View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by INRIA a CCSD electronic archive server

Runtime Multi-versioning and Specialization inside a Memoized Speculative Loop Optimizer

Raquel Lazcano, Daniel Madroñal, Eduardo Juarez, Philippe Clauss

To cite this version:

Raquel Lazcano, Daniel Madroñal, Eduardo Juarez, Philippe Clauss. Runtime Multi-versioning and Specialization inside a Memoized Speculative Loop Optimizer. CC 2020 - 29th International Conference on Compiler Construction, Feb 2020, San Diego, United States. 10.1145/3377555.3377886 . hal-02457425

HAL Id: hal-02457425

https://hal.inria.fr/hal-02457425

Submitted on 28 Jan 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Runtime Multi-versioning and Specialization inside a Memoized Speculative Loop Optimizer

Raquel Lazcano
Centre of Software Technologies and Multimedia Systems
Universidad Politécnica de Madrid
Madrid, Spain
raquel.lazcano@upm.es

Daniel Madroñal
Centre of Software Technologies and Multimedia Systems
Universidad Politécnica de Madrid
Madrid, Spain
daniel.madronal@upm.es

Eduardo Juarez
Centre of Software Technologies and Multimedia Systems
Universidad Politécnica de Madrid
Madrid, Spain
eduardo.juarez@upm.es

Philippe Clauss
INRIA CAMUS, ICube laboratory
University of Strasbourg
Strasbourg, France
philippe.clauss@inria.fr

Abstract

In this paper, we propose a runtime framework that implements code multi-versioning and specialization to optimize and parallelize loop kernels that are invoked many times with varying parameters. These parameters may influence the code structure, the touched memory locations, the workload, and the runtime performance. They may also impact the validity of the parallelizing and optimizing polyhedral transformations that are applied on-the-fly.

For a target loop kernel and its associated parameters, a different optimizing and parallelizing transformation is evaluated at each invocation, among a finite set of transformations (multi-versioning and specialization). The best performing transformed code version is stored and indexed using its associated parameters. When every optimizing transformation has been evaluated, the best performing code version regarding the current parameters, which has been stored, is relaunched at next invocations (memoization).

CCS Concepts • **Software and its engineering → Just-in-time compilers**; **Dynamic compilers**; **Runtime environments**.

Keywords runtime speculative optimization and parallelization, multi-versioning, memoization, specialization, polyhedral model

ACM Reference Format:
Raquel Lazcano, Daniel Madroñal, Eduardo Juarez, and Philippe Clauss. 2020. Runtime Multi-versioning and Specialization inside a Memoized Speculative Loop Optimizer. In *Proceedings of the 29th International Conference on Compiler Construction (CC '20), February 22–23, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377555.3377886

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

CC '20, February 22–23, 2020, San Diego, CA, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-7120-9/20/02…$15.00
https://doi.org/10.1145/3377555.3377886

1 Introduction

In code optimization, multi-versioning is a well-known approach to generate code that may adapt to a changing execution context: several versions of an original code snippet are generated at compile-time, each one being the result of some different optimizing transformations. When launching the resulting code, some runtime decisions are performed in order to select the convenient version to be run. For example, most mainstream compilers (icc, gcc, clang) generate multi-versioned code when handling automatic vectorization: when some dependences across loops cannot be disambiguated at compile-time, a vectorized code is still generated, but guarded by some dependence tests ensuring its validity. Some more ambitious proposals can be found in the literature, where different advanced loop optimizing and parallelizing transformations are applied to generate multi-versioned loop kernels [11, 25]. Another well-known optimization strategy is to generate specialized code, *i.e.* code where some parameters are instantiated as constants to predicted values, thus yielding more efficient code [9, 13, 23].

However, static multi-versioning, *i.e.* several versions generated at compile-time, may be efficient when all future execution contexts are known: all the generated versions must correspond to a potential execution context. These execution contexts might be related to the hardware execution platform where the code is actually run, or to properties of the target code that impact the validity or the performance of the generated versions. When the execution contexts are mostly unknown at compile-time and cannot be predicted, such an approach becomes impractical. The same holds with specialization and unpredictable values.

An answer to such limitations would be to generate the code versions at runtime, when execution contexts are obviously known. Nevertheless, such an approach includes several steps that might be very time-consuming: profiling, analysis, transformation, and just-in-time compilation.

On the other hand, several dynamic code optimizers and parallelizers have been proposed, which are mostly based on speculation. They may be classified as implementing one of the three following mechanisms: (1) an optimized version of the target code is launched without any previous validity check, that is performed while the optimized code is running [19, 21, 26, 29]; (2) an initial «light» code version is launched, solely devoted to compute the memory addresses that will be referenced, thus ensuring the validity of a following optimized version of the code regarding data dependences (inspector-executor mechanism) [27, 28]; (3) an initial analysis is performed on-the-fly on some initial sample of the whole execution, through instrumentation and profiling, to build a prediction regarding the data dependences and take advantage of advanced optimizing transformations. The prediction is then continuously verified while the optimized code is running [7, 16, 30]. In every approach, when the verification fails, a rollback mechanism is triggered.

However, such speculative optimizers usually apply a single optimizing transformation and do not handle multi-versioning, mostly because of the inherent time-overhead. Thus, the generated code may be inefficient in the context actually met by the code at runtime, or some optimization opportunities may be missed. Another downside occurs when the target code is relaunched many times: such optimizers start again all their analysis and transformation steps at each invocation, yielding a useless time-overhead.

Memoization [2, 15, 20] is also a well-known optimization technique to speed up functions by storing the results of calls and returning the cached result when the same inputs occur again. To our knowledge, memoization has barely been applied to runtime optimizers, for storing the generated optimized codes in order to relaunch them whenever the associated execution contexts occur again.

In this paper, we propose an optimizing framework whose goal is to generate on-the-fly several transformed versions of loop kernels that are intensively used and relaunched many times. Specifically, we focus on stream-based, data-driven applications employed in domains such as video and image processing [31] and neural networks [1]. This kind of applications can be specified with a dataflow Model of Computation (MoC), which is a directed graph of nodes representing computations, the so-called *actors*, and arcs, which represent First-In First-Out (FIFO) queues where data tokens interchanged between actors are stored [18]. Due to their natural expressivity of task parallelism and analyzability, dataflow MoCs are increasingly popular [4]. Dataflow MoCs are especially useful for applications performing the same processing to a stream of data; consequently, they run on a loop, and hence they invoke actor kernels iteratively.

At each invocation of a given kernel, our framework either generates and launches a new code version, which is then stored if better performing when compared to previous versions; or launches the cached version that has been elected as the best performing version for its related parameter values. Indeed, the handled kernels may depend on some parameters that impact the code structure, the touched memory locations or the workload. Such parameters are transmitted to the optimizer in order to characterize the associated kernel. The optimizing transformations that are applied on-the-fly by our framework are polyhedral optimizing and parallelizing loop transformations [5] as tiling, interchange, splitting, fusing and skewing. The multi-versioning runtime optimizer has been implemented as an extension of the Automatic Speculative Polyhedral Loop Optimizer (Apollo) framework [7, 30].

While a traditional approach is to use some profile information for generating multiple versions statically, our multi-versioning framework works entirely at runtime for three main reasons: (1) Parameters values may be unknown at compile-time and only discovered during real executions; (2) With a static approach, too many versions would have to be generated at compile-time (parameters values combined with optimization opportunities), even inefficient ones, and the final executable file, embedding all versions, would be huge; (3) Our framework also takes advantage of one of the main features of Apollo, which is that it handles codes that cannot be handled statically, due to memory references using indirections or pointers, or to unknown loop bounds as in while-loops [7, 30]. This paper brings the following contributions:

- A runtime multi-versioning system for loop kernels;
- Applying polyhedral optimizing and parallelizing transformations on-the-fly;
- Taking advantage of memoization by storing the best performing optimized loop kernels indexed by self-impacting parameters;
- For relaunching afterward the best performing cached version according to the current execution context;
- Using a speculative approach to successfully parallelize loops that may only be parallelizable in some circumstances;
- Implemented as an extension of the speculative polyhedral loop optimizer Apollo.

The paper is organized as follows. In Section 2 some required background is recalled, regarding the polyhedral model and the speculative polyhedral loop optimizer Apollo. Runtime multi-versioning and specialization, which are the main contributions of this paper, are described in Section 3. Experiments showing the effectiveness of our framework are presented in Section 4. Related work is addressed in Section 5. Conclusions are given in Section 6.

2 Background

2.1 The polyhedral model

The polyhedral model [5, 12] has been proven to be a powerful mathematical and geometrical framework for analyzing and optimizing for-loop nests. The requirements are that (i) each loop iterates according to a unique index variable whose bounds are affine expressions of the enclosing loop indices, and (ii) the memory instructions that are handled are limited to access simple scalar variables or multi-dimensional array elements referenced using affine expressions on the enclosing loop indices. Such loop nests are analyzed precisely with respect to data dependences that occur among the statements and across iterations. Thus, advanced optimizing transformations are proven to be semantically correct by preserving the dependences of the original program. The loop nest optimizations (e.g., loop skewing, loop interchange) are linear transformations of the iteration domains that are represented geometrically as polyhedra. Each tuple of loop indices values is associated with an integer point contained in a polyhedron. The order in which the iterations are executed translates to the lexicographic order of the tuples. Thus, transformations represent re-orderings of the iterations and are defined as scheduling matrices, which are equivalent to geometrically transforming a polyhedron into another equivalent form. Representing loop nests as polyhedra enables one to reason about the valid transformations that can be performed.

Although very powerful, the polyhedral model is restricted to a class of compute and memory intensive codes that can be analyzed accurately and transformed only at compile-time. However, most legacy codes are not amenable to this model due to indirect or pointer based accesses to static or dynamic data structures, which prevent a precise dependence analysis to be performed statically. For these reasons, even though the model is powerful, its applicability is limited. While *Pluto*[1] is considered the state of the art polyhedral compiler, it can only handle statically analyzable, affine-characterized, codes.

However, codes that do not exhibit characteristics suiting the polyhedral model at compile time may still be in compliance with the model, although this compliance can only be detected and validated at runtime. Targeting such codes for automatic optimization and parallelization imposes to immerse the polyhedral model in the context of speculative and dynamic parallelization. This is the main goal of Apollo framework [7, 30], whose main features are recalled below.

2.2 Apollo

For loop nests that cannot be analyzed statically, but which exhibit a polyhedral behavior at runtime, Apollo's strategy for making the polyhedral model applicable relies on speculation coupled with runtime verification. It consists of observing initially the original code during a very short sample of the whole run. If a polyhedral behavior is observed on this sample, Apollo speculates that the behavior will remain the same for the rest of the loop nest execution. Thus, Apollo abstracts the loop to a polyhedral representation, reasons about the intra and inter iteration dependences, applies and validates polyhedral optimizations and parallelizing transformations. As long as the prediction remains true, the generated parallel code is semantically correct by definition of the polyhedral model. In order to continuously verify the prediction, and thus verify the correctness of the speculatively optimized program, Apollo implements a decentralized runtime verification system embedded in the parallel code. Each of the parallel threads verifies independently if the next memory location touched has an address equal to the predicted address corresponding to the polyhedral representation. A similar verification is performed for the inner loops' bounds. Such a verification strategy has significantly less overhead when compared to a centralized system traditionally used in Thread-Level Speculation (TLS) systems [19, 21, 26, 29].

Apollo consists of two main parts: a static part implemented as a set of passes of the Clang-LLVM compiler[2], and a dynamic part implemented as a runtime system.

At compile-time, Apollo's static phase: (1) precisely analyzes memory instructions that can be disambiguated at compile-time; (2) generates an instrumented version to track memory accesses that cannot be disambiguated at compile-time; (3) generates elementary pieces of code called code-bones [7], which are units of code in the LLVM Intermediate Representation (LLVM-IR), that will be instantiated, assembled and scheduled at runtime to result in the final optimized code. Additionally, some code bones are devoted to speculation verification.

At runtime, Apollo's dynamic phase: (1) runs the instrumented version on a sample of consecutive outermost loop iterations; (2) builds a linear prediction model for the loop bounds and memory accesses from the instrumentation data; (3) sends the linear model to the Pluto compiler, used as a library, in order to compute dependences and generate an optimizing and parallelizing transformation; (4) selects and instantiates the code bones, and generates using the LLVM JIT compiler an optimized parallel version of the original sequential code, semantically correct with respect to the prediction model, and which is going to run for a slice of the original outermost loop (also called a *chunk*); (5) backs up memory locations which are going to be modified according to the prediction, during the execution of the next chunk; (6) during the execution of the multi-threaded code, each thread verifies independently if the prediction still holds. If not, a rollback is initiated to recover the execution of the last chunk and the system attempts to build a new prediction model for the rest of the future chunks.

A dedicated pragma, added in source code, makes every loop nest in the pragma's scope being handled by Apollo.

[1] http://pluto-compiler.sourceforge.net

[2] http://llvm.org

Note that even if Apollo's main goal is to take advantage of polyhedral optimizations opportunities at runtime, it also implicitly achieves code specialization since many source code parameters are actually instantiated at runtime. This framework, whose source code is freely available[3], implements the basic functionalities required for our multi-versioning and memoization system of loop kernels.

3 Runtime multi-versioning and polyhedral optimization

3.1 General overview

As already mentioned in Section 1, our proposal of a runtime multi-versioning system is especially useful for stream-based, data-driven applications. Such dataflow systems often implement applications performing the same processing to a stream of data. Thus, they run on a loop and invoke code kernels iteratively. A classic scheme where continuous high performance computations of kernels are required is when a device, or system, produces dataflows that must be processed as fast as possible, to yield an apparent «real-time» behavior of the device. The different kinds of output that may be requested are associated to specific kernels, and quantitative and/or qualitative data properties are associated to specific kernel parameters. Since the target execution platform may not be always the same, and the kernel parameters may be unpredictable, a fully dynamic system is required to provide high performance for the computed kernels: the kernels must be automatically parallelized and optimized on-the-fly; in a way that is adapted and specialized for the current execution context; that is characterized by the current kernel parameter values and hardware resource characteristics.

These goals relate to automatic parallelization, multi-versioning and specialization of programs. Since we exclusively focus on loops which can take advantage of the polyhedral model, polyhedral parallelizing and optimizing transformations must be applied on-the-fly. For a given loop kernel, there are many polyhedral transformations which are valid regarding data dependences, and which could be applied. Although the state of the art polyhedral compiler Pluto implements heuristics to decide which transformations may be the most beneficial, its purpose is to generate generally well-performing code, whichever the actual execution context is. Since Pluto is a static compiler, it needs to anticipate runtime behaviors without any runtime information, as it is for any mainstream compiler (excepting when profile-guided compilation is used). Moreover, Pluto offers several invocation flags in order to activate or deactivate some transformations, or to set some transformation parameters. Thus, it is up to the user to select the convenient optimizing strategy.

Our runtime multi-versioning system is depicted in Figure 1. It is organized in two main phases: the training phase

[3]http://apollo.gforge.inria.fr

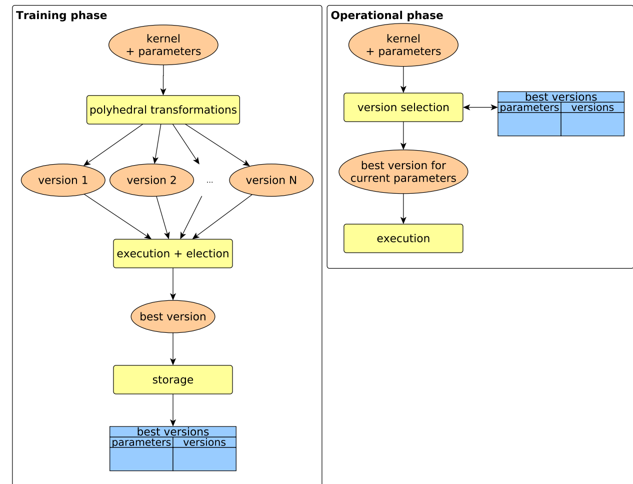

Figure 1. Overview of the multi-versioning training and operational phases

and the operational phase. For each loop kernel and its associated parameters, the training phase is successively relaunched until every polyhedral transformation, among a fixed set of transformations, has been applied to generate a new kernel version. Each polyhedral transformation may be more or less beneficial in some circumstances. For example, an obvious multi-versioning scheme would be to apply loop tiling with several different tile sizes: depending on the cache memory hierarchy or the input data size, one or another tile size may yield the best performing code. Then, among all the generated versions, the best version is elected by comparing the respective execution times. This version is finally stored and indexed relatively to its kernel parameters. Since the kernels may be invoked many times with the same parameters, the stored versions will be relaunched many times and provide directly the best performance, without any further required code analysis or transformation. If a kernel that has already been encountered is launched again, but with parameters that are different than in previous invocations, a new related training phase is launched. Note that our approach includes a two-fold optimization strategy: (1) applying the best polyhedral optimizing transformation among a set of potential transformations and (2) specializing the code regarding the current kernel parameters.

As soon as the training phase associated to a set of kernel parameters has been completed, the operational phase is launched whenever the same parameters are encountered. It simply consists in loading and launching the best version associated to the current and already encountered parameters. Thus this phase implements the memoization of our system.

3.2 Implementation

Our runtime multi-versioning system has been implemented as an extension of the speculative polyhedral loop optimizer

Apollo, since it already implemented several required features (see Subsection 2.2). However, although it successfully applies polyhedral optimizations at runtime, it is blind to the number of times the same kernel is launched. As shown in Figure 2, Apollo's dynamic phase can be roughly summarized in three steps: each time a kernel is invoked, it (1) runs a small subset of consecutive outermost loop iterations to instrument the code and build a prediction model; (2) calls Pluto to generate an optimizing transformation; and (3) generates the optimized version using the LLVM Just-In-Time (JIT) compiler and runs it by chunks, *i.e.*, slices of the outermost loop. But if the same kernel is launched again, the same steps are always repeated, re-instrumenting the code and re-generating the same transformation again. This behavior yields useless time-overheads, since the profiling and code generation steps could be launched once for all invocations of the same kernel and associated parameters.

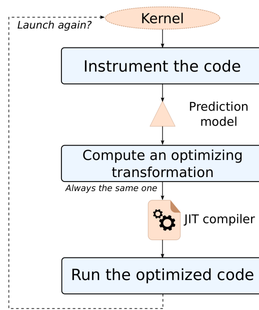

Figure 2. Apollo diagram.

To do so, we have modified and extended Apollo's dynamic phase as depicted in Figure 3. If it is the first time the kernel is ever launched, Apollo behaves «normally», as described before, by generating and applying an optimizing transformation and running the optimized code for a slice of the outermost loop. At this step, Apollo has been modified to apply one optimizing polyhedral transformation among a fixed set of transformations. Each transformation is characterized by a specific set of flags used to invoke the Pluto compiler. Although the number of flag combinations has been set to 10 for our experiments (see Section 4), it can be easily modified and customized by the user. As soon as this process finishes, some important information related to the tested optimizing transformation is stored:

- A unique identifier for the handled kernel and associated parameters;
- The prediction model built after the instrumentation;
- An identifier of the tested transformation;
- The specialized and optimized code generated by the LLVM JIT compiler;
- The execution time of the optimized code.

Next time the same kernel with the same parameters is launched, the prediction model is automatically retrieved, so the instrumentation phase is skipped. Then, Pluto is called again, but this time with a different flag combination, among the fixed set of combinations, to generate and evaluate a different transformation. Once the optimized kernel finishes its execution, the last tested transformation is compared with the stored one regarding its execution time. If the new one is better, it substitutes the stored one. Otherwise, the only information saved is the identifier of the transformation, in order to keep track of the transformations that have already been tested. This process is repeated each time the same kernel is invoked with the same parameters and until all flag combinations have been tested. When this occurs, the best optimized code is automatically launched, avoiding the overhead of calling both Pluto and the LLVM JIT compiler.

Comparing optimized versions: To compare transformations in terms of performance, the comparison criterion must be correctly defined. There are two main issues to address for this purpose. First, in order to counterbalance Pluto's overhead induced from dependence analysis and optimizing transformation selection, Apollo launches in parallel a thread executing simultaneously some iterations of the original loop kernel. When Pluto finishes, this thread is stopped, and the optimized chunk is then launched from the iteration where the thread ended. Consequently, the starting iteration of the optimized chunk depends on the time spent by Pluto in generating the optimizing transformation, and thus cannot be predicted. The second issue is related to the shape of the loop kernel: non-rectangular iteration domains may occur when some loop bounds depend linearly on some surrounding loop iterators. In such cases, the total number of iterations for a fixed-size slice of the outermost loop is not constant, for any outermost loop bound.

Both issues are addressed by computing the exact number of iterations that have been run for any slice of the outermost loop, whichever the loop bounds and the shape of the iteration domain. Instead of incrementing a dedicated counter which would yield some time overhead, this number can be directly computed through the evaluation of the associated Ehrhart polynomial [8]. Ehrhart polynomials are particular polynomials expressing the exact number of integer points inside a parameterized polyhedron. They can be computed using existing software tools like the barvinok library [32]. However, invoking such a library at runtime would be too time-consuming. Hence we implemented in our system the runtime computation of the Ehrhart polynomial associated to each handled loop kernel, whose variables are the outermost loop bounds defining a slice of executed iterations. Then, by dividing the measured execution time by the related and evaluated Ehrhart polynomial, we obtain the average execution time per iteration, which is a figure of merit that can be used to accurately compare two optimized versions.

Ehrhart polynomials are generated as follows. Following the theory presented in [8], for a loop nest of depth d, the

associated Ehrhart polynomial is a polynomial whose degree is at most d, and whose variables are the lower and upper bounds of the outermost loops, which are defining a slice of executed iterations. Hence it is of the general form:

$$ep(lower, upper) = \sum_{i=0}^{d} \sum_{j=0}^{d-i} c_{ij}\, lower^i\, upper^j$$

where coefficients c_{ij}'s are rational numbers that have to be found to generate the Ehrhart polynomial. The number of coefficients that must be determined is:

$$\#unknowns = \frac{d^2}{2} + \frac{3\,d}{2} + 1$$

which is also the number of linear equations that are required to get a solvable system of linear equations. For this purpose, $\#unknowns$ different small and valid combinations of $lower$ and $upper$, for which the total number of iterations is known thanks to Apollo's profiling phase, are used to set a system of linear equations, which is then solved by invoking a solver already implemented in Apollo for other purposes.

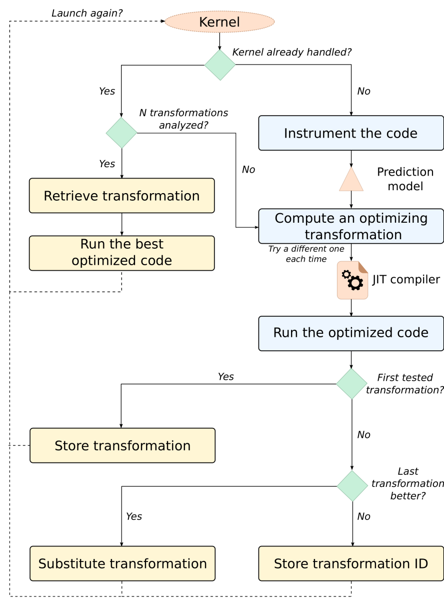

Figure 3. Apollo diagram with multi-versioning and memoization.

The impact of kernel parameters: The handled kernels may depend on some parameters, which in turn can affect the workload, the touched memory locations and/or the code structure. These parameters depend on the application (*e.g.* size of the image, video resolution), and they can be translated to loop bounds, memory accesses, etc. The impact they can have in the loop kernel can be summarized in two main categories: (1) The new values of the parameters can cause a variation in the linear functions predicting the memory accesses, and thus they can invalidate the prediction model built by Apollo for that loop kernel in previous invocations; (2) The new values of the parameters can directly impact the loop bounds, and thus the transformation selected as being the best may not be the best anymore.

Since all of these features directly affect the transformation and, obviously, the specialized code, these changes must be taken into consideration when applying the multi-versioning and memoization mechanism described above. As a result, the framework must be reactive to possible changes in these parameters. In parameterized dynamic dataflow applications, the parameters that may change at runtime are clearly identified, so they can be fed to Apollo's extended runtime system. When there is a change in some dynamic parameter, the process shown in Figure 3 is restarted.

Time-overhead investment: To fully exploit the possibilities offered by combining a dataflow system with a runtime polyhedral parallelizer like Apollo, another functionality has been added. As already stated before, while Pluto is computing the optimizing transformation, Apollo launches a thread in parallel to execute the original loop kernel sequentially. The goal of this mechanism is two-fold: (1) to counterbalance Pluto's overhead; and (2) to counterbalance Apollo's overhead in such a way that, if the sequential thread finishes before Pluto, Apollo's execution is aborted, and the results of the sequential thread are provided to the user. In that way, it is guaranteed that the loop kernel is not slowed down too much by Apollo when the handled loop kernel is too small regarding its workload. In the context of dataflow applications, a new compromise may be reached. As these applications are running on a loop, we consider the possibility of allowing Apollo to spend more time in generating the optimizing transformation, as hypothetically, this time will be counterbalanced by the speedup provided for the next invocations of the same kernel. This functionality has been implemented as a timeout, configured by the user and activated if the sequential thread finishes before Pluto. This extension benefits to small loop kernels, which could not be handled profitably by Apollo before, as it is illustrated by our experiments in Section 4.

Speculative optimization & parallelization: Apollo implements a speculative approach enabling the application of polyhedral optimizations to loop kernels that cannot be handled by static polyhedral compilers, but which exhibit a polyhedral-compliant memory behavior at runtime. Thus some speculative optimizations may fail and imply Apollo's rollback mechanism to be triggered.

An important consequence is that our multi-versioning system handles loop kernels that cannot be handled statically. This feature is particularly interesting for kernels whose memory behavior depends on the shape of the input data: if the encountered shapes can be classified and take part of the kernel parameters, then a dedicated multi-versioning scenario is launched for each kind of encountered data shape,

and a best version especially optimized for the current kind of shape is generated. This aspect is illustrated in Section 4 with `spmatmat`, which is a sparse matrix-vector multiplication, and where versions dedicated to the shape of the input matrix are generated by our system (dense or diagonal matrix).

Our runtime multi-versioning system registers every optimizing transformation whose related optimized code has been successfully run for at least one chunk. During the operational phase, if several transformations have been successfully tested, the best one is launched for each chunk, leaving a chunk running the original serial code between each one. If, after a rollback, the system is unable to find a valid transformation, the original code is launched from the moment the last optimization chunk finishes until the end of the whole kernel invocation.

4 Experiments

The presentation of our experiments is organized as follows: first the hardware platforms and the benchmark programs are presented in Subsection 4.1; then, the parameters used to evaluate the multi-versioning mechanism are given in Subsection 4.2; finally, speed-ups provided by our framework are presented and discussed in Subsection 4.3.

4.1 Description of the experiments

The benchmarks were run on two hardware platforms:

- **Namaka**: A general-purpose multicore platform including an AMD Ryzen™Threadripper™1950X processor with 16 cores running at 2.4 GHz, with 32 GB of RAM, running Linux 4.15.0-65 and with cache sizes of 32K L1d, 64K L1i, 512K L2, 8192K L3. The experiments have been conducted on the 16 cores of the processor, with one thread per core.
- **Magerit**: A cluster of 68 Lenovo ThinkSystem SD530 nodes, each one including two Intel Xeon Gold 6230 processors with 20 cores running at 2.1 GHz, with 192 GB of RAM, running Linux 3.10.0-957 and with cache sizes of 32K L1d, 32K L1i, 1024K L2, 28160K L3. The experiments have been conducted on the 20 cores of one Intel Xeon Gold 6230 processor, with one thread per core, using 4 GB of RAM per core.

The set of benchmark programs has been built from two different benchmarks suites: the Apollo benchmarks[4] and a selection of loop kernels from the Polybench benchmark suite[5]. In total, 20 different benchmark programs have been used to evaluate our framework: 11 from Polybench and 9 from Apollo, including `spmatmat` with two different configurations related to the shape of the input matrices.

Note that programs from the Apollo benchmarks cannot be handled by a static compiler, since they contain either memory references using pointers or indirections, or while-loops. Although programs from the Polybench benchmark suite may be handled statically, such programs must still be handled at runtime when input parameters are unknown at compile-time, and when too many versions have to be evaluated, which is actually our context of use.

The Apollo benchmarks can be configured with three different problem sizes: `small`, `medium` and `large`; the Polybench benchmarks also provide those three sizes, but also size `extralarge`. We have observed that size `small` induces a workload that is too weak for parallelization, thus it is not included in our experiments, while sizes `medium`, `large` and `extralarge` are. Thus in the following, we rename the Apollo benchmarks sizes in the same way. They are used in our experiments as being dynamic parameters associated to the loop kernels. For program `spmatmat` from the Apollo benchmarks, one additional qualitative parameter indicates whether the input matrix is dense or diagonal.

Every benchmark code has been compiled using Apollo compiler, *i.e.* `apolloc` or `apolloc++`, which are based on clang 6.0.1, and optimization flags `-O3 -march=native`. Runs were performed with OpenMP parallelization and after having set the environment variable OMP_PROC_BIND=true to avoid thread migration and bind the threads to processor cores, and the environment variable OMP_NUM_THREADS to 16 for platform Namaka and 20 for platform Magerit. Moreover, programs were launched by using the linux tool `taskset` to explicitly select specific processor cores.

4.2 Setup of the multi-versioning system

Our multi-versioning system has been configured to test 10 different polyhedral optimizing transformations, for each benchmark program and associated parameters. Additionally, the original code is also considered as a candidate, since there are cases where the original serial code is the best option, particularly when the workload is very low. As described in Section 3, the set of tested transformation candidates can be easily modified by the user. Although Pluto offers many possible flag combinations, we reduced the number of tested combinations to the 10 that may potentially provide the best performance. The involved Pluto flags are: `--tile`, to activate loop tiling with tile sizes that may be customized; `--intratile`, to promote intra-tile execution order for data locality; `--parallel`, to enable OpenMP loop parallelization; and `--identity`, to prevent loop interchange and skewing. The selected combinations of these flags are shown in Table 1.

The timeout related to the generation of the optimizing transformation has been set to two times the execution time of the original serial loop kernel: if the original code, which is launched simultaneously to the generation of the optimizing transformation, completes first, then the same time, as the original code execution time, is granted in addition to finish the generation of optimized code.

[4] http://apollo.gforge.inria.fr/download
[5] https://sourceforge.net/projects/polybench

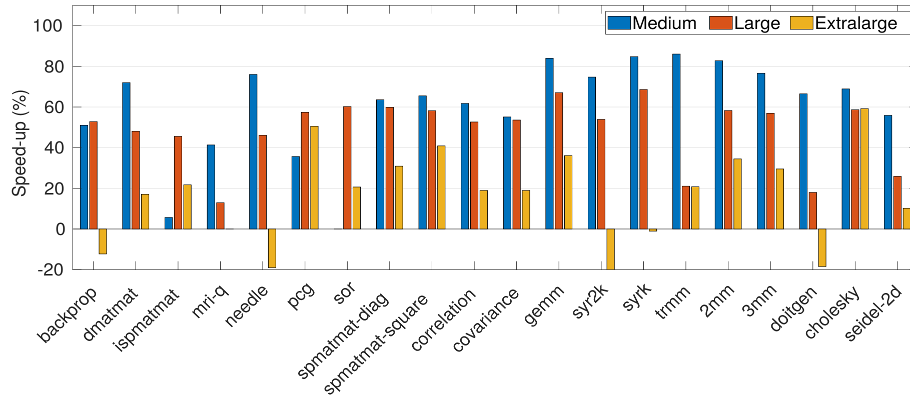

Figure 4. Speed-up with 100 invocations against Apollo without multi-versioning (Namaka: 16 threads/cores)

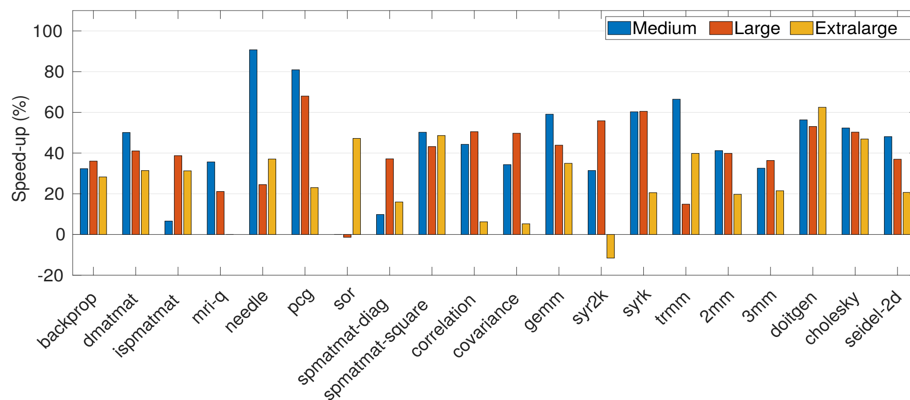

Figure 5. Speed-up with 100 invocations against Apollo without multi-versioning (Magerit: 20 threads/cores)

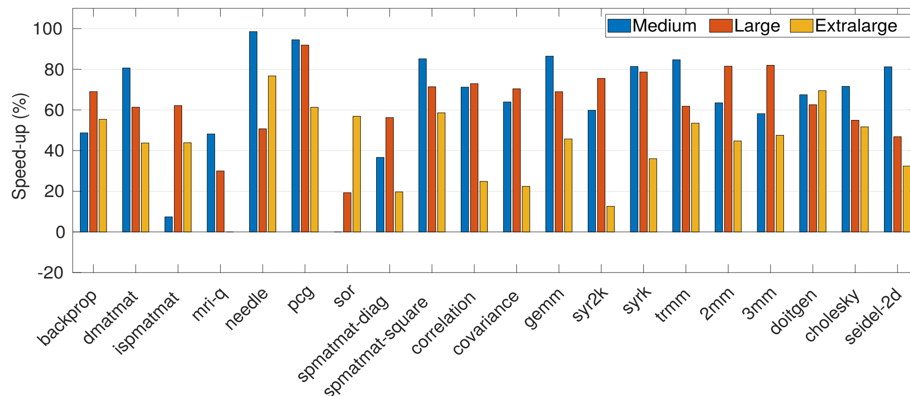

Figure 6. Maximum speed-up with unlimited invocations against Apollo without multi-versioning (Magerit: 20 threads/cores)

4.3 Performance results

Our framework has been run by invoking 100 times each loop kernel, for each problem size among `medium`, `large` and `extralarge`. Additionally, for kernel `spmatmat`, which implements a sparse matrix-vector multiplication, two different kinds of inputs were provided: a dense square matrix and a diagonal matrix. This scenario has been repeated 5 times, and the average execution times have been used to compute the speed-ups against the execution of Apollo without our multi-versioning system (one-shot Apollo). The obtained speed-ups are shown in Figures 4 and 5 for hardware platforms Namaka and Magerit respectively. Note that no `extralarge` input is provided in the Apollo benchmarks for `mri-q`.

Table 1. Pluto flag combinations used for multi-versioning

ID	Flag combination
0	no transformation (original code)
1	intratile + parallel
2	tile $32 \times 32 \times 32$ + parallel
3	tile $32 \times 32 \times 32$ + intratile + parallel
4	tile $32 \times 32 \times 32$ + intratile + identity + parallel
5	tile $64 \times 64 \times 64$ + parallel
6	tile $128 \times 128 \times 128$ + intratile + parallel
7	tile $32 \times 32 \times 64$ + parallel
8	tile $64 \times 64 \times 32$ + parallel
9	tile $16 \times 64 \times 32$ + parallel
10	tile $16 \times 64 \times 32$ + identity + parallel

Significant speed-ups are obtained for most of the loop kernels and problem sizes on both platforms, since our multi-versioning system allows to select a better performing optimized kernel version than the unique version generated by one-shot Apollo. However, a few slow-downs can be observed for kernels `backprop`, `needle`, `syr2k` and `doitgen` and size `extralarge` on platform Namaka (also on platform Magerit for `syr2k`). In these cases, 100 invocations is not enough to counterbalance the time overhead induced by the training phase of multi-versioning.

This issue is highlighted when extrapolating the measures to an unlimited number of invocations, such that the training phase execution time becomes negligible. The associated theoretical speed-ups are shown in Figure 6 for platform Magerit. One can observe that kernels `backprop`, `needle`, `syr2k` and `doitgen` may after all provide speed-ups above a certain number of invocations.

The exact numbers of invocations required to counterbalance the time overhead of the training phase are given in Table 2. Symbol ∞ denotes cases where the training phase can never be counterbalanced, since no optimized code versions could be generated with the timeout value used in these experiments. It is the case with kernel `sor` on Magerit with size `medium`. However, note that the related slow-downs, showed in Figures 5 and 6, can be considered as negligible.

The flag combinations characterizing the best elected versions are given in Table 3, by using the ID's of Table 1. One can observe that some kernels, (*e.g.* `correlation`), are composed of several loop nests which are handled separately by Apollo. Thus several different combinations of optimizations are tested and selected by the multi-versioning system for one unique kernel. Note also that in several cases, the multi-versioning system wisely selects the original serial kernel version, either because the transformed versions are slower, or the timeout did not enable the generation of any optimized version, especially for small workloads. More importantly, one can observe that different optimizations are selected for a given kernel when changing the problem size or the platform. In particular, the best performing tile sizes are obviously different on Namaka and Magerit, due to different cache sizes

Table 2. Minimum number of invocations required to counterbalance the time-overhead of runtime multi-versioning (M=Medium, L=Large, E=Extralarge)

Program	Namaka			Magerit		
	M	L	E	M	L	E
backprop	32	31	326	34	48	49
dmatmat	20	28	37	38	34	29
ispmatmat	14	18	32	1	38	29
mri-q	27	52	NA	26	30	NA
needle	24	38	207	1	52	52
pcg	48	29	24	15	27	63
sor	41	24	39	∞	108	17
spmatmat-diag	25	19	25	74	34	19
spmatmat-square	24	27	20	42	40	18
correlation	36	36	62	38	31	75
covariance	42	33	63	47	30	77
gemm	14	22	25	32	37	24
syr2k	21	34	168	48	26	193
syrk	14	23	104	26	24	43
trmm	12	63	39	22	76	26
2mm	15	33	48	36	52	56
3mm	19	37	56	45	56	55
doitgen	23	60	264	17	16	1
cholesky	20	1	1	27	1	1
seidel-2d	36	30	54	41	22	37

and types. This fully justifies the automatic adaptation of optimized code through runtime multi-versioning.

5 Related Work

To our knowledge, there is no proposal of another runtime framework performing multi-versioning on loop kernels optimized and parallelized on-the-fly using polyhedral techniques, and furthermore taking advantage of memoization.

In contrast, there are several proposals related to one of our framework's feature, however mostly accomplished at compile-time, *i.e.*, statically. This is mostly due to the time-overhead of dynamic code generation, which is significantly lowered in our approach thanks to speculation and the collaborative static-dynamic mechanism of Apollo, based on the static generation of code-bones [7], instantiated and scheduled at runtime to reflect a polyhedral optimization.

As mentioned in the introduction, multi-versioning is implemented in mainstream compilers for two main goals: (1) generating vectorized loops guarded by some dependence tests; (2) enabling programmers to specify multiple versions of a function, where each function is specialized regarding some hardware feature.

The ATLAS system [33] achieves empirical tuning of BLAS kernels where numerous variants are repeatedly executed on the target architecture, in order to find the best one.

Diniz and Rinard [10] propose dynamic feedback, a technique for selecting code variants based on measured execution times. A program has alternating sampling and production phases. In the sampling phase, code variants, generated

Table 3. Optimizations selected by the multi-versioning system to generate the best performing code (M=Medium, L=Large, E=Extralarge)

Program	Loop nest	Namaka			Magerit		
		M	**L**	**E**	**M**	**L**	**E**
backprop	1	0	2	3	0	2	3
dmatmat	1	3	1	6	1	1	9
ispmatmat	1	0	0	6	0	1	9
mri-q	1	2	6	NA	6	6	NA
needle	1	0	0	5	0	0	6
	2	0	0	6	0	0	5
pcg	1	0	4	4	4	4	4
sor	1	0	0	3	0	0	9
spmatmat-diag	1	0	3	8	0	3	3
spmatmat-square	1	3	1	5	3	1	1
correlation	1	0	0	6	0	9	9
	2	0	0	0	0	0	0
	3	0	1	6	0	0	0
	4	0	0	0	0	0	0
covariance	1	0	1	0	0	9	9
	2	0	0	6	0	0	0
	3	0	0	0	0	0	0
gemm	1	0	3	5	0	1	9
syr2k	1	0	10	10	0	10	10
syrk	1	0	4	4	0	4	10
trmm	1	0	6	8	0	3	8
2mm	1	1	2	9	0	1	2
	2	0	1	9	0	1	1
3mm	1	0	3	7	0	1	1
	2	0	2	7	0	1	1
	3	0	7	9	0	1	1
doitgen	1	0	0	3	0	7	3
cholesky	1	0	0	0	0	0	0
seidel-2d	1	1	1	1	1	1	7

at compile-time using different optimization strategies, are executed and timed. This phase continues for a user-defined interval. After the interval expires, the code variant that exhibited the best execution time is used.

PetaBricks [3] provides a language and a compiler where having multiple implementations of multiple algorithms to solve a problem is the natural way of programming. The associated runtime system uses a choice dependency graph to select one or another algorithm and implementation at different steps of the whole computation, thus resulting to an optimized hybrid algorithm. Such an approach is suitable for programs where it is obviously possible to switch from one algorithm to another while still making progress in the whole computation.

One of the earliest methods proposed for generating multiple version loops was proposed by Byler *et al.* in [6]. In this technique, several variants of a loop are generated at compile-time and the best version is selected based on runtime information. Pradelle *et al.* [25] propose a multi-versioning framework where several optimized versions of a loop kernel are generated at compile-time using Pluto. Then, these versions are profiled offline in order to set up the relevant runtime selection criteria. Doerfert *et al.* [11] propose a framework to handle assumptions based on Presburger arithmetic, in order to derive a minimal set of preconditions to validate polyhedral optimizations at runtime. However in these proposals, the optimized loop kernels are generated at compile-time, which limits the scope to the related execution contexts (parameter values, potential optimizations, etc.).

Similarly, iterative compilation [17] searches through the transformation space to find the best optimizations. In [24], Pouchet *et al.* propose advanced techniques using an heuristic and a genetic algorithm to traverse huge polyhedral optimization spaces.

Consel and Noël [9] address code specialization through the generation of code templates with «holes» at compile-time, that may then be selected at runtime by filling holes with runtime values and relocating jump targets. Grant *et al.* propose DyC [14] which is a staged compiler that generates a runtime compiler from an annotated program. Then, the runtime compiler generates the executable using runtime values. Note that DyC also implements memoization since the generated executables are stored for potential reuse. Oh *et al.* [22] automatically specialize loops based on patterns. An offline profiling phase collects value profiles to identify static instructions that always produce the same value.

6 Conclusion

In this paper, it has been shown that runtime multi-versioning and specialization using polyhedral loop optimizations can be effective, thanks to a speculative approach using prediction based on instrumentation by sampling. Significant speed-ups may be obtained when invoking many times some loop kernels, characterized by dynamic parameters, as it is implemented by dataflow systems.

Among the perspectives of further extensions, one interesting option would be to store the best elected versions and their associated execution contexts into a file on disk, which could then be reloaded each time the invoking dataflow system is relaunched. This added feature would save the time overhead of the training phase and thus provide systematically the theoretical speed-ups for an unlimited number of invocations (Figure 6).

Acknowledgments

The authors acknowledge the computer resources and technical assistance provided by the Centro de Supercomputación y Visualización de Madrid (CeSViMa). This work has been supported by Universidad Politécnica de Madrid under the Programa Propio RR01/2015, by CERBERO european project (Grant No.: 732105) and by the Spanish Government through PLATINO project (TEC2017-86722-C4-4-R).

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.

[2] Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2003. Selective Memoization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. ACM, New York, NY, USA, 14–25. https://doi.org/10.1145/604131.604133

[3] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: A Language and Compiler for Algorithmic Choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/1542476.1542481

[4] Florian Arrestier, Karol Desnos, Eduardo Juarez, and Daniel Menard. 2019. Numerical Representation of Directed Acyclic Graphs for Efficient Dataflow Embedded Resource Allocation. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 101 (Oct. 2019), 22 pages. https://doi.org/10.1145/3358225

[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[6] Mark Byler, Michael Wolfe, James R. B. Davies, Christopher Huson, and Bruce Leasure. 1987. Multiple Version Loops. In *ICPP*.

[7] Juan Manuel Martinez Caamaño, Manuel Selva, Philippe Clauss, Artiom Baloian, and Willy Wolff. 2017. Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones. *Concurrency and Computation: Practice and Experience* 29, 15 (2017), e4192. https://doi.org/10.1002/cpe.4192 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4192 e4192 cpe.4192.

[8] Philippe Clauss. 1996. Counting Solutions to Linear and Nonlinear Constraints Through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs. In *Proceedings of the 10th International Conference on Supercomputing (ICS '96)*. New York, NY, USA, 278–285.

[9] Charles Consel and François Noël. 1996. A General Approach for Run-time Specialization and Its Application to C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. ACM, New York, NY, USA, 145–156. https://doi.org/10.1145/237721.237767

[10] Pedro C. Diniz and Martin C. Rinard. 1997. Dynamic Feedback: An Effective Technique for Adaptive Computing. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI '97)*. ACM, New York, NY, USA, 71–84. https://doi.org/10.1145/258915.258923

[11] Johannes Doerfert, Tobias Grosser, and Sebastian Hack. 2017. Optimistic Loop Optimization. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE Press, Piscataway, NJ, USA, 292–304. http://dl.acm.org/citation.cfm?id=3049832.3049864

[12] Paul Feautrier and Christian Lengauer. 2011. Polyhedron Model. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Springer US, 1581–1592. https://doi.org/10.1007/978-0-387-09766-4_502

[13] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. 1999. An Evaluation of Staged Run-time Optimizations in DyC. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99)*. ACM, New York, NY, USA, 293–304. https://doi.org/10.1145/301618.301683

[14] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. 1999. An Evaluation of Staged Run-time Optimizations in DyC. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99)*. ACM, New York, NY, USA, 293–304. https://doi.org/10.1145/301618.301683

[15] Marty Hall and J Paul McNamee. 1997. Improving software performance with automatic memoization. *Johns Hopkins APL Technical Digest* 18, 2 (1997), 255.

[16] Alexandra Jimborean, Philippe Clauss, Jean-François Dollinger, Vincent Loechner, and Juan Manuel Martinez Caamaño. 2014. Dynamic and Speculative Polyhedral Parallelization Using Compiler-Generated Skeletons. *International Journal of Parallel Programming* 42, 4 (01 Aug 2014), 529–545. https://doi.org/10.1007/s10766-013-0259-4

[17] Toru Kisuki, Peter M. W. Knijnenburg, Mike F. P. O'Boyle, François Bodin, and Harry A. G. Wijshoff. 1999. A feasibility study in iterative compilation. In *High Performance Computing*, Constantine Polychronopoulos, Kazuki Joe Akira Fukuda, and Shinji Tomita (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 121–132.

[18] Edward A Lee and David G Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245.

[19] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. 2006. POSH: a TLS compiler that exploits program structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 158–167.

[20] Donald Michie. 1968. "Memo" Functions and Machine Learning. *Nature* 218, 306 (April 1968). https://doi.org/10.1038/218019a0

[21] Cosmin E. Oancea, Alan Mycroft, and Tim Harris. 2009. A Lightweight In-place Implementation for Software Thread-level Speculation. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '09)*. ACM, New York, NY, USA, 223–232. https://doi.org/10.1145/1583991.1584050

[22] Taewook Oh, Hanjun Kim, Nick P. Johnson, Jae W. Lee, and David I. August. 2013. Practical Automatic Loop Specialization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 419–430. https://doi.org/10.1145/2451116.2451161

[23] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. 1997. Tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI '97)*. ACM, New York, NY, USA, 109–121. https://doi.org/10.1145/258915.258926

[24] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. 2008. Iterative Optimization in the Polyhedral Model: Part Ii, Multidimensional Time. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 90–100. https://doi.org/10.1145/1375581.1375594

[25] Benoît Pradelle, Philippe Clauss, and Vincent Loechner. 2011. Adaptive runtime selection of parallel schedules in the polytope model. In *2011 Spring Simulation Multi-conference, SpringSim '11, Boston, MA, USA, April 03-07, 2011. Volume 6: Proceedings of the 19th High Performance Computing Symposia (HPC)*. 81–88. http://dl.acm.org/citation.cfm?id=2048588

[26] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. 2010. Speculative Parallelization Using Software Multi-threaded Transactions. *SIGARCH Comput. Archit. News* 38, 1 (March 2010), 65–76. https://doi.org/10.1145/1735970.1736030

[27] Lawrence Rauchwerger and David A Padua. 1999. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems* 10, 2 (1999), 160–180.

[28] J. H. Saltz, R. Mirchandaney, and K. Crowley. 1991. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.* 40, 5 (May 1991), 603–612. https://doi.org/10.1109/12.88484

[29] J Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C Mowry. 2005. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems (TOCS)* 23, 3 (2005), 253–300.

[30] Aravind Sukumaran-Rajam and Philippe Clauss. 2015. The Polyhedral Model of Nonlinear Loops. *ACM Trans. Archit. Code Optim.* 12, 4, Article 48 (Dec. 2015), 27 pages. https://doi.org/10.1145/2838734

[31] Giuseppe Tagliavini, Germain Haugou, Andrea Marongiu, and Luca Benini. 2015. Adrenaline: An openvx environment to optimize embedded vision applications on many-core accelerators. In *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. IEEE, 289–296.

[32] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. 2007. Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions. *Algorithmica* 48, 1 (2007), 37–66.

[33] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC '98)*. IEEE Computer Society, Washington, DC, USA, 1–27. http://dl.acm.org/citation.cfm?id=509058.509096