The handle http://hdl.handle.net/1887/25180 holds various files of this Leiden University dissertation.

**Author:** Rietveld, K.F.D.
**Title:** A versatile tuple-based optimization framework
**Issue Date:** 2014-04-10
CHAPTER 13

Summary & Future Perspectives

In this thesis, we have investigated a solution for the unification of imperative and declarative codes. This has resulted in the forelem intermediate representation. Declarative codes can be represented in the form of loops performing tuple accesses with simple loop control. These loops are especially suited to be made part of the workflow of traditional optimizing compilers. In fact, we have shown that many established compiler transformations can be re-targeted to operate on forelem loops. Through the application of these transformations, queries expressed in forelem loops can be optimized to performance comparable with that of contemporary state-of-the-art database systems (as described in Chapter 6) and transformations can be carried out that intertwine execution of the application code with the execution of the data access code as was illustrated by the examples in Chapters 3, 4 and 5.

The automatic reduction methodology discussed in Chapter 7 exemplifies the strength of the design of the forelem framework. The described prototype compiler, powered by the forelem framework, is capable of eliminating up to 90% of the instructions executed by two web applications. This process reduces database applications to their essence and unlocks more possibilities for the optimization of the performance of these applications.

Part of the automatic reduction process is to create a local copy of the data that is operated on. For large, distributed, deployments of web applications this may form a bottleneck. Whether this is the case depends on the application. In the case of e-business processing, the workloads are often read-dominant [64], so a local data copy is advantageous. On the other hand, when write actions are performed that trigger updates in all local copies of the data, the performance advantage of the vertically integrated application is most likely lost. To provide a solution to these problems, an analysis method is needed to determine when vertical integration and performing all reads on a local copy of the data is beneficial and when it is not. In Chapter 8 an initial study towards such an analysis was presented. Given a set of query mixes executed by an application it can be determined whether vertical integration will be beneficial.
The application of the forelem framework in the domain of database applications has been successful. Due to the generic nature of this framework, we found that this framework is also applicable to other application domains. In Part II of this thesis the application of this framework in a number of these different application domains has been studied. A focus was the optimization of the used data storage format together with the code operating on this data. Data structures are in fact reassembled, by first translating the data structure into tuples on which a sequence transformations are performed leading to the generation of a new data storage format. Transformations that affect the generation of the final data storage format were described in Chapter 9.

These transformations set up a large search space of possible loops and data storage formats that can be generated from a single initial forelem loop. In Chapter 10 this search space was characterized and explored. For a number of sparse matrix kernels, it was shown that through effective exploitation of this search space an optimized code can be found that in most cases outperforms implementations of this routine found in sparse algebra libraries, but is at least on par in performance.

Chapter 11 introduced the ready clause, which allows dependencies between tuples to be described in a natural way. As a consequence, sets of tuples that can be processed in parallel can be deduced in a straightforward manner. Two execution models were presented for the parallel execution of forelem loops. Using these techniques, a parallel code for triangular solve could be automatically deduced and was shown to be competitive in performance to hand-optimized parallel implementations of triangular solve.

Finally, Chapter 12 discussed how distributed execution of forelem loops can be controlled by a compiler. Next to the optimization of local data storage formats that have been discussed in Part II of this thesis, this would give the forelem framework the capability to optimize the data decomposition and distribution as well. These capabilities were used to give an initial impression of the viability of the forelem framework to optimize Big Data applications.

Although two different classes of applications were discussed in this thesis, database applications and sparse matrix algebra, the optimization methodologies were based on a single intermediate representation. Many optimizations that were proposed for database applications in Part I of this thesis can also be applied to the irregular applications discussed in Part II of this thesis and vice versa. This gives rise to a lot of avenues for further research, a number of which will be briefly discussed in this chapter.

Based on the automatic global integrated optimization process that performs vertical integration of database applications, there are several directions for future work that are interesting from both a scientific and engineering point of view.

At the core of the global optimization process is the ability to express (SQL) queries in terms of forelem loops. The majority of SQL queries, including nested queries and group-by queries, can be written as forelem loops using the techniques described in this thesis. For a production system, future work is needed is support these parts of the SQL standard that cannot yet be expressed in the forelem intermediate representation.
Also the new common intermediary level, where application and data access codes are combined, should be further exploited. We intend to look into the application of existing code transformations at this new level as well as to investigate new basic code transformations that enhance the performance of the application. Existing transformations to be investigated include traditional loop transformations and vectorization. Furthermore, using the established technique of Def-Use analysis, elimination of unused query results and redundant queries is obtained for free. New basic code transformations will be researched, that can optimize code patterns found in database applications that cannot be handled by the existing compiler techniques. These transformations will rely on combining knowledge from both the application program and its queries. This may lead to new sophisticated techniques for the automatic optimization and merging of what were originally separate queries.

Chapter 3 proposed the Loop Collapse and Reverse Loop Collapse transformations. These transformations affect the schemas of the tables in addition to the loop structure of the code. By applying these transformations, the schema of the tables used by the application can be optimized based on the operations that are performed on the tables by the application code. The new intermediary level provides a good test bed to study the effectiveness of these transformations. Techniques can be developed to automatically optimize schemas of database tables, based on the different queries that are performed on these tables and the further processing of the data by the application code. Furthermore, the relation of these techniques with materialization discussed in Chapter 9 should be investigated.

In the context of sparse matrix algebra, we want to examine the effects of exploiting specific sparse matrix characteristics in the transformation process. This may result in the automatic generation of hybrid data storage formats. We also intend to investigate the effects of combining the loop blocking transformation with the materialization and concretization transformations. Materialization of a loop nest that is blocked should result in a blocked data storage format. The investigation should focus on whether further transformations can be devised to result in new forms of blocked data storage formats and the performance characteristics of the different formats should be explored.

The experimental evaluations presented in Part II of this thesis have focused on sparse matrix algebra. The optimization techniques that have been described are generic in nature, however. To demonstrate the effectiveness of our approach, we intend to conduct an extensive experimental evaluated of the proposed optimization techniques on a large variety of irregular codes. This will include further, more complicated, sparse matrix algebra routines, as well as routines from different domains such as graph algorithms.

Chapter 12 presented an initial overview of the extensions of the forelem framework to be able to control distributed execution of forelem loops. Through this control, different codes making use of different data decompositions and distributions can be generated automatically. The end goal is that within the forelem framework, from a single initial representation of a computation in the forelem intermediate representation, different variants can be generated for serial, locally parallel (multicore CPU or GPU), distributed and combined locally parallel and distributed execution. These different variants make use of different local data
storage formats and different data decompositions and distributions.

Methods for the automatic optimization of data decomposition and distribution within the forelem framework have not been investigated and remain a topic for future work. For Big Data applications it must be taken into account however that often the data to be processed is already stored and the data generation code is not part of the optimization process. The volume of the stored data may prohibit preprocessing, reformatting or redistribution of the data to better suit the computation. Strategies have to be investigated to find a middle ground between the reformatting of the existing data and the optimization of the computation. Note that, although many of such capabilities can be implemented in existing systems such as Hadoop, or are already implemented (e.g. binary storage in between jobs), there is at this moment no possibility for automatic optimization because many of these details are obscured from optimizing compilers.

At all levels, substantial improvements are possible in the code generators. For example: the code generator for CPU code can be extended with support for multi-core processing and SIMD instructions; the code generator for GPU code can be improved with optimizations to address coalescing, memory banking and interleaved computation; and the code generator for distributed codes can make better use of MPI by using better performing MPI primitives and to support interleaved computation and communication. Finally, the generated MPI code should also be made capable of fault tolerance by supporting continued execution if one of the nodes failed. Next to being able to handle a static loop schedule determined at compile-time, the code should be able to handle some amount of dynamic scheduling as well to allow for load balancing and failure recovery.

The forelem framework is presented as a versatile framework that unifies optimization of imperative and declarative codes. For the different application areas of this framework, different transformations have been devised. While such transformations were defined within the context of a particular application area, these transformations are generic in nature and are well of use in other application domains. For example, transformations that have been described in Part I in the context of database applications can also be applied on generic codes described in Part II.

Finally, it is interesting to see how techniques developed specifically for a single application domain can be used for the optimization of problems from another domain. So far, the focus of this research has been on the domains of database applications and sparse matrix algebra. It will be exciting to see whether investigations into other application domains will unveil further code transformations that can be expressed in a generic nature and that will subsequently result in improved performance of code from other application domains. This in particular shows the strength of the forelem framework as a versatile framework for the specification of code transformations and the optimization of tuple-based codes.