Adaptive Runtime Selection of Parallel Schedules in the Polytope Model

Benoit Pradelle, Philippe Clauss, Vincent Loechner

Adaptive Runtime Selection of Parallel Schedules in the Polytope Model

Benoît Pradelle, Philippe Clauss and Vincent Loechner
CAMUS, INRIA Nancy-Grand Est and LSIIT, Université de Strasbourg, CNRS, France.
{benoit.pradelle, philippe.clauss, vincent.loechner}@inria.fr

Abstract

There is often no unique version of a program that provides the best performance in all circumstances. Compilers should rely on an adaptive runtime decision to choose which optimizing and parallelizing transformations will lead to the best performance in any execution context. We present a new adaptive framework solving two drawbacks of existing methods: it is effective since the very first execution, and it handles slight variations of input data shape and size.

In our proposal, different code versions of parallel loop nests are statically generated by the compiler. At install time, each version is profiled in different execution contexts. At runtime, the execution time of each code version is predicted using the profiling results, the current input data shape and the number of available processor cores. The predicted best version is then run.

Our framework handles several versions of possibly tiled parallel loops, using the polytope model for both the profiling and the dynamic selection phases. We show on several benchmark programs that our runtime system selects one of the most efficient version with a very low runtime overhead. This quick and efficient selection leads to speedups compared to the usage of a unique version in every execution context.

Keywords: Adaptive code selection, runtime performance evaluation, parallel loop nest, polytope model

1. INTRODUCTION

Program performance is very difficult to ensure in general with the ever extending and changing variety of processor architectures and execution contexts that an application can meet. This is particularly true with the proliferation of multicore architectures, where parallel execution of applications introduces an even higher level of performance uncertainty. Hence static compilers are unable to decide which optimizing code transformations have to be applied to take advantage of the underlying platform resources and reach good performance [13]. Even worse, there is no unique optimized version providing the best performance in all situations [1, 11]. Hence the best performance relies on multiple versions of codes having the same functional feature.

For instance, we observed on `gemm`, a matrix multiplication code using simple control and data structures, that depending on the input data, distinct parallel versions provide the best performance, while running on a single machine. Further, we observed that the version providing the best performance is also varying depending on the computer, even with the same input data. As a consequence, performing adaptive version selection is particularly important in a library including such a function, and even more important if the library is distributed as a binary on several platforms.

From several optimizing and parallelizing code transformation alternatives, a usual approach is to do iterative compilation: compiling and profiling the execution of different versions in order to select the most efficient one [13, 14]. However, results obtained from such a strategy are closely related to the execution context when profiling on the compilation platform. The execution context includes the target architecture, the input data size and shape, and the processor load.

Another solution is dynamic work distribution, as the dynamic schedule implemented in OpenMP [12] or work stealing methods such as Cilk [5]. Those methods are adapted to task parallelism, and when applied to data parallelism they often lead to a high and unpredictable overhead: the task granularity is much smaller in this latter case, increasing the relative cost of work distribution control.

Other studies [1, 11, 16, 19] have proposed adaptive runtime selection between several algorithms, code extracts, or versions of a function. Those tools can handle any kind of codes however their granularity of adaptiveness is limited to many complete executions of the versions. We will discuss them further in the related work section.

The contributions of this work are: (1) a static/dynamic collaborative framework providing a fast selection process between nested loops characterized by distinct parallel versions, where the selection process is launched each time a loop nest is executed, thus providing very high adaptiveness; (2) the construction of a ranking table for each considered version and target multicore processor, obtained through an install time profiling run; (3) an implementation of the framework and an evaluation of its effectiveness with several benchmark programs and execution contexts.

We use the polytope model [3] to generate multiple versions of a loop nest. This model provides many loop transformation techniques, implemented, for instance, in the automatic loop parallelization tools PLuTo [6] and LeTSeE/PoCC [13, 14]. The main differences between the versions are the parallel schedules, the tile sizes and their dimension, and the loop unroll factors.

Typically, loops that fit the polytope model can be found in intensive scientific applications: exit conditions and array access functions are affine functions of iterators and parameters, and there is no test on data values. The polytope model provides an accurate frame allowing us to envisage automatic generation of the parallel code versions and to extract high level information from the source code.

Some previous related proposals have chosen to characterize more general applications using empirical measurements such as the observed behavior of previous executions [11] or machine learning techniques [17]. They have to assume that the empirical measurements used are representative of all the execution contexts in the general case. In the polytope model, the behavior of the loop nests is fully deterministic and statically analyzable. We use this property to select very efficiently a code version without making such strong assumptions on the similarity between the current execution context and the previous ones.

Using a collection of benchmarks (from SPECOMP [2], BLAS [4], and PLuTo [6]), we show that our runtime system almost always selects the most efficient out of three to seven distinct versions, and selects another good performing version in the other cases. The time overhead is in order of a millisecond on our test platforms, thus being quite negligible in most cases and allowing speedups compared to the best static version.

The rest of the paper is organized as follows. In the next section, some background on the polytope model is recalled. Section 3 describes our framework in detail. Experiments are presented in Section 4, and related work in Section 5. Finally, Section 6 gives our conclusions and some perspectives.

2. POLYTOPE MODEL

We consider loop-intensive codes that can be handled by the polytope model: all loop bounds are affine expressions of the outer loop iterators and parameters. Thus, we exclude for example `while` loops, or `for` loops with additional control in the body of the loops such as `break` or `goto` statements.

2.1 State of the art

The polytope model allows us to represent a nest of d loops as a d-dimensional polytope called an *iteration domain*. Each loop bound is an affine inequality on the variables, that defines a geometric half space. All of these half spaces, corresponding to all loop bounds, intersect as a convex polytope.

Each statement of a loop-intensive code is associated to an iteration domain, computed from the loops that surround this statement. If the loop bounds depend linearly on integer parameters, then the loop nest is parametric and the corresponding polytope is also parametric (which is often the case in intensive computation loop codes, taking as input an $M \times N$ matrix for example).

Using the polytope model, one can transform the iteration domains in order to generate a new scanning loop nest, changing the schedule of the statements [9, 10] and leading in the end to a new, possibly parallel, loop nest. Those transformations are expressed by *transformation matrices*.

In order to be valid, that is to say to respect the program semantics, the transformations must respect the dependencies of the program. The dependence analysis computes which iteration depends on which other. A set of affine constraints automatically built from the access functions of the data allows us to compute the dependence vectors [8] and to ensure the correctness of the transformations.

Tiling can be done by increasing the dimension of the iteration domains by the number of outer iterators to scan tiles, and are easily coupled to the original iterators with additional constraints [6]. Unrolling can also be done, after polyhedral transformation.

We use CLooG [3] to generate the resulting code, from the statements iteration domains and the transformation matrices.

2.2 Generating different versions

Many performance issues result from the choice of the transformation matrices: degree of parallelism, load balancing, communications volume, cache locality, and vectorization capability depend on this choice. Our framework is based on the capability of generating many different valid versions in order to choose the most performing one at runtime.

For our benchmarks, we generated different schedules by hand. In order to automatically generate many versions, current automatic parallelizers such as PLuTo [6] and LeTSeE/PoCC [13, 14] could be slightly modified: instead of searching for the unique best optimization using heuristics, apply all the possibly efficient optimizations. It could also be interesting to generate code versions considered as less efficient due to some hardware hypothesis. For example a code version expected to be inefficient due to its cache misuse could be the best one on a hardware accelerator without any cache (FPGA, GPU, ...).

2.3 Ehrhart polynomials

The runtime selector presented in Section 3 uses polytope integer points counting, in order to compute the number of iterations that a statement will perform: it is the number of integer points contained in the iteration domain.

This counting is achieved using Ehrhart polynomials [7]: an Ehrhart polynomial associated to a parametric polytope is a piecewise pseudo-polynomial in the parameters of the iteration domains, that expresses the number of integer points contained in the polytope. It is generated at compile time using the Barvinok library [18], and evaluated at runtime, when the parameters values are known.

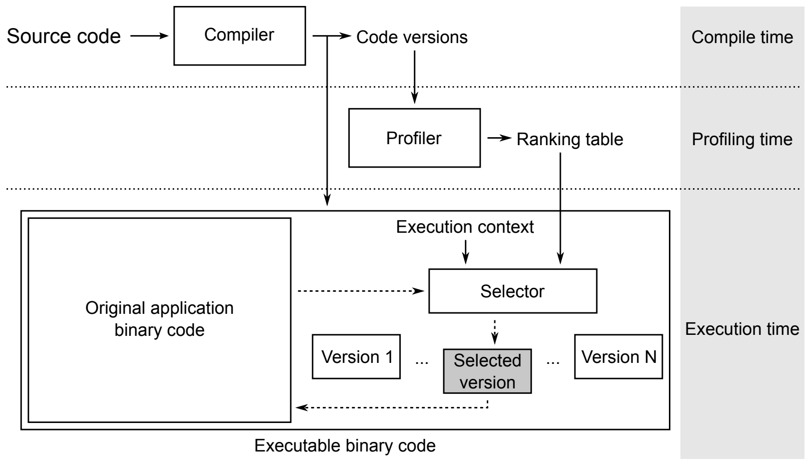

Figure 1. Framework overview

3. SELECTION FRAMEWORK

While other proposals select the best available code version after many executions, our framework aims to select it before any loop nest execution. To achieve this goal, we predict each code version execution time and run the version predicted to be the best one. This prediction is not necessarily exact, but it is accurate enough to rank the different code versions.

To predict those execution times, our framework first profiles each code version in various execution contexts and with different number of active threads. The profiled execution times are gathered in a *parametric ranking table*, see for instance Table 1. A runtime prediction step, executed before each execution of a loop nest, adapts those results according to the current context in order to predict an execution time for each code version. It is embedded in the application binary and is run through a simple function call replacing all the target loop nest executions. As in every version selection framework, all the versions are embedded in the binary file, leading to increase the application size. This increase is linear in the number of versions but is often negligible relatively to the total application size: in our experiments we measured a few kilo-bytes for a loop nest version.

Figure 1 shows a global overview of the framework. The different phases are represented vertically from the generation of the different code versions at compile time to the execution of the loop nest. Automatic generation of the versions is described in Subsection 2.2. The profiling and selection phases are detailed below.

3.1 Initial profiling

3.1.1 Accuracy

The profiling phase, run once on the target computer, typically at install time, is in charge of measuring the execution time of each code version in some representative contexts. However many factors can impact the execution time. Those factors have to be taken into account in order to measure execution times in contexts that can safely be considered as representative. We distinguish two main kinds of factors: *static* and *dynamic* factors.

The impact of static factors is constant during the execution and are naturally taken into account when measuring the execution time of any run. They are thus considered by our profiling step.

The second category consists in *dynamic* factors whose impact is not constant. We define the *dynamic extern* factors as factors depending on events provoked outside of the considered execution, usually other applications. Such factors are considered as out of the scope of the profiling step.

Dynamic intern factors depend on the considered code execution. Among all the potential factors, we identified two main dynamic intern factors relevant to current architectures that might have a major impact on execution time.

First, the overall memory performance of the application is a dynamic intern factor. To handle it, the domain size is doubled between two consecutive measurements until having less than one percent of difference in the consecutively measured execution times. The profiling result is then the last measurement. It allows us to measure the performance of each version when the data cannot fit in the first cache levels without needing any architecture-dependent information.

We observed that this first factor has a small impact compared to the load balance. When parallelizing a loop nest, a parallel loop iteration is a task mapped to a thread. Thus there is a strong link between the parallel loop bounds and the maximal number of simultaneously active threads. The loop bounds depend on which code transformation has been applied and often on the input data size. Thus, this factor is very variable and can lead to large performance loss.

To characterize the load balance impact, each version execution time is measured with a number of parallel iterations increasing from 1 to the number of processor cores during consecutive measurements. The profiling result is therefore parametrized by the maximum number of simultaneously active threads.

3.1.2 Equity

The other key point is to ensure that the comparison based on the predicted execution times is as fair as possible. It means that, despite their differences, all the code versions have to be profiled in a controlled and comparable execution context.

As explained before, the domain size is increased during consecutive profiling runs. Then different versions can reach different domain sizes depending on their characteristics. Thus the ranking table actually contains execution times per loop nest iteration instead of absolute execution times.

To be efficient, codes parallelized with polyhedral transformations are usually tiled [15]. Significant performance variations can be observed between codes where tiles are fully executed and where only incomplete tile executions occur. To handle this variation, we have chosen to build the profiling domains exclusively with full tiles.

3.1.3 Building a profiling domain

The profiling code of each version is built such that those accuracy and equity requirements are met. The profiling domain has to be made of full tiles and its shape has to be precisely controlled. Iteration domains in the polytope model are defined by a set of linear constraints. To achieve our objectives, the framework removes all the iteration domain boundary constraints from the set of constraints. Then it bounds the parallel loop with a new parameter `par_sz` and the other dimensions at the first tiling level with a new parameter `others_sz`. The `par_sz` parameter will be instantiated with the successive number of threads to repeat the measurements for all the possible number of active threads. The `others_sz` parameter is increased during the profiling to control the domain size until reaching stable measurements. Only the first tiling level is constrained by this new parameter in order to execute full tiles when there are more than one tiling levels. If the domain is not tiled, all the dimensions are constrained.

Figures 2 and 3 illustrate the profiling code generation. In Figure 2, we present a code summing the columns of array `A` in array `S`. The code is fully tiled and parallelized. Figure 3 shows the corresponding profiling code. The domain boundaries are eliminated and the domain is now controlled by the two new parameters `par_sz` and `others_sz`. While `others_sz` is doubled until reaching a stable measurement, `par_sz` is incremented from 1 to the number of available processor cores. The result is made of the last execution times per iteration for each considered parallel loop trip count.

```
for (iT=0; iT<=M/64; iT++)
  forall (jT=0; jT<=N/64; jT++)
    for (i=64*iT; i<=min(64*iT+63,M); i++)
      for (j=64*jT; j<=min(64*jT+63,N); j++)
        S[j]+=A[i][j];
```

Figure 2. Sample code

```
do {
  old_result = copy_array(result);
  others_sz = others_sz * 2;
  for (par_sz=1; par_sz<=NB_CORES; par_sz++){
    start = time();
    for (iT=0; iT<others_sz; iT++)
      forall (jT=0; jT<par_sz; jT++)
        for (i=64*iT; i<=64*iT+63; i++)
          for (j=64*jT; j<=64*jT+63; j++)
            S[j]+=A[i][j];
    result[par_sz] = (time() - start) /
        (others_sz * par_sz * 64 * 64);
  }
}while (difference(result, old_result) > 0.01
    && enough_memory(other_sz));
```

Figure 3. Sample profiling code

For each value of both parameters, the execution time is measured using the regular operating system timing function. The expression used to compute the number of iterations `(others_sz * par_sz * 64 * 64)` is the Ehrhart polynomial representing the iteration domain size.

The full ranking table, built during the profiling step, is two dimensional. One dimension is made of the different versions while the second dimension represents the number of available processors. For a given version, and a given number of processors *P* available to the application, the ranking table gives the average execution time per iteration of this version when using *P* processor cores. A sample ranking table is presented in Table 1.

3.2 Runtime selection

The runtime selector is in charge of instantiating the parametric ranking tables resulting from the profiling step, to predict each version execution time.

# of cores	version 1	version 2	version 3
1	30 *ms*	28 *ms*	32 *ms*
2	10 *ms*	14 *ms*	15 *ms*
3	7 *ms*	9 *ms*	8 *ms*
4	5 *ms*	8 *ms*	6 *ms*

Table 1. Sample parametric ranking table built on a 4-cores processor for 3 versions. The table content is made of execution times per iteration.

3.2.1 Iteration count measurement

The profiling phase considers most of the code versions performance factors, however the load balance has to be evaluated at runtime: the data size and the number of available processor cores impact the maximum number of active threads. To evaluate this load balance, the runtime selector executes a very simplified copy of each loop nest version called *prediction nest*. The prediction nest counts how many iterations each thread executes in this code version. The number of iterations executed by each possible quantity of threads is then deduced and used to select the entries in the ranking tables.

To build the prediction nest, we consider a code version loop nest. All the loops and statements enclosed in the parallel loop define the computation achieved in a parallel iteration. The workload of this computation is evaluated as the Ehrhart polynomial counting the number of iterations of the loops enclosed in this parallel loop. Thus, at compile time, we syntactically replace the content of the parallel loop by a computation of this Ehrhart polynomial. The value defined by the polynomial is added to a counter specific to the thread which executes the current parallel iteration. At the end of the prediction nest execution, each counter will then contain the number of iterations that one thread has executed.

Figure 4 presents a sample code and its associated prediction nest. Observe that the innermost loop and the statement are replaced by a counting code. This counting code increments the thread-specific counter with the trip count of the removed loop.

It is assumed here that the mapping of the parallel iterations to the threads is the same in the prediction nest and in the corresponding code version, thus excluding dynamic mapping from the scope of our framework. To evaluate the impact of this drawback, we measured the performance of the most efficient version of each benchmark presented in Section 4 using the OpenMP dynamic mapping. We observed that, in average, this dynamic mapping leads to a slowdown compared to static mapping. This slowdown reaches more than 20 % on some architectures. The same experiment has been performed using Cilk [5]: the parallel loop is transformed into the `cilk_for` control structure. We observed slowdowns of 45 % in average, reaching more than 100 % in some cases. This illustrates that those dynamic scheduling or work-stealing systems can suffer from high overheads and are not suited to deal with regular loop nests.

To deduce the number of parallel iterations executed by each thread quantity, the counters array `cnt` is sorted in descending order. For each position $1 \leq i < nb\_threads$, we can state that `cnt[i-1] - cnt[i]` iterations have been executed in parallel by exactly `i` threads. For example, consider a 4-cores computer where the following numbers of iterations have been measured by the prediction nest:

```
for (i=0; i<M; i++)
  forall (j=0; j<N; j++)
    for (k=i; k<j; k++)
      A[j][k] = B[i] * 2;

for (i=0; i<M; i++)
  forall (j=0; j<N; j++)
    cnt[thread_id] += (j>i)?j-i:0;
```

Figure 4. Sample loop nest (top) and its corresponding prediction nest (bottom)

`cnt` = {`1000, 800, 200, 200`}. We can deduce that one thread has executed (1000 - 800) iterations while the others were idle, two threads have executed (800 - 200) iterations in parallel and four threads have simultaneously executed 200 iterations. There are never exactly three active threads in this case.

3.2.2 Predicting the execution time

Once the iteration counts are known, the execution time can easily be deduced using the parametric ranking table. For example, consider the previous example where 200 iterations are executed by one thread, 600 by two threads and 200 by four threads. For this example, we consider the profiled execution times of the first version in Table 1. The execution time is then computed as $200 \times 4 \times 5\,ms$ for the part of the iteration domain executed by four threads, $600 \times 2 \times 10\,ms$ for the iterations executed by two threads plus $200 \times 1 \times 30\,ms$ for the sequential iterations. The sum predicts an execution time of 22 seconds.

The prediction time computation is executed with every version of the loop nest in the current execution context, before running the best one.

4. EXPERIMENTS

We run our experiments on three computers with different multicore processors: an AMD Opteron 2431 processor with six cores , an AMD Phenom II 965 processor with four cores and an Intel Core i7 920 processor with four cores and hyperthreading activated. The experiments are run on Linux 2.6.35 systems.

The benchmark codes are 12 common polyhedral loop nests. The code `2mm` is made of two matrix multiply ($D = A \times B \times C$), `adi` is the ADI kernel provided for example with PLuTo, `covariance` is a covariance matrix computation, `gemm` and `gemver` are taken from BLAS [4], `jacobi-1D` and `jacobi-2D` are the 1D and 2D versions of the Jacobi kernel, `lu` is a LU decomposition kernel, `matmul` is a simple matrix multiply, `matmul-init` is a matrix multiply combined with the initialization of the result matrix, `mgrid` is a kernel extracted from the `mgrid` code in SPECOMP [2] and `seidel` is a Gauss-Seidel kernel as provided with PLuTo.

Those kernels are typically put in libraries and called many times by applications cumulating the benefits of our system.

For each code, many versions are generated. One of those version is automatically generated by the PLuTo parallelizing compiler. Other versions have been designed by an expert, changing the number of tiling levels and the tile sizes, and making polyhedral loop transformations. As explained in Subsection 2.2, those versions could have been generated by automatic tools. The details about those versions are available on request. From each version, the profiling and prediction codes are generated by a set of fully automatic scripts in our implementation. The result of the prediction codes determines the framework choice in each execution context.

The runtime system is evaluated on each of the three computers, for each code with five different problem sizes leading to execution times near to a couple of minutes. Each measurement is repeated with a number of threads varying from one to the number of logical cores simulating different resource availability. The presented statistics enclose more than one thousand runs, each one being the median value out of five executions.

4.1 Performance variation

During our experiments, we have observed many performance variations. First, there are performance variations for a given code version among different computers for a fixed problem size and number of cores. For example with `adi`, when considering a problem size of 2448 and 5 available processor cores, version 2 is the best one with the Opteron processor but the worst one with the Core i7 processor.

We have also observed performance variations when the number of available cores on a given computer varies for a fixed problem size. For example, with the Core i7 processor for a problem size of 9996, the first version of `gemver` is the best one when only one processor core is available but the worst one when all the cores can be used.

Finally, when the problem size changes there are performance variations for a fixed computer and a fixed number of available cores. For example, with all the processor cores of the Phenom processor, the fourth version of `mgrid` is inefficient for a problem size of 400 but is the best one for a problem size of 496.

The best version is therefore not necessarily the same in all the execution contexts and our runtime system succeeds in selecting the best one in those presented cases.

We have observed that the transformations making a version perform better than another are the tile size, the number of tiling levels and the parallel schedules. All those transformations interact and have a complex impact on performance, changing with the execution context. This emphases the need of a dynamic system, such as our framework, able to consider all of those transformations together.

4.2 Evaluation results

We present in Table 2 a quantitative study of our runtime system efficiency. All the presented speedups are averages of all the execution contexts where the number of active processors and the data size vary. They take the dynamic system overhead into account. We measured this overhead, induced when predicting the execution times, to be less than a tenth of milliseconds per code version.

Processor	Program	Framework speedup	Max static speedup
Phenom	`2mm`	100.0 %	100.0 %
	`adi`	101.5 %	97.5 %
	`covariance`	100.0 %	99.9 %
	`gemm`	102.8 %	96.9 %
	`gemver`	100.0 %	99.8 %
	`jacobi-1d`	99.7 %	100.0 %
	`jacobi-2d`	99.5 %	100.0 %
	`lu`	100.0 %	100.0 %
	`matmul`	100.0 %	100.0 %
	`matmul-init`	100.0 %	100.0 %
	`mgrid`	98.0 %	98.1 %
	`seidel`	100.1 %	99.0 %
Opteron	`2mm`	100.0 %	100.0 %
	`adi`	102.2 %	97.3 %
	`covariance`	100.0 %	99.8 %
	`gemm`	101.4 %	96.7 %
	`gemver`	99.9 %	99.8 %
	`jacobi-1d`	99.6 %	100.0 %
	`jacobi-2d`	100.0 %	100.0 %
	`lu`	100.0 %	100.0 %
	`matmul`	100.0 %	100.0 %
	`matmul-init`	100.0 %	100.0 %
	`mgrid`	97.8 %	98.5 %
	`seidel`	100.6 %	98.3 %
Core i7	`2mm`	100.0 %	100.0 %
	`adi`	102.4 %	97.5 %
	`covariance`	96.9 %	99.7 %
	`gemm`	100.0 %	93.5 %
	`gemver`	89.5 %	91.6 %
	`jacobi-1d`	99.6 %	99.9 %
	`jacobi-2d`	90.6 %	99.6 %
	`lu`	93.1 %	98.3 %
	`matmul`	100.0 %	98.5 %
	`matmul-init`	100.0 %	100.0 %
	`mgrid`	100.5 %	97.0 %
	`seidel`	99.9 %	99.6 %

Table 2. Speedup of our framework compared to the best static version and speedup of the best static version compared to the best version in each context.

We define the *best static version* as the version that performs best in average in all the considered execution contexts on an architecture. Such version could be the one statically chosen by a perfect offline system. If a best static version would be the fastest one in every execution context, no speedup can be expected from any runtime system.

First we show the speedup of our framework compared to the best static version. We can see that our system sometimes leads to a slowdown compared to this version. Those slowdowns are mostly due to incorrect choices in some execution contexts. Such incorrect choices are made when the approximations of the profiling step lead to mispredict the execution time of some versions in some of the execution contexts. Two typical examples are `gemver` and `jacobi-2d` on the Core i7 platform. We can see however that our framework usually selects another well performing version in those cases as the slowdown remains very limited. It is important to note that this slowdown is related to a perfect static system which is extremely complex to build as it would require to statically select the best parallel version for each architecture. Such a system does not exist currently.

Our framework reaches some speedups in some cases (*e.g.* `adi` or `gemm`), meaning that the execution time is, in average, shorter than the execution time of the version that would have been selected by a perfect static system. This speedup is impossible to reach with only one version, illustrating the importance of using a dynamic selection system.

In the second column, we show the speedup of the best static version compared to the best version in each execution context. No speedup higher than 100 % can be expected here: it is impossible to have any version faster than the fastest version in a specific context. A speedup of 100 % here means that a unique version is the best one in all the tested execution contexts. Those speedups illustrate that even a perfect static system cannot reach the maximum performance in all the cases.

Considering those statistics, we can say that our framework is able to select a very well performing version in all the presented cases. In some cases, it is even able to outperform any hypothetical perfect offline system.

5. RELATED WORK

Iterative compilation and dynamic work distribution (work-stealing or dynamic scheduling) have been discussed in the introduction and in Subsection 3.2.1. This section focuses on adaptive runtime selection methods.

In the ATLAS project [20], empirical timings are used in order to choose the best method for a given architecture "in a matter of hours". Further it does not handle parallel programs.

With the ADAPT system [19], a specific language allows the user to describe optimizations and heuristics for applying these optimizations dynamically. However, the resulting optimizer is run on a free processor or on a remote machine across the network.

PetaBricks [1] 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. Further, such a changing behavior must not induce overheads due to a compulsory cache flush for instance.

Mars and Hundt's static/dynamic SBO framework [11] consists in generating at compile-time several versions of a function that are related to different dynamic scenarios. These scenarios are identified at runtime thanks to the microprocessor event registers. Execution is dynamically rerouted to the code relevant to the current identified scenario. However, execution is not rerouted during a function call, but for the next calls. Further, it seems difficult to use this approach with parallel program since it is actually challenging with multicore processors to deduce accurate global multithreaded program behaviors from registers disseminated on the cores.

The STAPL adaptive selection framework [16] runs a profiling execution at install time to extract architectural dependent information. This information is used at runtime combined with previous runs and training runs performance measurements through machine learning to select the best version. Their system requires many training runs before being able to take good decisions.

More recently Tian *et al.* [17] propose an input-centric program behavior analysis for general programs, a statistical approach where program inputs have to be characterized differently depending on the target application – input size, data distribution, etc. –, and program behavior is represented by relations between some programs parameters, as for instance loop trip-counts. This modeling is used to achieve some dynamic version selection. Such relevant statistical relations seem difficult to be determined for any kind of programs. The approach necessarily suffers from approximations, hardware characteristics are not handled and overall, this work does not consider parallel and multicore programs.

6. CONCLUSION AND PERSPECTIVES

In this paper, we present a new framework for adaptive code selection of parallel loop nests at runtime. It handles varying input data and varying execution contexts, such as the number of cores or the cache architecture. We showed on a set of benchmarks that our framework is efficient: depending on the input data shape, on the target architecture and cores availability, the version performing best is not the same, and our method mostly selects the best performing one with a low runtime overhead. Further, it provides speedups that could not have been reached with one statically selected version when considering several calls with different input data shapes.

We are able to consider very different versions where a wide range of optimizations can be applied, from common compiling optimizations to complex polyhedral transformations. Further, our initial profiling phase is also very fast, lasting less than a couple of minutes in our experiments.

As a perspective, we plan to improve our parametric profiling phase by enriching the structure of the simplified nest in order to better consider more complex iteration domains. Another interesting extension would be to consider loop nests that are not strictly polyhedral. Finally, a complete dynamic system, *i.e.* without any initial profiling phase, could be built either by alleviating some mechanisms or by using actual runs to build the parametric ranking table.

REFERENCES

[1] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: a language and compiler for algorithmic choice. In *PLDI '09*, pages 38–49. ACM, 2009.

[2] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPEComp: A new benchmark suite for measuring parallel computer performance. In *WOMPAT '01*. Springer-Verlag, 2001.

[3] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13*, pages 7–16, 2004.

[4] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28:135–151, 2001.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5), 1999.

[6] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI '08*, pages 101–113. ACM, 2008. `pluto-compiler.sourceforge.net/`.

[7] P. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing*, 19(2):179–194, 1998. Kluwer Academic Pub.

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

[9] P. Feautrier. Some efficient solutions to the affine scheduling problem, part 1 : one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, 1992.

[10] P. Feautrier. Some efficient solutions to the affine scheduling problem, part 2 : multidimensional time. *Int. J. of Parallel Programming*, 21(6), 1992.

[11] J. Mars and R. Hundt. Scenario based optimization: A framework for statically enabling online optimizations. In *CGO '09*, pages 169–179, 2009.

[12] The OpenMP API specification for parallel programming. http://www.openmp.org.

[13] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *PLDI'08*, pages 90–100. ACM Press, June 2008.

[14] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *CGO'07*, pages 144–156. IEEE Computer Society press, March 2007.

[15] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *J. of Parallel and Distributed Computing*, 16(2):108–120, 1992.

[16] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *PPoPP '05*, pages 277–288. ACM, 2005.

[17] K. Tian, Y. Jiang, E. Z. Zhang, and X. Shen. An input-centric paradigm for program dynamic optimizations. In *OOPSLA '10*, pages 125–139. ACM, 2010.

[18] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, June 2007.

[19] M. J. Voss and R. Eigemann. High-level adaptive program optimization with ADAPT. In *PPoPP '01*, pages 93–102. ACM, 2001.

[20] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.