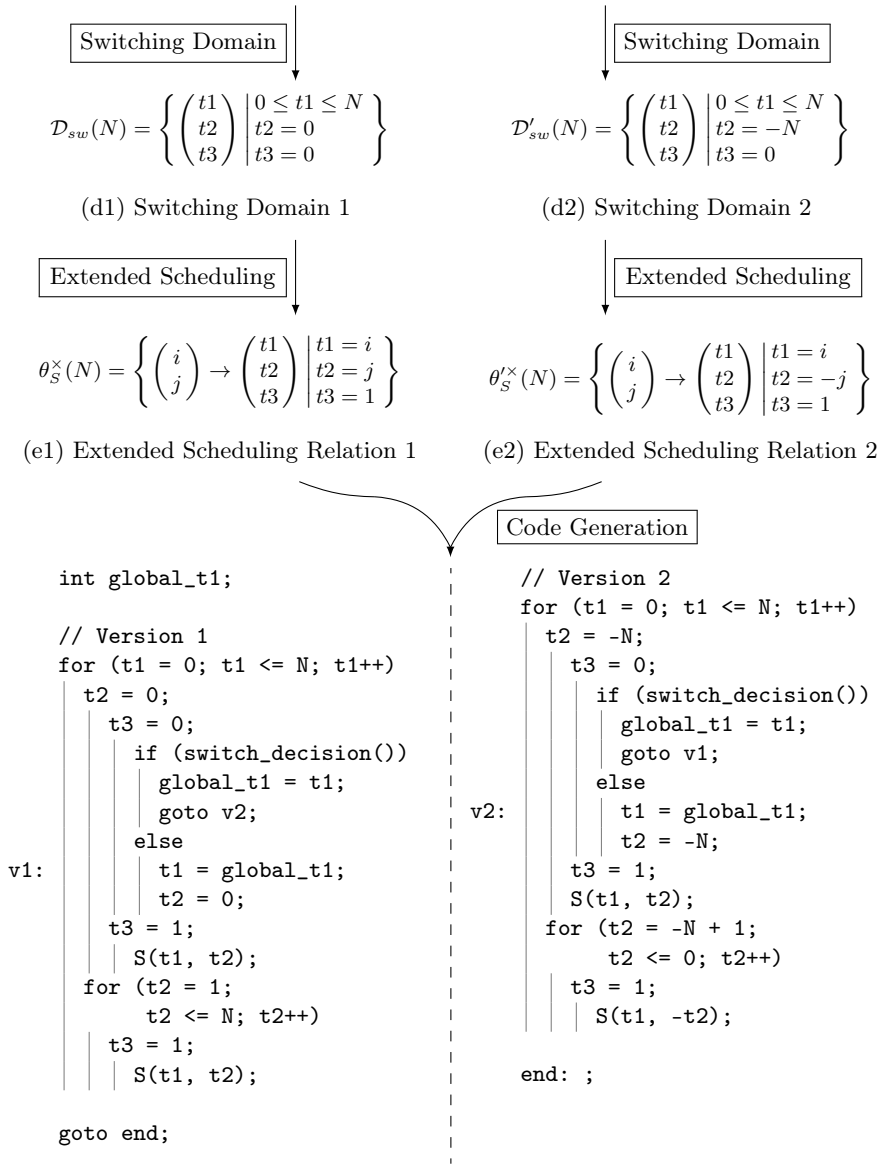
Generating a code that includes multiple versions of the original program with the ability of switching between them is a three step process. First we extend the original scheduling with one innermost output dimension set to 1. It ensures that the switching statement will be executed before any existing instance if they are scheduled at the same logical date, since that output dimension has been set to 0 for the switching domain[5]. Figures 2(e1) and 2(e2) show the extended schedulings of Figures 1(c1) and 1(c2). Next, we generate the code from the original domains and scheduling as in a classical polyhedral framework, with the CLooG tool [1]. The only difference is that we generate a code for each version and that we add the corresponding switching domain to each code generation problem. Each integer point of the switching domain corresponds to an execution of the switching statement. Finally some glue code is added to support switching: additional variables are created to communicate current common output coordinates while switching and labels/gotos are inserted to jump to the end of the code once one version terminates.

The switching statement itself is made of two parts. First, the *switching source* includes calling the runtime to decide about switching or not, communicating of current common output coordinates and actual switching (through goto statements). Second, the *switching sink* includes a label to be used as the target of a switch, receiving the common output coordinates and setting back the remaining output coordinates to the lexicographic minimum. Figure 2(f) shows the final code (spanning two columns) for our running example started in Figure 1. The switching source corresponds to the `if` part of the switching statement while the sink corresponds to the `else` part.

3.3 Runtime

The runtime switching decision system is as simple as possible to minimize the overhead. It is based only on the execution time and has two modes called

[5] If the last output dimension is not a common dimension, another solution without scheduling extension is to subtract 1 to its expression in the switching domain.



(d1) Switching Domain 1

(d2) Switching Domain 2

(e1) Extended Scheduling Relation 1

(e2) Extended Scheduling Relation 2

(f) Final Code Including Two Versions That May Switch To Each Other

Fig. 2: (Our Alternative End of Fig. 1) Generation of a Multi-Version Code

watching and *sampling*. In watching mode, the runtime simply checks that the performance is stable by measuring the time spent between two calls. Since switching statements are inserted at constant strides along output dimensions and SCoP execution time is typically not affected by data values, this measure

is precise enough for our purpose. If it is the first call to the runtime or if the watching mode detected a performance variation, due to, e.g., changes on the execution context or on the workload executed between two calls to the runtime, the sampling mode is enabled. This mode switches quickly between versions to detect the best performing one. Then a switch is performed to that version while the runtime is set back to the watching mode. A very important property of this strategy is that every computation contributes to the final result: no rollback is necessary if a bad optimization decision has been made.

4 Selecting Pertinent Versions

A key aspect of our optimization strategy is the selection and the ordering of the switchable versions to be part of the multi-version code. For this purpose we rely on a dedicated version generation phase and on an extensive empirical study of the version behavior.

To generate versions, we rely on the polyhedral compiler PoCC[6] which uses both the Pluto algorithm [3] and the Letsee iterative optimization engine [11] to compute efficient scheduling. Generating switchable versions is done by enforcing additional constraints discussed in Section 3.1: from a base version, other versions are generated by calling Letsee or Pluto with different strategies and/or tile sizes, such that they share common output dimensions. Different scheduling may often end up to the same executable code (a shifting on an output dimension may be removed by a loop normalization by the compiler). Such versions are discarded.

Once a set of versions has been generated for a given input code, they are evaluated separately by running them on pre-defined contexts. Contexts include various architectures, data sizes and system workloads. One context is a combination of these factors. Only the versions that are the best in at least one context are considered to be selected. Our results show that they are still too many. Some of them are performing the same way in several contexts: those duplicates are detected and discarded (in our study, we accept a performance loss of 10%). Finally to select a pre-defined maximum number of versions (in our study, 8), we associate an "efficiency" coefficient to each version on each context (depending on how far it is from the best version) and we model and solve the choice as a linear optimization problem to maximize the overall efficiency.

The order in which the selected versions are used during sampling by the runtime described in Section 3.3 is critical: small loops are likely to be entirely executed before the sampling is done. For this reason, best performing versions in most contexts including small problem sizes are used for sampling first.

5 Experimental Results

We evaluate the switchable scheduling approach on a selection of realistic execution contexts. Experimental results demonstrate the ability of this technique

[6] http://pocc.sf.net

to generate programs that can adapt themselves to their environment. Overall, its geomean speedup over a fixed optimization of a state-of-the-art automatic optimization and parallelization is **1.49** for our test cases.

Our experimental setup is three-dimensional. First, target architectures includes one ARM and several flavours of Intel x86 architectures: Olimex A20 ARM Cortex-A7 dual-core, Intel Core2 Quad CPU Q9550 2.83GHz, Intel Core2 Quad CPU Q6600 2.40GHz and Intel Core2 Quad CPU Q8200 2.33GHz. This selection notably spans different number of cores and cache sizes. Next, problem size ranges are small and medium as they are defined in the target benchmarks. Lastly, 5 workloads have been investigated: the target process may be running alone, with low (one process) or high (one process per core) computation intensive workload and with low or high memory access intensive workload.

We consider 12 benchmarks, typical compute-intensive kernels extracted from the PolyBench suite[7]. Our selection focuses on kernels including one main loop since it is the main target of our technique. We report below for all benchmarks a short description. Column #versions gives the number of different versions that have been generated using PoCC (duplicates have been removed); #best reports the number of best versions reported in the 40 contexts; and #nodup removes from the previous column the versions that behave in the same way as another one if we accept up to a 10% performance loss. It illustrates that the best version is indeed dependant on the execution context, but also that a limited number of versions is enough most of the time, hence with a reasonable impact on the generated code size.

benchmark	description	#versions	#best	#nodup
2mm	Linear algebra (BLAS3)	40	9	2
adi	Stencil (2D)	67	9	4
choleski	Cholesky Decomposition	16	12	4
durbin	Toeplitz system solver	23	17	4
fdtd-apml	Stencil (3D)	50	10	2
gemm	Matrix-multiply and addition	37	18	4
gramschmidt	Gram-Schmidt decomposition	59	12	2
jacobi-1d	Stencil (1D)	24	11	3
jacobi-2d	Stencil (2D)	19	7	4
lu	Matrix decomposition	19	8	2
mvt	Matrix Vector Product and Transpose	16	8	2
seidel-2d	Stencil (2D)	17	7	4

Figure 3 reports normalized mean performance for all execution contexts for each benchmark, worst corresponds to the worse (context-wise) version, baseline is the mean of all versions, roughly corresponding to the average performance a random strategy is likely to provide, best corresponds to the best (context-wise) solution, pluto is the default static Pluto (version 0.10) solution and switchable is the switchable scheduling solution. Overall, the difference between baseline and best with geomean **4.98** is the maximum speedup of the solution, it corresponds to an iterative compilation strategy, a high potential already demonstrated by previous work [12]. switchable corresponds to our solution with an overall geomean speedup of **4.36** against a random strategy, including a sensible yet acceptable overhead of the switching strategy, and of **1.49** over the default Pluto

[7] http://polybench.sf.net

solution. size growth shows the compiled switchable scheduling kernel size growth with respect to Pluto's solution, a limited increase. Sampling on bad versions may degrade performance significantly (e.g., gemm case). Also in jacobi-1d case, our strategy has lower performance than Pluto. This corresponds to situations where Pluto's solution is good enough while the overhead of switchable scheduling overcomes its benefits. We may complement our technique with a dynamic test as Pradelle et al. suggested [13] to prevent using switchable scheduling in such situation.

benchmark	worst	baseline	best	pluto	switchable	size growth
2mm	0.38	1	3.56	1.48	3.14	1.13
adi	0.13	1	4.46	2.98	4.08	1.07
choleski	0.74	1	1.89	1.35	1.52	1.02
durbin	0.25	1	2.14	1.74	1.90	1.04
fdtd-apml	0.08	1	2.77	2.19	2.61	1.07
gemm	0.31	1	8.42	1.39	5.70	1.04
gramschmidt	0.10	1	18.27	17.34	17.36	0.99
jacobi-1d	0.17	1	19.15	16.30	15.71	1.10
jacobi-2d	0.25	1	8.24	4.08	7.87	1.38
lu	0.24	1	4.42	3.02	4.82	1.04
mvt	0.55	1	2.28	1.54	2.12	1.06
seidel-2d	0.26	1	5.37	2.21	4.97	1.11

Fig. 3: Potential and Operational Performance Results (mean of all contexts, the baseline is the mean performance of all versions in all contexts)

6 Related work

The root of our work belongs to compiler optimization in the polyhedral model [7] and loop versioning [4]. The Pluto algorithm is a state-of-the-art compiler technique relying on the polyhedral model to build complex loop transformations with excellent parallelism-locality trade-offs using a target independent cost model [3]. It has been coupled with iterative frameworks to optimize for specific targets [12]. Those techniques create unspecialized or overspecialized optimization which may not be adequate for various execution contexts.

Static compiler techniques have been used to help runtime systems to optimize dynamically. The ADAPT framework provides runtime generation and specialization of code sections [18]. Because of the runtime overhead it fits well to programs with large execution time while we are using static techniques as much as possible to minimize runtime costs. Qilin provides adaptive mapping for parallel programs [10]. Unlike our method, it is not addressing the dynamic workload dimension of the execution context. Emani et al. proposed an adaptive mapping technique which primarily targets dynamic workload variations [5]. It impacts the OpenMP runtime behaviour whereas we target code restructuring.

Aggressive dynamic optimization techniques include thread-level speculation [14, 15]. They generate an optimistically optimized version and in case of mistake, they rollback to a conservative version. In comparison, we target a different

program class that can be analyzed precisely at compile time, and in case of a bad choice, no rolling back is necessary since every computation is useful by construction. Dynamic optimization involving polyhedral compilation is emerging. `EvolveTile` is a framework to perform a dynamic tile size selection [16]. Our approach also supports such optimization but with more restrictions on tile sizes and shapes because of the switchable scheduling class constraints. However, our technique supports a wider range of optimizations. Pradelle et al. target the same program class as our technique and involve versioning as well [13]. Their approach is to use profiling to build predictive tests according to dynamic factors to choose the best version of a kernel before executing it. Our approach is acting at a finer grain as we focus on switching from kernel versions during computation. `VMAD` is an infrastructure for dynamic profiling with the unique ability to discover static behavior, which is not visible at static compilation time [9]. `VMAD` supports dynamic version selection. Some forms of switchable scheduling are possible within this framework and are under investigation.

7 Conclusion

This paper addresses the problem of taking advantage of the best optimization while computing in an ever more dynamic environment, focusing on static control loop nests. Our proposal differs from just-in-time compilation approaches which have to rely on low-overhead techniques as well as static compilation approaches that generate a code which can be either too generic or too specialized. Instead, we propose a mixed static-dynamic scheme which builds on state-of-the-art static polyhedral compilation techniques with empirical study to select pertinent optimizations and a low-overhead runtime mechanism to switch to the best optimization during computation, depending on the current execution context. Our technique introduces a special class of optimization called switchable scheduling and a code generation method to build a program that takes advantage of multiple such optimizations. Experimental evidence demonstrate both the potential of this approach and its effectiveness at generating codes that perform well on various environments.

Ongoing work includes a code generation technique to allow versions to lie inside their own functions, to benefit from per-version low-level compiler optimization options. More aggressive versioning and switchable-scheduling generation under time constraint are also under investigation.

References

1. C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, Sept. 2004.
2. F. Bodin, T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *W. on Profile and Feedback Directed Compilation*, Paris, Oct. 1998.

3. U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI'08 ACM Conf. on Programming language design and implementation*, Tucson, USA, June 2008.
4. M. Byler, J. R. B. Davies, C. Huson, B. Leasure, and M. Wolfe. Multiple version loops. In *International Conference on Parallel Processing*, Aug. 1987.
5. M. Emani, Z. Wang, and M. O'Boyle. Smart, adaptive mapping of parallelism in the presence of external workload. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10, 2013.
6. P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.
7. P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, Dec. 1992.
8. 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. *Int. J. of Parallel Programming*, 34(3):261–317, June 2006.
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*, volume 7210 of *LNCS*, pages 220–237, Tallinn, Estonia, Mar. 2012.
10. C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO-42. 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 45–55, Dec 2009.
11. L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 90–100, Tucson, Arizona, June 2008. ACM Press.
12. L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *SC'10*, New Orleans, USA, Nov. 2010.
13. B. Pradelle, P. Clauss, and V. Loechner. Adaptive Runtime Selection of Parallel Schedules in the Polytope Model. In *19th High Performance Computing Symposium - HPC 2011*, Boston, United States, Apr. 2011.
14. L. Rauchwerger and D. Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 218–232, New York, NY, USA, 1995. ACM.
15. J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The stampede approach to thread-level speculation. *ACM Trans. Comput. Syst.*, 23(3):253–300, Aug. 2005.
16. S. Tavarageri, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Dynamic selection of tile sizes. In *18th IEEE Int. Conf. on High Performance Computing (HiPC'11)*, Bangalore, India, Dec. 2011.
17. R. Upadrasta and A. Cohen. Sub-polyhedral scheduling using (unit-)two-variable-per-inequality polyhedra. In *ACM Symposium on Principles of Programming Languages*, POPL '13, pages 483–496, Rome, Italy, 2013.
18. M. Voss and R. Eigenmann. ADAPT: Automated de-coupled adaptive program transformation. In *Int. Conf. on Parallel Processing*, pages 163–170, 2000.
19. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2000.
20. M. Wolfe. *High performance compilers for parallel computing.* Addison-Wesley Publishing Company, 1995.