

IST ACOTES Project
Deliverable D3.4
**Task-Level Optimization
Prototype**

IST-034869

Public

History

Project Number	IST-034869
Project Title	Advanced Compiler Technologies for Embedded Streaming
Deliverable Type	Prototype

Deliverable Number	D3.4
Title of Deliverable	Task-Level Optimization Prototype
Nature of Deliverable	Public
Internal Document Number/version	acotes-d3.4-final
Contractual Delivery Date	31 May 2008
Actual Delivery Date	25 August 2008
WP(s)	3
Author(s)/Affiliation	Albert Cohen, Louis-Noel Pouchet, Konrad Trifunovic (INRIA)

Abstract

This deliverable describes the task-level optimization libraries designed for the ACOTES tool chain. Based on the polyhedral model of compilation, these libraries support automatic parallelization and adaptation of thread-level parallelism to a target. The libraries can also be applied to loop nest optimization of single-threaded code. Early experimental validation is described in the document, as well as associated publications.

Keyword list

Automatic parallelization, loop nest optimization, static control loops, polyhedral model, task fusion, task coarsening, loop fusion, loop tiling.

Contents

1	Introduction	3
2	Iterative Task-Level Optimization and Parallelization	3
2.1	Problem Statement	4
2.2	Proposed Approach	4
2.2.1	Building a Search Space of Program Partitions	5
2.2.2	Traversing the Search Space	5
2.2.3	Instantiating a Full Program Version	6
2.3	Implementation	6
2.4	Experimental Results	6
3	Semi-Automatic Task-Level Optimization	7
3.1	Task Fusion	7
3.2	Task Blocking	8
3.3	Internal Compiler Representation	8
4	Pipeline Parallelism and Distributed Memory Platforms	8

1 Introduction

The goal of the ACOTES project is to substantially improve the programmer’s productivity in developing applications that process massive streams of data on programmable, parallel embedded architectures. The target application areas of ACOTES are video processing and advanced radio communications for consumer systems. Work Package (WP) 2 of ACOTES defined a Streaming Programming Model (SPM) and an Abstract Streaming Machine (ASM). The ASM itself consists of two parts: the first part deals with the inter-core/tile/cluster characteristics of the target platform, including coarse-grained Data Level Parallelism (DLP) features, and serves as an input to the high-level analyzes of WP2 and optimizations of WP3; the second part deals with the capabilities of the target platform at the level of each individual core/tile/cluster (including fine-gained DLP), and serves as an input primarily to the low-level optimizations and primarily auto-vectorization (WP4).

This deliverable presents the first prototype of a task-level optimization framework, aiming at offering performance portability for applications following the SPM on platforms described by the ASM. Portability of performance requires taking code generation, scheduling, mapping and tuning decisions at both runtime and compilation time. Runtime decisions are handled by the ASM runtime itself, and studied in WP2. Interactions between this optimization framework and finer grain optimizations have been discussed in Deliverable D4.2. Therefore, the present deliverable focuses on task-level optimizations performed at compilation time.

The structure of the document is as follows: Section 2 describes the automatic task-level optimization framework; Section 3 discusses interactions of this framework with the SPM and ASM for semi-automatic task-level optimization; Section 4 outlines future extensions to model pipeline parallelism in distributed memory platforms.

2 Iterative Task-Level Optimization and Parallelization

The ACOTES approach for streaming applications consists of compiler-assisted mapping of streaming tasks to multi-processor systems in order to achieve cost-effective systems, both in terms of energy and in terms of design costs. The analysis and transformation techniques will automate large parts of the partitioning and mapping process, based on the properties of the application domain, on the quantitative information about the target systems, and on programmer directives, so called pragmas. Streaming is based on components that consume and produce (vector) data and are controlled by external components. The pragmas define the

input, outputs and control variables. These pragmas should hint where the borders of the components are. The actual components are then to be built based on information from the ASM. The decision should take into account the needed resources (memory, processing, real-time requirements, etc.). Real-time requirements such as frame-size and delay are provided by the pragmas as well. The ASM provides the processing parallelism and processor vectorization capabilities available, and in addition communication overhead (processing and delay) between the processors. Task-level optimizations base their parallelism-enhancing decisions on the pragmas and the resources needed by each constructed component mapped to each processor. Of course, delay and overhead should be minimized.

2.1 Problem Statement

Our approach consists in coupling iterative optimization in the polyhedral model with a scalable and effective loop tiling algorithm. This approach is currently restricted to a so-called *static control part* (SCoP) of a program, a single-entry-single-exit region of the control-flow where loops bounds and conditional predicates are affine functions enclosing loop counters and invariant parameters. Such a restriction is not unrealistic for the application domain studied in the ACOTES project, as demonstrated by the vast literature on loop nest optimization. In addition, most non-affine control flow arises from inner conditional branches implementing algorithmic modes or options that can easily be approximated as static control on media applications; see Palkovic’s PhD thesis [4] for an in-depth study of such extensions. Another restriction of our approach is the ability to generate bulk-synchronous data-parallelism only, with an OpenMP syntax and semantics. This restriction is annoying for streaming applications that most naturally map to a computation pipeline that cannot be represented with data-parallel forall loops and synchronization barriers, and although it does not appear to cause serious performance overheads on shared-memory platforms, it is likely to trigger load imbalances, synchronization overhead and memory usage overflow on distributed-memory architectures. We will sketch a complementary approach to alleviate both restrictions in Section 3.

We designed and implemented the LEgal Transformation SpacE Explorer (LetSee) implements tunable algorithms for the construction of a search space of legal and distinct affine schedules, representing arbitrarily complex sequences of loop transformations. It comes with a series of iterative and feedback-directed traversal techniques, leveraging our studies of the performance distribution [6, 5]. Typically these techniques rely on subspace partitioning and Generic algorithms to increase scalability on the larger search spaces (up to 10^{50} possible program versions in our test suite).

Independently, but relying on the same principles and tools, powerful model-based techniques to address program optimization and parallelization through tiling have been designed. Bondhugula introduced P_Lu_To, a practical locality and communication optimizer in the polyhedral model [2]. The key principle is to try to fuse a maximum number of statements inside common loops, which have the property of being permutable: this allows to tile the created loop nest. This technique is called the *tiling hyperplanes* approach, as it tries to transform the code to find common tiling hyperplanes (that is, common permutable loops) for the statements. A very interesting property is that these hyperplanes expose coarse grain parallelism if possible, generating parallel tiled output code. Further processing such as loop unroll-and-jam is eventually performed on the generated code to increase its performance.

The current cost model in P_Lu_To is geared towards maximal fusion, to maximize the number of statements contained in a tile. This results in general in an improved locality. However, a careful distribution/fusion tradeoff may often be more profitable, partly because of locality properties (cache or local memory constraints), or because of downstream transformations triggered in the compiler (e.g., vectorization, as studied in Deliverable D4.2). Deciding where to distribute or fuse can quickly become a combinatorial process. Moreover, predicting the profitability of this decision is very hard, severely challenging the design of a portable decision heuristic. We propose to use iterative optimization to solve this problem.

2.2 Proposed Approach

To select which statements should (try to) be inside a common tile, we build in LetSee the set of legal partitionings of the program. A partition is defined by assigning to each statement a partition number.

PLuTo is then applied on each of these partitions, looking for maximal fusion of the set of statements belonging to the same partition.

For instance, for a program with 4 statements, the point

$$v = \{c_1^1 = 0, c_2^1 = 0, c_3^1 = 1, c_4^1 = 1\}$$

represents two distinct partitions $l_1 = \{S1, S2\}$ and $l_2 = \{S3, S4\}$. In other words, $S1, S2$ are fused in a loop l_1 and $S3, S4$ are fused in a loop l_2 , executed after l_1 has completed.

2.2.1 Building a Search Space of Program Partitions

We build and traverse in LetSee the set of all possible points corresponding to legal fusion and distribution of all statements. This set is represented as a convex set by affine conditions on the values of c_k , in such a way that each point in this set represents a legal partitionning of the input program, according to the polyhedral dependence graph. The construction algorithm can be sketched as follows:

1. initialize the set as the full-space;
2. for all pairs of statements, do;
 - check if the two statements can be fused, and add the corresponding constraint(s) to the space if needed;
 - check if the two statements can be distributed, and add the corresponding constraint(s) to the space if needed;
3. perform a projection of the space, to propagate transitively connected constraints.

2.2.2 Traversing the Search Space

We iteratively try different splitting strategies of the original program, at the first loop level. To the contrary of the case of legal multidimensional schedules, the set of legal partitionning can be very small. For instance, for the example shown in Figure 1, there exists only 8 different legal partitionnings at the first loop level.¹ This authorizes an exhaustive search, but it is not always possible on larger loop nests. More scalable techniques based on sampling in the search space are currently under active investigation.

```

for (i=0; i<M; i++)
  for (j=0; j<M; j++)
    A[i][j] = A[i][j] + u1[i] * v1[j] + u2[i] * v2[j];
for (i=0; i<M; i++)
  for (j=0; j<M; j++)
    x[i] = x[i] + beta * A[j][i] * y[j];
for (i=0; i<M; i++)
  x[i] = x[i] + z[i];
for (i=0; i<M; i++)
  for (j=0; j<M; j++)
    w[i] = w[i] + alpha * A[i][j] * x[j];

```

Figure 1: Code for GEMVER

¹Allowing loop interchange, e.g., to enable the fusion of the first two loops; the legality is precomputed by LetSee, but the required interchange is computed by PLuTo after the partitioning is set by LetSee. Notice that fusing the first two loops may be legal, but the resulting loops are not interchangeable and do not allow further tiling of the fused body.

2.2.3 Instantiating a Full Program Version

A program version is represented as a *scheduling matrix* of that program. It is then provided to a code generator which will apply that scheduling matrix to the input program to generate a compilable code.

A partitioning of the input program can be seen as a partial scheduling matrix for that program. Specifically, it corresponds exactly to the first row of a scheduling matrix, where all coefficients are set to 0, and the one of the last column (corresponding to the constant part) is set to the chosen partition number.

To compute the rest of the scheduling matrix, P_{Lu}To is called and provided the partial scheduling matrix as well as the dependence graph. As an output, a full scheduling matrix is computed by applying the tiling hyperplanes approach.

2.3 Implementation

We use the code generator C_{Loo}G [1] to apply the scheduling function to the input program. We developed a wrapper around C_{Loo}G to ease the development of automatic parallelization techniques. This wrapper allows to:

- work on the polyhedral representation of the program (C_{Loo}G);
- work on the AST representation of the program (CLAST: C_{Loo}G Abstract Syntax Tree);
- automatically synchronize the two representations (via code generation or polyhedral extraction);
- automatically synchronize the dependence graph corresponding to the application of a scheduling function to the program;
- automatically detect parallel loops, at any level.

It is worth noting that a translator from CLAST to GIMPLE is currently available in Graphite, and that the C_{Loo}G polyhedral representation is also being constructed in Graphite, see Deliverable D3.9.

Finally, we plugged the two libraries LetSee² and P_{Lu}To³ to this wrapper, to realize the complete toolchain.

2.4 Experimental Results

We report a detailed analysis for the GEMVER benchmark, a simple yet representative loop nest belonging to the 2D signal-processing domain (of interest to the ACOTES project). For this code, we generated all the 8 different legal splittings, and applied P_{Lu}To on each partition resulting from a given split. The resulting parallel outer-most loops (the “tile” loops, as opposed to the inner “point” loops) are tagged with a pragma in OpenMP (static scheduling is used). It is worth noting that all-fused and all-distributed strategies are both legal here, and hence present in the considered space. The performance distribution is reported in Figure 2(a), for N=4000 on a bi-Opteron⁴ running 4 threads, using GCC version 4.3 -O3 (with automatic vectorization).

We plot on the y axis the execution time of each version, in second. A first observation is that the range of the execution time is large: there is a $13\times$ difference between the best and the worst point. Also, only 2 out of 8 are significantly improving the performance of the original code, as compiled with ICC version 10.1 (the proprietary compiler reference for this platform, represented by the first horizontal bar), with a best speedup of $2\times$. This highlights that pure random techniques on larger spaces (typically coming from larger programs) would fail to converge in a reasonable time to the best solution. Finally, the slowest point is actually corresponding to maximally fuse all statements, which is the default heuristic of P_{Lu}To. This highlights the benefit of our approach: on the GEMVER example, maximal fusion is forbidding to improve performance, while our technique eventually discovers a $2\times$ speedup with ICC.

²<http://www-rocq.inria.fr/~pouchet/letsee/>

³<http://www.cse.ohio-state.edu/~bondhugu/pluto/>

⁴2 x 2.6 GHz Dual-Core AMD Opteron

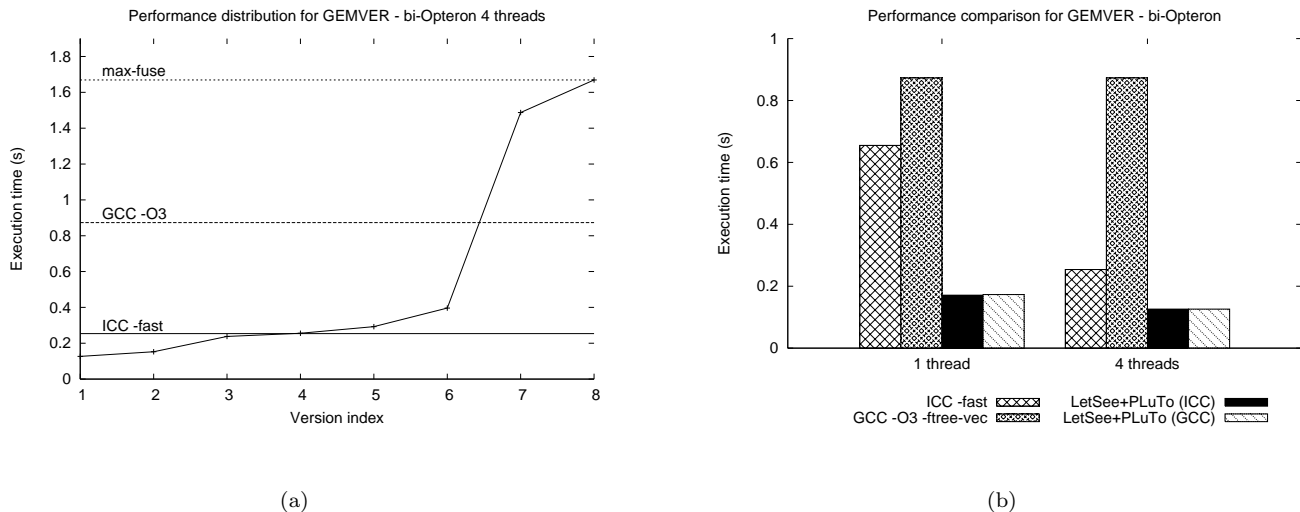


Figure 2: Performance Comparison for GEMVER

We report also in Figure 2(b) the execution time, for 1 and 4 threads, of ICC (version 10.1) and GCC (version 4.3) on the original code, and the best found version with LetSee+PLuTo using either of those compilers as a back-end. A key observation here is that our technique is able to remove most of the performance gap between ICC and GCC, which reach almost the same performance as back-ends of our tools, while there is up to a $3.5\times$ difference between them on the original code (note that GCC 4.3 -O3 does not perform automatic parallelization, unlike ICC 10.1).

We also conducted similar experiments on an Intel Core2Duo, from which the same observations were reported. Moreover, on Core2Duo we are able to achieve a $2.48\times$ speedup over the equivalent sequence of MKL calls, with 4 threads. This allows to hope for a solid overall performance discovered by our technique, closer to the actual machine peak than more traditional approaches.

3 Semi-Automatic Task-Level Optimization

On more complex benchmarks, it is not considered practical to rely entirely on automatic parallelization (due to non-static control-flow and resource management, in particular). The ACOTES project proposes the SPM pragmas and C extensions to allow the programmer to decompose a streaming application into a set of communicating tasks, amenable to semi-automatic parallelization (e.g., adaptation of the parallelism to the target platform, dealing with locality and mapping issues).

Starting from a streaming application written according to the SPM, two major task-level, compilation-time optimizations need to be performed to increase performance portability:

- task fusion;
- task blocking or coarsening.

3.1 Task Fusion

Task fusion consists in statically scheduling a producer with some of its consumers, avoiding runtime scheduling and synchronization overhead, and avoiding memory copying or communication costs. It requires an exact characterization of the instancewise flow (producer-consumer) dependences. This exact characterization can

be done in the polyhedral model using our affine transformation framework, yet the dependence analysis needs to be enhanced to reason about the SPM semantical constructs.

3.2 Task Blocking

Task blocking (or coarsening) consists in strip-mining the outer time loop of a task, hoisting the synchronization (and communication) primitives towards the outermost time loop to reduce the synchronization grain. It does *not* involve schedule modifications (unlike loop tiling), hence does not require exact dependence information. However, to avoid deadlocks, the dependence distance over any cyclic inter-task dependence must always remain higher than the task blocking factor.

3.3 Internal Compiler Representation

Both optimizations are already supported by our task-level optimization framework. However, the current implementation only supports a fully-automatic parallelization mode, where the source program is written in plain C.

Extending these optimizations to SPM programs require a complex interaction between the SPM semantics, its internal compiler representation, the task-level optimization framework, and the ASM performance model.

For both task fusion and blocking, inter-task dependences need to be analyzed at compilation time from the SPM program semantics. This cannot be done if the SPM is expanded early to low-level ASM runtime function calls (the ACOLib).

INRIA and NXP collaborate on the definition of an intermediate language to represent low-level SPM semantical primitives alongside GCC's GIMPLE internal representation. This will be part of the optional outcome of the ACOTES project, but is still considered a high-risk and long-term research and development project. In the short term, we thus plan to recover precise dependence information via task/function inlining (performed by GCC) and careful mapping of SPM primitives to low-level array/pointer operations. This mapping will need to preserve enough pointer aliasing information to allow for a precise disambiguation and dependence analysis. We will conduct this integration step during the third year of the project, enabling the processing of SPM programs through our task-level optimization framework.

4 Pipeline Parallelism and Distributed Memory Platforms

The previous sections deal with static control parts (SCoPs) being automatically or semi-automatically parallelized on shared-memory platforms, following a bulk-synchronous data-parallel execution model. Targeting the ACOTES architecture platforms require the automatic generation of communications, and scalable parallelization can only be achieved with a memory- and load-balancing-conscious pipelining of the stream-computing tasks. We are thus designing an extended task-level parallelization and optimization framework, also based on the polyhedral model, yet supporting more expressive parallelization schemes (mostly, pipeline parallelism to avoid spurious barrier synchronizations). Beyond affine scheduling and loop tiling, this extended framework supports affine partitioning with synchronization-avoidance and storage-mapping optimization heuristics [3]. It will be presented and released in the third year of the project.

Notice that we do not consider data and code placement issues on distributed memory architectures in WP3 (this is left to the programmer, and dealt with in WP2 and WP5).

References

- [1] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *IEEE Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT'04)*, pages 7–16, Juan-les-Pins, Sept. 2004.

- [2] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelization and locality optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, Tucson, Arizona, June 2008. ACM Press.
- [3] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *ACM. Conf. on Principles and Practice of Parallel Programming (PPoPP'01)*, pages 102–112, 2001.
- [4] M. Palkovič. *Enhanced Applicability of Loop Transformations*. PhD thesis, T. U. Eindhoven, The Netherlands, Sept. 2007.
- [5] L.-N. Pouchet, C. Bastoul, J. Cavazos, and A. Cohen. A note on the performance distribution of affine schedules. 2nd Workshop on Statistical and Machine learning approaches to ARchitectures and compilaTion (SMART'08), Göteborg, Sweden, January 2008.
- [6] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 90–100, Tucson, Arizona, June 2008. ACM Press.