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 7

Automatically Reducing Database Applications To Their Essence

7.1 Introduction

In this chapter, we propose an optimization technique specifically targeting the minimization of the number of instructions\(^1\). Essentially, the idea behind the technique is to locate and eliminate unnecessary instructions. These are instructions that can be omitted without affecting the course of execution and the output of an application. As a result of this elimination, an application is reduced to its essence. As can be concluded from Chapter 2, the number of instructions of the resulting executables can be significantly reduced when database applications are reduced to their essence. Note that this approach does not take cycles per instruction (CPI) into account at this point, but solely focuses on the reduction of the number of instructions. If the CPI remains around the same level, this implies a reduction of execution time by the same amount, directly improving the performance of the software. More likely, however, is that the CPI will slightly increase. This is attributed to the fact that the instructions performing memory traffic to and from the database tables, instructions characterized by a higher CPI, are not eliminated. Nevertheless, if the CPI would increase by 50\%, then still up to 92.5\% of the total number of cycles to be executed is eliminated. This is a drastic improvement and as a consequence the targeted hardware platform is more effectively exploited.

The large number of instructions that can be eliminated from database applications stems from that fact that these applications are typically developed with a modular approach. At the foundation a database management system (DBMS) is used and these systems have traditionally been developed as separate, independent software (server) applications. This independence makes database systems modular and enables their use in a variety of applications. Between database

\(^1\)Note that traditional compiler optimizations which target code compaction should not be confused with the target of this chapter. Code compaction could still be used on the resulting codes from our optimizations.
systems and database applications various framework layers are often used to facilitate development. This layered and modular approach allows for rapid prototyping, development and deployment.

However, this approach does come at a price. It is well known that the cost of the overhead induced by this modular and layered approach is significant. More importantly, the stacking of layers obscures the essence of the database application. The essence of the application can only be captured by breaking down these layers. This has as result that the number of instructions is drastically reduced, having a direct, very advantageous, effect on the application’s performance. Although the fact that overhead is created by these layered approaches might be obvious, the amount of overhead that is induced by these methods, up to 95% of the total amount of instructions, is rather surprising (see later on in this chapter).

Another consequence of this stacking is that compiler optimizations are mostly restricted to the application part of database applications while the optimization of the DBMS server is mostly delegated to the query optimizer. In fact, there is generally no integration of these two optimization efforts. If a query optimizer is not aware of how the data is used within the application, or if the application optimizer cannot influence optimization of the data access done by the query optimizer, a database application can never be optimized to its full potential automatically. Therefore, exposing the essence of a database application has as additional advantage that application optimization and query optimization can be targeted integrally, further increasing the performance of the application. In our approach, both the application and its queries are optimized using optimizing compiler techniques [104]. Optimizing compilers have been very effective in high performance computing as well as in general computing by optimizing loop structures, data structures, register allocation, data prefetching, etc.

The process of capturing the essence of the application consists out of automatically stripping the layers of which a database application is built up. These layers include a high-level development (scripting) language, frameworks that facilitate rapid development in that language, the DBMS API layer, and so on. By eliminating these layers, the essence of the application is parsed into a common (compiler) intermediate representation. In this intermediate representation, all database accesses are exposed as accesses to arrays of structures, governed by simple loop control structures. On the other hand, this approach allows current development methodologies for DBMS applications to remain in place. For example, current development environments and frameworks to develop Java-based database applications or PHP-based web applications have been and are serving programmers very well. The reduction process as proposed in this chapter will be part of the backend code development process, so that the DBMS application development methodology will not be directly affected by the reduction process. Rather, an application is developed and tested as usual, but before extensive deployment the code is passed through the code optimization backend to eliminate as much overhead as possible. This way, we continue to take advantage of the available software development tools which enhance programmer productivity and combine this with a code optimization backend that significantly reduces the number of instructions to be executed, thereby also improving the performance of the application.
The effectiveness of the proposed reduction scheme is validated using two web applications: RUBBoS [74] and RUBiS [75]. Both web application benchmarks have been developed by a collaboration between Rice University and INRIA. We show that on average 75% of the instructions can be eliminated, and in specific cases up to 95%, without affecting the execution and output of the application.

The remainder of this chapter is organized as follows. In Section 7.2 the results of the initial study presented in Chapter 2, in which the instructions executed by RUBBoS and RUBiS benchmarks were manually reduced, are briefly reiterated. Section 7.3 describes the methods underlying the automatic instruction reduction process. Section 7.4 discusses how these methods are implemented and can be deployed within an operational workflow. In Section 7.5, we validate the effectiveness of our approach by presenting the results generated with our prototype compiler. Section 7.6 describes further optimizations that are possible on top of the results reported in this chapter. Section 7.8 presents our conclusions.

7.2 Attainable Results

Chapter 2 presented an initial study into the reduction of unnecessary instructions in database applications, among which the RUBBoS and RUBiS benchmarks. As a result, we have shown the potential reductions of instructions that are possible by hand optimizing these applications. For the purpose of identifying which instructions were eliminated, we categorized the non-essential instructions as “PHP overhead”, “MySQL overhead” and “SQL API overhead”.

Figure 7.1, presents the quantification of the different categories for the RUBBoS and RUBiS benchmark in terms of the number of instructions. As can be seen from this figure, the reduction ranges from 70.2% to 93.3%. From this figure follows that up to 93.3% of instructions can be eliminated, without affecting the results of the program. A direct result of this drastic reduction of the number of instructions is a significant improvement in execution time and energy consumption. For more details, the reader is referred to Chapter 2.

7.3 Vertical Integration

The results described in the previous section serve as a target for an automatic optimization method. This automation is achieved by breaking down the layers from which database applications are commonly built up. We refer to this process as “vertical integration”. This section describes how the different layers are broken down automatically.

7.3.1 PHP layer

PHP scripts are typically executed by parsing this script, followed by code generation and execution, resulting in a start-up overhead. Furthermore, PHP code cannot be integrated with Apache and the various PHP database modules, which are written in C. As a first stage in vertical integration, the PHP code is translated
Figure 7.1: Quantification of the amount of executed non-essential instructions, in $10^5$ of instructions, to generate a single page for different components from the RUBBo$S$ and RUBi$S$ benchmarks.
to C++ code. This translation thus serves two purposes: parsing and code generation overhead is eliminated and further vertical integration with the database modules is enabled.

To accomplish this code translation, our toolchain contains a source-to-source translator, which translates PHP source code to C++ source code. The source-to-source compiler was extracted from the HipHop for PHP project, that has been developed by Facebook [42]. The result of the automatic invocation of the HipHop source-to-source translator is a generated C++ code, which is subsequently compiled and linked against the HipHop runtime. This runtime contains implementations of the PHP built-in functions and data types that are used by the generated C++ code. It also features a built-in web server, which replaces the typical use of the Apache HTTP server. The web server is vertically integrated into the executable, further reducing the code size.

### 7.3.2 DBMS layer

The result of the previous step is a C++ source code, that performs calls to a DBMS API to execute SQL queries. This can in fact be compared to an Embedded SQL code. In this step, a vertical integration is performed of the database application and the DBMS that performs data accesses based on the submitted queries. As part of our prototype compiler, we have developed a source-to-source translator that scans the C++ code for calls to a given DBMS API, and replaces the use of this API with C++ code. At this moment, only the MySQL API is supported, but extensions to support other DBMS APIs are straightforward.

This replacement implies that calls to the DBMS API that submit a SQL query for execution are also replaced with code performing the identical operation. To achieve this the forelem intermediate representation is used, in which SQL queries can be expressed as a series of simple loop structures. This intermediate representation has been designed such that it integrates well in the workflow of traditional optimizing compilers. After parsing a SQL query into an Abstract Syntax Tree (AST), a forelem loop is generated immediately from this AST. No query plan is generated to support this translation. Subsequently, queries are optimized with loop transformations, rather than sophisticated query planning. Another advantage of the design of this intermediate representation is that it allows for straightforward integral optimization of the application code and its queries. Further optimizations that are possible within this framework, but not yet accomplished automatically, are described in Section 7.6.

Our source-to-source translator will detect DBMS API calls in the C++ code. Important calls to detect are for example: opening a connection to the DBMS, sending a query to the DBMS, retrieving the result set, accessing the result set and releasing the result set. The translator will annotate the semantics of these operations in the C++ AST. Commonly, to be able to submit a query to a DBMS, a parameter must be passed that specifies the DBMS connection to use. The values of these parameters are only known at runtime. Therefore, the code translator performs an advanced static analysis on the database application code, in order to deduce which connection is used in which DBMS API call. Similarly, such an analysis is used to find exact query strings that are passed to the DBMS calls.
Query strings that are constructed at runtime in the database applications, are composed out of multiple calls to string manipulation routines.

As soon as all necessary data has been collected, the DBMS API calls can be translated. A prerequisite for a successful translation of a call to execute a query in the DBMS is that the connection is known as well as the query string. The connection information is used to obtain table metadata and the data of the tables at a later stage. Using libforelem, a library we developed that can manipulate forelem ASTs, the SQL query is parsed. A check is done whether the query conforms to the database schemas. For example, the following SQL query taken from RUBiS:

```
SELECT item_id, bids.max_bid
FROM bids, items
WHERE bids.user_id=$userId
AND bids.item_id=items.id
GROUP BY item_id
```

is written in forelem as:

```
forelem (i; i ∈ pBids.user_id[$userId])
  forelem (j; j ∈ pItems.id[Bids[i].item_id])
  \[I = I \cup (Bids[i].item_id, Bids[i].max_bid)\]
forelem (i; i ∈ pI)
  \[I = I \cup (I[i].item_id)\]
forelem (i; i ∈ pI.distinct(item_id))
  \{
    forelem (j; j ∈ single(pI.item_id[I[i].item_id]))
    \[R = R \cup (I[j].item_id, I[j].max_bid)\]
  \}
```

In this loop nest, the notation pBids.user_id[$userId] denotes an index set. This index set only contains subscripts i into table Bids for which the user_id value of the tuple equals the value in the variable $userId. The forelem library will determine how to generate code for this index set. If an index is defined on the table in the original SQL database, an explicit index will be generated in the data generation phase, that is kept updated when writes are done to this table. In case an explicit index is not defined, libforelem may choose to insert code before the loop to generate the necessary index set for this loop at runtime.

The forelem loop does not have a particular iteration order. The order in which subscripts are stored in the index set, or the order in which these are iterated, is not defined. Because of this, we are not limited in the range of transformations and iteration schemas we can apply. In fact, index sets are the essence of forelem loop nests as they encapsulate iteration and simplify the query loop code so that aggressive compiler optimizations can be successfully applied.

A large variety of SQL queries can currently be expressed in terms of the forelem intermediate representation. A special syntax is available for expressing the use of aggregate functions. For the distinct keyword, a distinct tag exists for the index sets. So, application of duplicate elimination will not complicate the expression of the loop in forelem. This way, loop transformations can still be applied to the loop effectively. Joins are simply represented by nested forelem loops.
Group by queries can be expressed in *forelem* using aggregates, the *distinct* tag and by using multiple *forelem* loops. This results in an initial expression of the query in *forelem*, like the example shown above, which is subsequently subjected to transformations at the *forelem* level.

Observe that the structure of the loop allows for straightforward integration with imperative application codes. Our source-to-source translation extends the C++ AST with the *forelem* AST, so that code transformations can be defined that target both the C++ as well as the *forelem* code. After these transformations have been performed, C++ source code is generated for the *forelem* code when the C++ source file is rewritten. The C++ code that is generated from the *forelem* loop accesses the tables through a simple array of structures. Note, that such transformations are not possible if an existing DBMS were simply integrated into the same process as the application program. This does not result in the DBMS API being broken down. To be able to reduce the maximum amount of instructions, interpretation of the DBMS API calls and the executed queries is a necessity.

The result of this query inlining is typically a loop that generates a result set, see the $T$, $G$ and $R$ sets in the example above. Any suitable data structure can be used to store this result set. The data structure used can be adapted to the characteristics of the query, used tables and the application itself. For example, if only a few results are expected, a more efficient data structure can be used to store these results, contrary to the use of more advanced data structures for storing large result sets. DBMS API calls that retrieve and access the result set are translated into C++ codes that operate on the result set generated by the inlined query code. For a more detailed discussion the reader is referred to Chapter 2.

After all uses of DBMS API have been translated by the source-to-source translator, the translator will determine which database tables and indices are used by the translated queries. Because the operations performed by the DBMS are being integrated into the application program, the accessed data must be migrated as well. The used tables and indices are fetched from the DBMS and stored into local binary files as arrays of structures, that are accessed by the application program using memory-mapped I/O.

### 7.3.3 DBMS API layer

In the initial generated code, the influence of the DBMS API can still be found. For example, at the original location of a DBMS query call, a loop evaluating the query and generating a result set can now be found. A bit further on in the generated source file, a loop will be found that accesses this result set. These two loops can be merged, eliminating the need to explicitly create a result set. Code transformations like these become possible now that the DBMS API layer has been removed and the application code and queries are expressed in a common intermediate. Currently, our prototype optimizer does not perform such transformations yet.

### 7.4 Incorporation in an Operational Workflow

Current development methodologies to develop database applications are serving programmers very well. A change in methodology to have the developers focus
on instruction reduction manually is not cost effective. In general, hand optimization of code is an elaborate and expensive task and may not weigh up to the costs that are potentially saved. A clear advantage of our approach is that no modification to current development methodologies is needed and the translation to a significantly more compact code is performed fully automatically. This automatic translation process can be easily integrated into an operational workflow as part of the deployment phase where a new version of the application is to be deployed on the production servers.

Many deployments of web applications are distributed, because a single server can typically not handle the load. However, once a database application has been vertically integrated, it operates on a data store that is local and private to that application. Update actions, i.e. insert, delete and update statements, that are performed by the application are applied on this local data store. For these updates to become visible to other clients of the same database, this data must be distributed to these clients. So, where database systems store data at a centralized location and provide access to this data through an API, vertically integrated applications store data locally and need a method to distribute updates.

A solution is needed to deal with the absence of a (remote) central data store. A straightforward solution is to introduce a central data store in addition to the local data stores and submit all updates to this central data store. The central data store must then ensure all local data stores are kept synchronized. Essentially, this means that write actions that are performed must update both a local and remote data store, potentially hampering performance. This problem is similar to the problem of synchronizing local database caches, of which several schemes have been described in the literature [88, 64, 7, 76, 78]. In Chapter 8, this problem will be discussed in more detail and a trade-off analysis is described to support a decision support process to determine whether it is worthwhile to vertically integrate a code that executing a certain query mix.

### 7.5 Validation

To validate the effectiveness of our approach, we have performed experiments with the RUBBoS [74] and RUBiS [75] benchmarks. For each benchmark, instruction count measurements have been conducted on three versions of the code. The first instruction count is the original version of the code written in PHP and executed using Apache and MySQL. The second count is based on a vertically integrated code, that has been transformed by our vertical integration compiler using the procedure described in Section 7.3. The third count is the result of optimizing the code by hand, also reported in Section 7.2.

All experiments have been carried out on an Intel Core 2 Quad CPU (Q9450) clocked at 2.66 GHz with 4 GB of RAM. The software installation consists out of Ubuntu 10.04.3 LTS (64-bit), which comes with MySQL 5.1.41 and Apache 2.2.14. The instruction count measurements have been performed using the oprofile software, sampling the INST RETIRED hardware performance counter present on the Intel Core 2 CPU.
The different PHP scripts, or components, that make up each benchmark have been benchmarked separately. For the instruction count measurement, each component has been executed multiple times, if applicable with different CGI arguments. This is to straighten out fluctuations, incorporate the effect of different CGI parameters and to collect enough samples for the profiler to produce meaningful results. Execution of a component is triggered by requesting the respective page with an HTTP client (e.g. `elinks -source`). The final result is the average number of instructions executed for a single execution of the component. `oprofile` reports instruction count samples per process and only the instructions executed of these processes that play a role in the generation of the page have been aggregated. So, other processes running on the system did not influence the measurements. The HTTP client has been excluded from the aggregate instruction counts.

### 7.5.1 Read-only Operations

Currently, our vertical integration compiler is able to transform the code of 7 PHP scripts in RUBBoS and of 9 PHP scripts in RUBiS. The realized instruction count reductions are shown in Figure 7.2. The bars on the left of the dashed line are the results for the RUBBoS benchmark components, the results on the right for RUBiS. PHP scripts that have been executed with different input parameters are marked with "*" and the average was taken of the results. The white bar indicates the percentage of instructions that has been eliminated, the gray the percentage of instructions that remain.

The results show that our current vertical integration compiler is able to realize a significant (43.0% to 87.0%) reduction in instructions for the different components of RUBBoS and up to 95.3% for the RUBiS components.

Three of the RUBBoS components and 4 of the RUBiS components have also been optimized by hand. The results of the hand-optimized codes are shown in Table 7.1. Note that even though our compiler is a work-in-progress prototype, it is already able to produce codes that achieve a performance close to hand-optimized codes for these benchmarks (differences in the range of only 0.4% to 5.3%). Continued development of the prototype will decrease the gap between automatically and hand optimized codes. Secondly, we notice that RUBBoS and RUBiS are quite simple codes, where the optimization techniques that become possible after incorporation of the query codes cannot be utilized to their fullest potential. For example, in Chapter 2 we showed that in other applications the integration opportunities were much more versatile, in particular in the phase after incorporation of the query codes, in the order of another 10%. Therefore, we believe that the results reported in this section form a lower-bound for results that can be achieved for more complicated applications that we plan to survey in the future.

### 7.5.2 Read/Write Operations

In this subsection, we present the reduction in instructions that can be obtained for read/write components. Results for automatic and hand optimized codes for 4 read/write components of the RUBBoS benchmark are shown in Figure 7.3. We
Figure 7.2: Percentage in instruction count reduction over the original Apache/MySQL execution for 7 of the RUBBoS benchmark components on the left and 9 of the RUBiS benchmark components on the right. Results realized by an automatic transformation of the code are shown. A "*" indicates an average is shown of performing experiments with different arbitrarily chosen input parameters.
Table 7.1: Percentage in instruction reduction over the original Apache/MySQL execution for only these RUBBoS and RUBiS benchmark components that have been optimized by hand.

<table>
<thead>
<tr>
<th>Component</th>
<th>Automatic</th>
<th>Hand</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUBBoS StoriesOfTheDay</td>
<td>70.8%</td>
<td>76.1%</td>
</tr>
<tr>
<td>RUBBoS ViewComment</td>
<td>88.0%</td>
<td>93.3%</td>
</tr>
<tr>
<td>RUBBoS ViewStory</td>
<td>87.0%</td>
<td>92.7%</td>
</tr>
<tr>
<td>RUBiS SearchByCategory</td>
<td>65.6%</td>
<td>70.2%</td>
</tr>
<tr>
<td>RUBiS ViewBidHistory</td>
<td>72.7%</td>
<td>73.5%</td>
</tr>
<tr>
<td>RUBiS ViewItem</td>
<td>77.4%</td>
<td>77.8%</td>
</tr>
<tr>
<td>RUBiS ViewUserInfo</td>
<td>71.9%</td>
<td>74.9%</td>
</tr>
</tbody>
</table>

have selected these components from the RUBBoS benchmark, because they exhibit more interesting query mixes compared to the components found in RUBiS. The translation performed for the write operations is obtained by performing only an update of the local data instead of updating data in a remote DBMS. The results indicate that a reduction in the number of instructions executed is possible of around 70%.

Although notable speedups can be achieved with vertical integration of Read/Write components, frequent updates to local data stores is not a scalable solution for distributed deployments of the application. Therefore, we must consider the performance effect of distributing the updated data to the other nodes in the system. A number of methods to solve this problem will be described in Chapter 8. In general, for codes with many write actions, the cost of distributing the updates may not weigh up to the benefits attained by vertical integration. To quantify this, in Chapter 8 a trade-off analysis is presented to support automatic decision making whether or not a code with a certain query mix should be vertically integrated. In this chapter, we assume a setup where updates are immediately applied on the local data and are synchronously submitted to a main DBMS. Other setups are of course possible and a similar trade-off analysis can be carried out in these cases.

Using this methodology, we make a prediction of the speedup that can be achieved in query processing when a transition is made to query evaluation local to the application and additionally sending updated data to a central DBMS. In order to do so, the speedup is correlated to different ratios of remote writes against local writes. That is, we can control the amount of write actions that are also applied at a central location. For writes that are only applied locally, this has as a possible consequence that components that require this data can only be executed on one particular host. This way, a distributed deployment of a web application can be tuned in different ways.

The speedup predictions as described above can be displayed graphically in a contour plot, see Figure 7.4. From the figure, the predicted speedup can be read for a given component and remote/local write ratio. Note that all speedups reported in the figure are above 1.0. So, even if all write actions are applied remotely as well, it is predicted that an overall speedup can be achieved. This is due to the fact that every component performs at least a single read action. In case of Store-
Figure 7.3: Percentage in instruction reduction over the original Apache/MySQL execution for 4 R/W components of the RUBBoS benchmark. Results realized by an automatic transformation of the code and hand-optimized code are shown.

Figure 7.4: Contour plot of the predicted speedups (contour lines) for different read/write benchmark components and remote/local write ratios.
Further Optimizations

In the previous section, we have described that up to 95% of the instructions executed by database applications can be eliminated. As has been discussed in the Introduction of this chapter, a reduction of the number of executed instructions has a direct impact on the performance of the application.

Figure 7.5 shows the percentage reduction of the page generation time of the surveyed benchmarks realized by the automatic code transformation performed by our compiler. The page generation times were measured by storing time stamps at the start and end of the various PHP scripts and computing the time elapsed. Note that the percentage shown only reflects the reduction in page generation time. This excludes any speedup in the stages before execution of the PHP script has started, such as parsing the PHP script and generating executable code that is done by the PHP module in the Apache web server. Due to the nature of our measurements using oprofile, these were included in the instruction count measurements reported in Section 7.5. Due to the use of HipHop for PHP, the PHP parsing and code generation is no longer done at runtime, so the total reduction of execution time is larger.

The results show that for the majority of the benchmark components the page generation time is reduced by over 80%. For a number of pages, the page generation time is reduced by around 95%.

These achieved speedups are a direct result of the instruction count reduction. To accomplish this reduction in instructions, the layers used to compose the database application had to be broken down. For example, the use of a modular interface to the DBMS has been removed and replaced with query evaluation within the same process. This has reduced the number of executions to be executed, as data no longer has to be boxed for transfer to and from a DBMS. Important is that the elimination of the use of a DBMS has as a significant side effect that time is no longer lost by transferring data between two processes on the system and context switching overhead. So, even though the expected total reduction of execution time is larger than reported, the reported reductions in page generation time are still larger than the reported reduction in instruction count.

Note that all the results reported in this chapter up till now, have been obtained without any advanced compiler optimizations, like loop transformations and code optimizations, see the previous chapters. Due to the use of the forelem intermediate representation, both the application logic as well as the database queries are represented as loops. On these loop structures, a compiler can perform traditional, sophisticated loop optimization, resulting in a further substantial improvement in performance. Additionally, other effective optimizing compiler
Figure 7.5: Percentage in page generation time reduction over the original Apache/MySQL execution for 7 of the RUBBoS benchmark components on the left and 9 of the RUBiS benchmark components on the right. Results realized by an automatic transformation of the code are shown. A “*” indicates an average is shown of performing experiments with different arbitrarily chosen input parameters.
techniques can be applied to further optimize the code, such as optimizations for more efficient data structures, register allocation, cache usage and data prefetching. For an example of the transformations that can be applied after the DBMS API has been replaced with *forelem* loops, see Chapter 3 (Section 3.4).

We have demonstrated in this chapter that an automatic reduction of a database application to its essence is feasible. This uncovers a significant number of unnecessary instructions, which are automatically eliminated, having a direct beneficial impact on the application’s performance. The fact that the application program logic and its queries are expressed in a common intermediate representation, forms a basis for the definition of code transformation specifically for database applications.

### 7.7 Related Work

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 our approach 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. Furthermore, we accomplish a significant reduction in the number of instructions to be executed.

Holistic transformations for web applications are proposed in [66, 37], where transformations are performed on both the application code and the database queries performed by the application code. The division between application and database codes remains in place however. A similar approach for database applications written in Java is discussed in [21].

### 7.8 Conclusions

In this chapter, we have presented the results of our prototype compiler that is able to automatically reduce the number of instructions executed by database applications by up to 95%. These results have a direct effect on the performance of the application, with page generation times also being reduced to up to 5% of the original page generation times.

Next to these significant results, we have demonstrated that it is feasible to automatically translate applications to a substantially reduced form. This is accomplished through the vertical integration methodology. Our current prototype compiler is capable of achieving a performance near that of the hand-optimized codes. In future work, we will further improve our compiler technology to match and go even beyond the performance of the hand-optimized codes. Furthermore, we are working on developing novel compiler optimizations that will further improve the performance of queries translated using the *forelem* framework.
The current prototype of the reduction process is capable of processing PHP codes by a translation to C++ code using the HipHop for PHP project. Many other programming languages and frameworks are also used to develop database applications. The general nature of the forelem framework does not restrict its usage to the C and C++ programming languages. As such, our prototype can be extended to be able to process applications written in other programming languages.