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 1

Introduction

In imperative programming, the computation to be carried out is specified step by step. Declarative programming languages, on the other hand, allow the specification of what data must be retrieved, but not how. Naturally, these two different programming paradigms give rise to different manners by which codes expressed in these paradigms are translated to code that can be executed on a Central Processing Unit (CPU) and different manners by which these codes are optimized. Whereas code optimization for imperative programming languages focuses on reordering the steps by which the computations are carried out without affecting the final results of these computations and selecting efficient instructions to encode these computations for a particular CPU code, optimization for declarative programming languages focuses on determining an efficient execution plan for retrieving the specified data.

Application programs are in general written in an imperative programming language. Declarative programming languages are used to write codes, or queries, that specify data to be retrieved from a Database Management System (DBMS). Such queries can be performed by application programs and the retrieved data can be further processed by these application programs. Applications that perform such requests are referred to as database applications. In database applications, the data processing code is written in an imperative programming language and the data retrieval request is written in a declarative programming language. As a consequence, data processing code and data retrieval code undergo different and independent optimization procedures. For example, queries are optimized and executed in a DBMS, independent from the application code that further processes the result data.

In this thesis, a novel approach is presented for the optimization of data-intensive applications. This approach is implemented by the forelem framework and solves three important problems of existing approaches. Firstly, this framework unifies the programming of transactional (database) applications and the programming of other kinds of applications, such as high-performance parallel computational codes. An important difference between these two forms of programming is the kind of optimization that is performed in a DBMS and that is per-
formed on application codes by traditional optimizing compilers. The DBMS (query) optimizations are necessary to efficiently retrieve the desired data from a database, especially when the data set does not fit in main memory, while the traditional optimizations performed on application codes are vital for the generation of efficient machine code. The forelem framework unifies these seemingly distinct fields of programming by expressing queries as a series of array accesses governed by simple loop control, which are subsequently optimized by traditional optimizing compiler techniques accomplishing results similar to query optimization.

Secondly, the forelem framework provides a solution for the (semi-)automatic problem-specific optimization process for applications, which runtime is very much dependent on the underlying characteristics of the problem to be solved. Problem-specific optimizations often consist of the selection of a good data layout or data storage method. In general, compilers do not have the capability to optimize data layout or storage. Although techniques to extend compilers with the ability to optimize data structures have been researched, they have not yet found a widespread use. With the forelem framework, a universal approach is introduced for the optimization of an application’s data layout and storage. By incorporating details about the data access performed by the application into the optimization process, the application and its data access method can be synchronized. This synchronization leads to a better alignment of the application’s computational loops with the order in which data is accessed.

Thirdly, the unification of transactional programming and other kinds of programming enables the vertical integration of application code and data access frameworks. Applications typically access data through a framework that abstracts away peculiarities of accessing a particular file format, database system or distributed file system. Such frameworks inhibit optimizing compilers from potentially optimizing data access as performed by an application. The forelem intermediate representation provides a generic way for expressing data access, based on series of array accesses and simple loop control. In vertical integration, the data access operations that are performed through a data access framework, are expressed in this generic intermediate representation. As a result, the data access code is combined with the surrounding application code in the optimization process. Traditional analysis methods, such as Def-Use analysis, will detect and eliminate data access of which the results are unused, or will detect related data accesses that can be combined. For database applications, methods that optimize both the application and data access codes (in the form of queries) have been proposed [66, 37, 21]. However, because both codes are kept separate, these methods often rely on pattern recognition to detect possible code segments where a specific optimization can be applied. In the forelem framework generic optimizations can be applied, which unlock many more potential optimization opportunities.

Since the forelem framework was initially envisioned for database applications, it considers data to be stored as (multi)sets of tuples. So, the forelem framework operates on a tuple space. Due to the generic nature of the forelem intermediate representation, the framework is applicable to many different application areas. Examples of these application areas are: (1) vertical integration of database applications: queries in a database application are replaced with code segments that
evaluate these queries and directly access a data store, subsequently, the application and data access codes are optimized together (Chapter 7); (2) reduction of energy consumption: a reduction in energy consumption up to 90% can be obtained by the application of vertical integration coupled with aggressive optimization (Chapter 2); (3) exploration of the optimization search space: the forelem intermediate representation expresses data access using simple loop control, which enables re-use of traditional compiler loop transformations and in conjunction with this new transformations and heuristics are to be developed, as well as methodologies to effectively explore the optimization search space (Chapter 10); (4) data reformatting: an optimization process incorporating details about the application code and associated data access is able to optimize storage layout and format for this particular application (Chapter 9); (5) Big Data: new optimization techniques for Big Data applications can be devised using distributed forelem loops combined with automatic optimization of the data distribution and layout (Chapter 12); (6) universality: data access expressed in many different methods, such as SQL or MapReduce, can be translated to the forelem intermediate representation and vice versa.

This thesis describes a versatile tuple-based optimization framework. A versatile framework, because it is capable of optimizing traditional imperative codes (such as sparse matrix computations) as well as declarative codes (such as database queries). Although the framework is tuple-based due to initially being designed for the optimization of database applications, tuples are especially suited as an elementary data representation because they are the most fundamental objects that allow multiple values that are related with each other to be coupled. All data structures can be represented as tuples, in fact, computer memory can be expressed as tuples by creating pairs of address and an associated value. From a representation of the data in the form of tuples, many different data layouts can be automatically generated.

The first part of this thesis discusses the application of the forelem framework to database applications. The unification of transactional and imperative programming that is achieved by this framework enables the vertical integration of application code and data access frameworks. In past research also methods have been described that tried to overcome the division between DBMS (declarative) and application (imperative) codes. Such methods included object-oriented data systems integrated with object-oriented programming languages and specific database programming languages. The importance of this unification has been raised in the literature. For example it has been asserted that compilers for database programming languages must be extended to include database-style optimization in order to produce high performance codes [62] and a call has been made for a single expressive intermediate language instead of the use of specific representations for database queries and generic program codes [38]. Unfortunately, such approaches failed to get traction in the database community and it is still common practice to specifically program the database access code, that places requests for data retrieval, while keeping the division in place [37]. A major advantage of the approach described in this thesis is that it is not necessary to rewrite existing database application code in order to benefit from vertical integration.
As an initial target of our vertical integration efforts, we selected an important class of database applications: web applications. Web applications are these days ubiquitous and empower the modern, interactive World Wide Web. The increase of such web applications led vendors to significantly scale up their banks of web and database servers in order to handle all incoming requests. This has resulted in ever increasing complexity of distributed server architectures. Optimization of such systems not only leads to faster response times, but also to the ability to handle similar loads with a smaller amount of servers. For the implementation of this vertical integration, the forelem framework is introduced. The forelem framework is ideally suited to support an integrated, holistic optimization process as will be discussed in the first part of this thesis.

As will be described in Chapter 2 of this thesis, this global integrated optimization process is capable of aggressively optimizing web applications, eliminating up to 90% of the instructions that need to be executed without affecting the final result, resulting in a tremendous increase in performance. The further chapters in Part I of this thesis describe a framework within which this optimization process can be carried out. The main constituent of this framework is the forelem loop, which specifies iteration of a subset of a multiset of tuples, and transformations that can be carried out on these loops, as described in Chapters 3, 4 and 5.

The generic nature of the forelem framework also allow the developed techniques to be applied in other application domains. This is the focus of Part II of this thesis. In this part, the use of the forelem loop is explored for the optimization of irregular applications. To accomplish this, the computation to be optimized is expressed in terms of tuples: the data that is operated on is translated to tuples, computations are translated to forelem loops processing these tuples. Chapter 10 focuses on sparse matrix computations. By expressing these in terms of tuples, the computation is reordered as well as the way in which tuples are stored is reorganized. As will be described in the chapter, this results in a large search space of different variants of the same computation. In Chapters 11 and 12, the techniques are further generalized to irregular applications and extended to be capable of expressing distributed computations.

In this chapter, the necessary background knowledge for this thesis is introduced, as well as work that is related to this thesis. This background knowledge has been divided into three fields, Optimizing Compilers, Database Systems and Sparse Computations, that will be discussed in turn.

### 1.1 Optimizing Compilers

Code written in traditional compiled languages such as C and Fortran, is directly translated to machine code for a particular target architecture. The performance of this resulting machine code can be optimized by making use of optimizing compilers, that perform code transformations that are expected to improve the performance. In general, two levels can be distinguished at which optimization may take place. The optimizations can be performed on the structure of the original code, in which loops are still explicitly exposed. Examples of such transformations are constant propagation, common subexpression elimination and loop
transformations such as loop blocking and loop interchange. The other level at which optimization can be performed is the code generation level, where executable code for a particular target is generated from an internal representation of the program code. At this level, optimizations are performed such as instruction selection and scheduling. These optimizations are generally very specific to the instruction set architecture (ISA) of the target architecture.

Within the forelem framework, optimizations are defined that operate on the loop structure. These optimizations are based on traditional optimizing compiler transformations that are described in the literature. These transformations need slight adaptation to account for the semantics of the forelem loop. In this section, common optimizations that are carried out by optimizing compilers are introduced that will be referred to throughout this thesis. Optimizations that occur at the code generation level are not discussed in this thesis. These optimizations can be conducted on top of forelem transformations. Therefore, we will rely on common compiler tools, like for instance LLVM [63], to implement these backend transformations.

An important class of transformations are loop transformations, a number of which are now discussed. Loop blocking is often used to improve cache re-use of a loop nest by processing data in blocks. In [60] the influence of the stride of data access and the size of the blocks on cache efficiency is discussed. When Loop Blocking is used in conjunction with Loop Interchange, the data locality of a program can be further improved [35].

Using the Loop Interchange transformation it is possible to enable vectorization and/or parallelization by moving dependence cycles, and to increase cache re-use by improving the locality of the code. Loop Interchange can only be performed under certain conditions. Because the order in which statements inside loops are executed is changed by an interchange, the interchange of loops is valid only if the new order of statement execution preserves all dependencies of the old order. We will illustrate this using two examples from [104].

```
for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++)
```

The loop on the left can be interchanged. This Loop Interchange will eliminate the dependency in level 2 (iterated by j) which makes it possible to vectorize the inner loop.

Contrary, the loop on the right cannot be interchanged. For example for \(i = 3, j = 3\) the value \(A[3][4]\) is read and the value \(A[4][3]\) is written. The value \(A[3][4]\) is generated by the iteration \(i = 2, j = 4\). In fact this statement at iteration \(i = 3, j = 3\) has a dependency on the statement at iteration \(i = 2, j = 4\). With the current nesting assuming standard execution order, the iteration \(i = 2, j = 4\) will be executed before \(i = 3, j = 3\). However, when the two loops are interchanged, or swapped, this will no longer be the case. The dependency has then been broken which makes this instance of Loop Interchange invalid.

To formally verify whether transformations are valid for a loop nest, a technique known as data-dependence analysis [58, 3, 5] is employed. The analysis re-
sults in data-dependence relations which reflect the constraints on the statement execution order.

Three classes of dependencies are usually distinguished: (i) a true dependency, where a value is first defined and then used (Read-After-Write), (ii) an anti dependency, when a value is read and then written (Write-After-Read), and (iii) an output dependency when a value is written and written again (Write-After-Write). A dependency between loop statements is either loop independent, indicating that the dependency holds for equal iteration vectors, or loop carried, meaning that the dependency holds for different iteration vectors of the loop.

When there exists a data dependency in the original loop that no longer exists in the loop-interchanged loop, this is said to be a loop-interchange preventing dependency. Observe that this was indeed the case in the second loop we discussed above.

A related analysis is def-use analysis [2, 50]. In this analysis statements are analyzed to see whether they are a definition (an assignment) or a use of a value. From this information definition-use and use-definitions chains can be set up for a variable in a basic block. This can be used to reason whether a variable is assigned a constant or whether an assigned (defined) variable is used at all, etc. Optimizations such as constant propagation and variable substitution use this analysis.

For a more thorough and formal treatment of these analysis techniques, we refer the reader to the cited literature as well as to [104] for a concise overview.

Two loops (at the same level if contained in a larger loop nest) can be merged into a single loop under certain conditions using the Loop Fusion transformation [52]. For now, we only consider serial loops. Consider the separated loops and the fused loop:

```c
for (i = 0; i < 100; i++)
A[i] = B[i] + C[i];

for (j = 0; j < 100; j++)
D[j] = A[j] + X[j];
```

```c
for (i = 0; i < 100; i++)
{
  A[i] = B[i] + C[i];
  D[i] = A[i] + X[i];
}
```

Loop Fusion is defined if the loops to be fused iterate the same iteration space and valid if there does not exist a dependency which prevents fusion of serial loops. Fusion is prevented if there is a dependency from a use in the second loop to a definition in the first loop for which \(i > j\) holds true. When the loops would be fused, the value would be read before it is written. This can be verified by replacing the read of \(A[j]\) in the second loop with \(A[j + 1]\) and similarly in the fused loop.

Another transformation on loops is Loop Collapse. The Loop Collapse transformation is used to rewrite two levels of loops as one level of loop by using a one-dimensional representation of a two-dimensional array [100]. Loop Collapse is mostly used to enable vectorization and can only be applied on serial loops, that is, no loop-carried dependencies are present. Because two loop levels are collapsed into a single loop level, the vector length that can be used is increased [104]:

```c
for (i = 0; i < 100; i++)
{
  A[i] = B[i] + C[i];
  D[i] = A[i] + X[i];
}
```
1.1. Optimizing Compilers

int A[100][100], B[100][100], C[100][100];

for (i = 0; i < 100; i++)
for (j = 0; j < 100; j++)
    A[i][j] = B[i][j] * C[i][j];

Note that in languages that support vector statements, such as Fortran, the resulting single-level loop can be written as a single statement.

Other examples of loop optimizations are loop unrolling, loop rerolling, loop fission and iteration space morphing. Zima [104] provides a comprehensive treatment of this kind of compiler optimizations. Finally, [95] discusses how loops that traverse a data structure using a pointer can be turned into counted loops operating on arrays that indirectly access the original data structure. This technique enables the application of existing loop optimization methods which are often not successful when applied on the original pointer-traversing loop.

Loop Invariant Code Motion is a kind of common subexpression elimination where statements which are invariant under the inner loop iteration variable can be moved to an outer loop or completely out of the loop nest. If the statement is, for example, a memory load, then the pressure on the memory bus can be significantly reduced. A simple example of the application of this optimization is:

```
for (i = 0; i < 100; i++)
{
    Y = 154;
}
```

Vectorizing compilers have the capability to recognize reduction operations. Reduction operations are operations such as computing the sum or product of all elements of a vector, or determining the minimum or maximum element of a vector [77]. The compiler will replace a loop performing such an operation with a vector statement performing the same operation. The vector statement is implemented using specific vector instructions. Consider the following example, taken from [77], which computes the sum of a vector A:

```
DO I = 1, N
    A(I) = B(I) + C(I)
END DO

A(1:N) = B(1:N) + C(1:N)
ASUM = ASUM + SUM(A(1:N))
```

The loop, written in Fortran, can be replaced with two vector statements. The SUM function used in the example returns the sum of the vector provided as argument.

Scalar Expansion is a transformation that is typically used to enable parallelization of loop nests. Consider the following loop:

```c
for (k = 1; k <= N; k++)
{
    tmp = A[k] + B[k];
```
C[k] = tmp / 2;
}

Due to the loop-carried anti-dependency of tmp, subsequent iterations cannot write to tmp before tmp has been used in the assignment to C[k]. This is solved by the Scalar Expansion transformation which expands the scalar tmp to a vector:

```c
for (k = 1; k <= N; k++)
{
    tmp[k] = A[k] + B[k];
    C[k] = tmp[k] / 2;
}
```

Now that the loop-carried dependency has been broken, the loop can be parallelized.

In Global Forward Substitution, the right-hand side of an assignment statement is substituted into the right-hand side of other assignment statements [58]. This potentially eliminates flow dependencies, but also eliminates temporary variables in a subsequent dead code elimination phase. The well-known Constant Propagation optimization is considered a special case of Forward Substitution [77]. An example of Forward Substitution, adapted from [77], is:

```c
NP = N + 1
for (i = 1; i < N - 1; i++)
{
}
```

```c
for (i = 1; i < N - 1; i++)
{
}
```

In the code on the left side, there is a dependency between the two statements within the loop body. The compiler can typically not reason about the address of A[NP], thus it is possible that A[i] overwrites the value used by A[NP]. The code on the right side is the result after Forward Substitution. Now, NP has been substituted with N + 1. From the expression the compiler can determine that the assignment to A[i] will never reach A[N + 1], because the inclusive upper bound of the loop is N - 2. So, after Forward Substitution, the dependency no longer holds.

### 1.2 Database Systems

Part I of this thesis describes a global integrated optimization process for the optimization of web applications. The web applications that are used as a starting point, make use of a Database Management System (DBMS) for accessing persistent data. Within this optimization process, the application and data management codes are integrated with each other. This section briefly discusses work that is related to this integration process.
The purpose of a DBMS is to store data at a central location, concurrently accessible by multiple clients. It is the responsibility of the DBMS to safeguard the data, such that it does not become corrupted or inconsistent. To this extent, database systems implement the ACID properties: 
atomicity, consistency, isolation and durability. This ensures, respectively, that database transactions are always atomic; the database is always left in a consistent state; transactions that are running concurrently are fully isolated; and that committed data is guaranteed to be stored in a way such that it cannot be lost due to system crashes or power loss in the future. Clients can access this data by submitting queries, written in a declarative programming language such as SQL. A DBMS needs to translate these queries to a query plan, or execution plan, that specifies which actions need to be carried out and in which order, to process the query. During this translation stage transformations can be applied to the query plan to optimize the query execution, a process known as query optimization.

Work that is related to the global integrated optimization process as described in this thesis roughly touches two areas in the field of databases. The first area concerns the integration of the usage of database systems into imperative programming languages. The second area concerns applying optimizing compiler technology to query planning and optimization. In this section, both areas are explored in turn.

1.2.1 Integration of Database Systems Query Processing Into Imperative Programming Languages

Integration of query processing of database servers into programming languages is an area that has been under research for a long time. In this area, this often is referred to as the “impedance mismatch” [25, 65]. The impedance mismatch refers to the mismatch of elements of procedural programming languages and declarative database languages, such as procedural types versus database types, optimizations in procedural programming languages versus database query optimizations, concurrency versus transactions, etc. It is this mismatch that highlights the key problem in integrating usage of a DBMS in a programming language. Cook et al. [25] give a thorough review of the definition of this problem and approaches to solve it.

There are many examples of techniques that try to overcome this impedance mismatch. Most of these solutions focus on the integration of the database application programming interface (API) into programming languages and setting up a mapping between application code value types and database value types. Note that most of these techniques are tools that aid programmers to more effectively create database applications, and do not have as goal to support a integrated optimization process as is proposed in this thesis.

LINQ, for example described in [31], extends the programming language such that declarative queries can be naturally expressed. Programmers can express queries in LINQ without having to know how the query will be executed. Queries can be executed on a variety of data sets, for example on collections in main memory internal to the program and also on a DBMS. When a query expressed in LINQ is compiled, no code is generated to actually execute the query. This
is handled at run-time, when a query parse tree is built for the query. In case LINQ is used with a DBMS, LINQ will generate a set of SQL queries that are sent to the DBMS for execution. Given that the query parse tree is built at run-time, opportunities to extensively merge the query code with the application code at compile-time are not fully exploited.

Systems similar to LINQ also exist for example for the C++ language [39]. The focus is on solving the impedance mismatch and checking for correctness and security concerns (SQL injection attacks) at run-time. This is much like the static analysis of the correctness of SQL statements proposed by [20, 27]. In these papers, analyses are described to find security problems in the application code’s usage of database APIs and to incorporate the time spent in the DBMS in the application profiling respectively. The latter also supports the rewriting of SQL queries, which is for example possible when it is detected that three columns are projected in the query but only two of those are used in the code. In such cases, the programmer can be alerted by the development environment. Also, an approach to static analysis of strings containing SQL fragments has been described, which results in compile-time warnings of problems in these fragments [96]. This analysis is based on the SQL grammar specification and the schemas of the target database.

Note that these systems solely concentrate on the facilitation of integration of the database API into the programming language realm. They do not concern the impedance mismatch in optimization as described in [25] and do not propose techniques for the explicit break down of layers between the application and database codes.

A different way to integrate database and application codes is to express database queries in the same imperative language as the application code. This was the approach taken by a specific class of programming languages known as Database Programming Languages (DBPLs). These languages are characterized by the fact that they include the ability to iterate through sets. In order to obtain good performance it is critical that DBPL compilers are extended to include database-style optimizations, such as join reordering. Initial work into such compile-time optimizations is described in [62]. The described transformations make standard transformation-based compilers capable of optimizing iterations over sets that correspond to joins. This work was later extended to include transformations that enable the parallelization of loops in DBPLs [61].

The Tycoon project aimed to replace special-purpose representations for queries, programs and scripts with a single expressive intermediate language [38]. This intermediate language, the Tycoon Machine Language, was based on continuation passing style. Continuation passing style is a functional programming construct wherein a function call has as last argument another function to call (named the continuation) once the called function has finished execution [89]. The language was used to move towards an integrated database language where user-defined code and query expressions are fully integrated. No final results or benchmark figures were published [25].

Contrary to expressing application and database codes in a common representation to perform integrated optimizations, it is also possible to devise transformations that operate on both the original application and database codes simulta-
neously. Such transformations have been researched by several groups. Holistic transformations for web applications are proposed in [66, 37]. The papers argue that tracking the relationship between application data and database data might yield advancements. It is exactly this relationship that we aim to exploit with the global integrated optimization methodology that is described in this thesis, however, we do not track the relationship, we rather eliminate this relationship by integrating the application and DBMS codes.

A similar holistic approach for Java code bases is described in [21], which motivates the approach by stating that rewrites of queries and programs are done independently by the database query optimizer and the programming language compiler. This independence leaves out many optimization opportunities. Their approach centers around a tool which aims to bridge this gap by performing holistic transformations on the program code and queries.

Cheung et al. describe a system, StatusQuo, to optimize the performance of database applications written in Java accessing a database through JDBC or Hibernate by considering both the application code as well as the queries [22]. Similar to us they state that the hard separation between the application and database code often results in applications with suboptimal performance. The system is capable of automatically partitioning the database application into a Java and SQL code, for optimal performance. To accomplish this, it may rewrite SQL into Java code, or vice versa. Whereas StatusQuo translates imperative code into a declarative form, our system translates declarative code into an imperative form and generates executable code from this imperative form.

The UltraLite system, described in [102], combines application and database logic together in one program. Given a program using embedded SQL for the UltraLite system, the query is compiled to C code which executes this query. This is done by sending the query to the host database server for parsing and optimization. The returned plan is used to generate the C code. The C code makes use of UltraLite run-time functions which implement SQL functionality. Concurrent execution of queries is supported by the run-time library. UltraLite is meant for usage on mobile devices with no hard disk and very little memory.

While the integration of the application and database logic may seem similar to what we are proposing, the integration limits itself to simply placing the application and database logic in a single executable. The generated C code does call functions in the run-time library, which diminishes the possibility to fully integrate the application and database codes at a code level. We propose a much finer grained integration of application and database codes, by intertwining these codes, which is not described in the cited patent.

A different way to perform optimizations that affect both the application code as well as the database queries consists of migrating this integration to the start of the application’s design. GignoMDA is a framework to generate applications and database schemas for different programming platforms based on the Model Driven Architecture (MDA) approach [41]. The novelty of GignoMDA is that it promises to exploit cross-layer optimizations between the different layers of a database application in this framework. Typically, there is a presentation layer (e.g. a Web interface), a business logic layer and a persistence layer (e.g. the DBMS). By giving hints during the UML design process, for example hinting that a table
will mostly be used for read-only access, optimizations based on this hint can be carried out across all layers. The approach taken by GignoMDA only works on newly written applications using the MDA approach. This is different from our proposed code optimization backend, which works on existing codes and aims to do the optimization automatically rather than relying on hints.

### 1.2.2 Query Planning and Optimization

In query planning and optimization an execution plan is devised for a given query. Traditionally, the speed of disk I/O was the main bottleneck in query execution and thus query plans were traditionally optimized to minimize disk I/O. However, with the emergence of main-memory DBMSs such as MonetDB [16, 18, 17], the optimization objective has shifted from optimizing for minimal disk I/O to making best use of the available main memory bandwidth and exploitation of the CPU caches. Similar to optimizing compilers being equipped with techniques to improve caching reuse and utilization of memory bandwidth, query optimizers have to be equipped with such techniques as well.

The importance of optimizing for CPU cache re-use and vector processing was also demonstrated by the X100 query engine for MonetDB [18]. By mapping queries to primitives, simple C functions that apply a given operation on a given input vector in a tight loop, a one to two orders of magnitude performance improvement compared to existing DBMS technology was demonstrated. The sizes of the vectors to be used during the query processing are selected in such a way that they all fit in CPU cache. Optimizing compilers are responsible for compiling the primitives, written in C, into highly efficient code by application of aggressive loop pipelining and vectorization optimizations.

Another methodology is to architecture query optimization in such a way that use can be made of the transformations implemented in optimizing compilers. Recent research has explored the possibility of translating a query to an imperative code that can be processed and optimized by an optimizing compiler. For example, a strategy to transform entire queries to executable code is described in [57]. The technology, called “holistic query evaluation”, works by transforming a query evaluation plan into source code, based on code templates, and compiling this into a shared library using an aggressively optimizing compiler. The shared library is then linked into the database server for processing. Although significant speed-ups over traditional and currently-emerging database systems are achieved, this approach does not include the integration of application and query evaluation codes. Instead, these codes remain separated because the query code is isolated in a shared library. Optimizing compiler technology is used to compile a translation of the query plan into C/C++ code through the use of code templates into efficient executable code.

In [73] a data-centric approach to query compilation is described. SQL queries are translated to relational algebra, which is optimized and from which LLVM assembly code is generated. The query in LLVM assembly code is then executed using the optimizing JIT compiler included with LLVM. The approach is data-centric in that the LLVM code is written such that data can be kept in CPU registers as long as possible for optimal performance. This is given more importance...
than clearly maintaining the boundaries of relational operators. In fact, the relational operators are “blurred” when generating the code and the operators can be spread out over multiple code fragments. This technique results in very efficient query codes. However, note that just the query is taken into account during query compilation and the DBMS/application code split still exists.

DBToaster [1] is described as a novel query compilation framework for producing high performance compiled query executors that incrementally and continuously answer standing aggregate queries using in-memory views. DBToaster compiles queries into C++ code that incrementally maintain aggregate views at high update rates. The focus of DBToaster is on compiling queries to view maintenance code, contrary to the translation of entire queries which return the result of the query as you would normally receive it from a DBMS.

Compiler optimizations have also been used to take on the problem of multi-query optimization. In [9] an approach is demonstrated where queries are written as imperative loops, on which compiler optimization strategies are applied. The use of loop fusion, common subexpression elimination and dead code elimination is described. This work is tailored towards a certain class of analysis queries and not to generic queries. Furthermore, the loop fusion transformation described in the paper works by detecting multidimensional overlap. So, the strategy of loop fusion is used, but not an exact mapping of the traditional loop fusion optimization.

A different approach to multi-query optimization is described in [47], where optimization techniques are applied to the “algorithm-level” of a database program. In the algorithm-level, a query is represented as a sequence of algorithms, e.g., selection, join, that should be performed to compute the query results. The exact implementation of the algorithms is not made explicit at this level. As a consequence, knowledge is required about the implementation of algorithms that can appear in the representation by the optimizer in order to be able to carry out optimizations.

In [11], a “For-Loop Approach” is described to better handle aggregated subqueries that contain where clauses that overlap with the main query’s where clause. By introducing a “for-loop operator” and “for-loop program”, which are used together with relational algebra, the subqueries with overlapping where clauses can be integrated into the main query, eliminating redundant iterations over tables. The proposed “for-loop” operator and programs are meant to be an extension or tool to standard relational algebra.

As can be seen from this summary, many different approaches have been proposed to make DBMSs and especially query optimization more efficient. However, none of the approaches described above exploit (existing) compiler optimizations to their fullest extent.

### 1.3 Sparse Computations

In Part II of this thesis, the use of the forelem loop for the optimization of irregular applications will be explored. In particular, the utilization of the forelem intermediate for code and data structure generation of sparse matrix computations will be
discussed. Contrary to dense matrix computations, sparse matrix computations are irregular applications because specific data structures are used to store the matrix data instead of a regular two-dimensional array. From these specific data structures matrix elements with a zero value are omitted. An example of such a data structure is pointer-linked data structure in which elements that are located in the same row or column are linked to each other. Because these elements may not be placed in memory in consecutive order, the memory access may be very irregular. Irregular memory access in sparse computations is also caused by the selection of a data structure that does not store the matrix elements in the order in which they are accessed by the computation, also yielding ineffective use of the CPU cache next to irregular memory access.

There is a large body of literature on the optimization of sparse computations. More recent work covers overcoming the memory wall in modern CPUs using compression techniques [97, 56], the optimization of sparse matrix-vector multiplication on multicore platforms [98], the optimization for register reuse [44], and the implementation of matrix-vector multiplication on GPUs [14].

Many of the solutions described in the literature, such as specific algorithms and data structures are implemented in sparse algebra libraries. An extensive amount of work has been invested in designing several sparse libraries for different storage formats and computer architectures [14, 87, 81, 10]. While these libraries in general provide an adequate solution and are used frequently by code developers to generate optimal codes, use of these libraries is relying on pre-defined code. The libraries have fixed storage formats and especially when hybrid storage formats are needed, one cannot expect all different combinations to be pre-defined in the library.

Bik and Wijshoff described compiler techniques to automatically generate an implementation of a computation that operates on sparse matrix structures from an implementation of that computation expressed in terms of dense matrices that is supplied to the compiler [15]. User annotations about matrix statistics (e.g. its sparsity) or interactive user input is used to aid the compiler in selecting an efficient, pre-defined, sparse storage format for the matrices used in the computation.

Mateev et al. proposed a generic programming methodology to bridge the gap between algorithm implementation API and storage format API [68]. Algorithms are implemented as generic dense matrix programs, without considering a particular data storage format. The details of different, pre-defined, data structure formats are exposed using a low-level API. Their framework views sparse matrix formats are indexed-sequential access data structures and uses a restructuring technology based on relational algebra to convert a high-level algorithm into a data-centric implementation that exploits characteristics of the available sparse formats whenever possible. Matrices are considered as collections of tuples for the purpose of restructuring towards an existing sparse storage format. This restructuring process is covered in more depth in [55], which expresses the iterations, or tuples, that should executed as relational queries and finds a solution for these relational queries through the application of join reordering and determining suitable join algorithms to compute the joins.

Marker et al. described a method for the automatic parallelization and optimization of Dense Linear Algebra for distributed-memory computers called De-
sign by Transformation (DxT) [67]. Their method works by modeling algorithms in a data-flow graph. The graph contains nodes that represent redistribution operations or a LAPACK or BLAS function call. Optimization is carried out by applying graph transformations to find equivalent graphs that potentially exhibit better performance.

1.4 Contributions of Part I

The contributions of the first part of this thesis are developments towards a versatile intermediate representation for the optimization of codes and the development of a global, integrated, optimization framework for database applications. More specifically, the contributions of Part I of this thesis are:

1. An initial study of the cost of the overhead of the modular development methodologies for web applications that are in use today. This study shows that up to 90% of the instructions can be eliminated without affecting the final result, leading to substantial savings in energy consumption of web servers and a tremendous improvement of the performance of the web application. This work has been published in [84].

2. A tuple-based intermediate representation, the `forelem` loop, in which SQL queries can be naturally expressed in terms of loops governed by simple control. Extensions are described that allow nested and aggregate SQL queries to be represented.

3. The re-targeting of established compiler optimizations onto the `forelem` loops. The foundations of the intermediate representation and the re-targeted compiler optimizations have been published in [83].

4. Additional transformations for `forelem` loops and strategies for the optimization of database queries expressed in terms of `forelem` loops solely by using simple compiler optimizations. This optimization methodology results in executables codes that evaluate queries with a performance comparable to that of contemporary state-of-the-art database systems.

5. An automatic global integrated optimization process to reduce database applications to their essence. For two web applications it is shown that this automatic optimization process is very effective, on average eliminating 75% of the instructions and in specific cases up to 95% without affecting the execution and output of the application.

6. A trade-off analysis to support a decision process to determine whether it is beneficial to change the data access codes to be in-process (i.e. vertical integration) for a given sequence of queries. This analysis can be used by an automatic optimization process to determine for what parts of a web application it is beneficial to perform integral optimization, implying the construction of a local copy of the data. This work has been published in [85].
1.5 Contributions of Part II

The second part of this thesis contributes techniques to support automatic generation of code and data structures from a forelem representation of a problem in terms of tuples. More specifically, the contributions of Part II are:

1. Extensions to the forelem framework for the automatic generation of data storage formats from a representation of the code that operates on tuples. These extensions consist out of a materialization and concretization phase. In the case of sparse matrices, using these techniques established data storage formats, such as Jagged Diagonal Storage, can be automatically derived, that could up till now only be derived by hand. Part of this work has been published in [83].

2. A characterization of the search space consisting of many different variants of a sparse matrix computation represented in terms of tuples that can be automatically generated using the forelem framework. This characterization shows that by performing an exhaustive search through this search space, variants of the computation can be found that are in most cases faster than the implementations of these computations supplied by sparse algebra libraries, and at least on par in performance.

3. A further extension of the forelem framework, the ready clause, that allows dependencies between tuples to be naturally expressed. As such, generic irregular computations can be expressed in terms of tuples. By expressing dependencies as dependencies between tuples, it is trivial to deduce which operations on tuples can be executed at the same time. Preliminary experiments show that from an ordinary triangular solver code expressed in terms of a dense matrix, a highly parallel implementation is automatically derived that operates on sparse storage. This automatically derived implementation has a performance that is competitive to that of hand-optimized implementations. This work has been published in [86].

4. Initial work to support the expression of distributed forelem loops. By putting the optimization process in control of how forelem loops are distributed, optimal data decompositions and distributions can be determined. This extends the capability of the forelem framework to also optimize data composition and distribution in additional to the optimization data storage formats that are used locally. Part of this work has been published in [83].

1.6 Outline

This thesis is organized as follows. In Part I a framework is described for global, integrated, optimization of web applications. Chapter 2 presents an initial study of the cost of the overhead of the modular development methodologies for web applications that are in use today. It is shown that up to 90% of the instruction can be eliminated without affecting the final resulting, leading to substantial savings
in energy use of web servers and a tremendous improvement of the performance of the web application. This chapter has been published in [84].

Chapter 3 introduces the forelem loop, which is the main constituent of the forelem framework. A forelem loop specifies iteration of a subset of a multiset of tuples. SQL queries can be naturally expressed in terms of forelem loops. Transformations can be carried out on these loops to optimize the performance.

Chapter 4 extends the basic forelem framework introduced in Chapter 3 with syntax and transformations for handling nested SQL queries. In Chapter 5 further extensions are proposed for handling aggregate functions and group-by queries.

Chapter 6 describes how queries that are expressed in terms of forelem loops can be effectively optimized solely through the use of simple compiler optimizations.

Chapter 7 discusses how the global integration optimization process of database applications can be performed automatically for reducing database applications to their essence.

Chapter 8 describes a trade-off analysis that can be used by an automatic optimization process to determine for what parts of a web application it is beneficial to perform integral optimization, implying the construction of a local copy of the data, and for which parts this is not beneficial. This chapter has been published in [85].

Part II of this thesis explores the use of the techniques developed for the integral optimization of database applications in other application domains. Chapter 9 describes extensions to the forelem framework to support the automatic generation of data structures from a representation of a problem in terms of tuples. Parts of this chapter have been published in [83].

Chapter 10 characterizes the search space that consists of many different variants generated from an initial representation of a sparse matrix computation in forelem through the application of transformation. It is shown that by performing an exhaustive search through this search space, variants of the computation can be found that are in most cases faster than the implementations of these computations supplied by sparse algebra libraries.

Chapter 11 further extends the forelem framework with a ready clause. Using this clause, dependencies between tuples can be naturally expressed, resulting in a method that is especially suited for the automatic parallelization of irregular codes. This chapter has been published in [86].

Chapter 12 proposes a syntax for expressing and controlling distributed execution of forelem loops. It is described how this makes the forelem viable for the optimization of Big Data applications.

Finally, Chapter 13 summarizes the thesis and discusses future perspectives of the forelem framework.