INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

OutSystems Compiler Optimizations

Leonardo Monteiro Fernandes

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Júri

Presidente:	Pedro Diniz
Orientador:	Pedro Reis dos Santos
Co-orientador:	Lúcio Ferrão
Vogais:	Arlindo Oliveira

Novembro de 2007

OutSystems Compiler Optimizations

Leonardo Monteiro Fernandes

Dedicatory

I wish to thank everyone that supported me throughout this work, as certainly I couldn't do it all by myself. I cannot begin without thanking my parents and my whole family, for giving me all the support that I've ever needed.
Special thanks also to Lúcio Ferrão for being deeply involved with the project. Also many thanks to my teacher Pedro Reis dos Santos for its support in the literature research, and suggestions to this thesis.
Thanks to Rui Eugénio, who helped me a lot with the technical details of the OutSystems compiler.
And finally I wish to thank António Melo and Carlos Alves for their understanding of my needs when I needed time to dedicate myself.

Leonardo Fernandes

Abstract

Ideally, compilers should produce target code that is as good as can be written by hand. Unfortunately, this goal cannot be achieved in the usual case, and it's up to the optimizer to do its best job in approximating this ideal situation.

Traditional optimizing compilers have advanced to the point where the do an excellent job of optimizing a single procedure. Accordingly, optimization research has begun to focus on inter-procedural analysis and optimization. And for OutSystems, the lack of inter-procedural optimizations represents a weakness in supporting large-scale applications, with complex inter-procedural relationships and modularity requirements.

In this paper, we present a case study of optimizations in the OutSystems Platform compiler. It is presented the theoretic background behind optimization techniques, and it is given focus on designing an inter-procedural optimizer for the current OutSystems compiler.

Keywords: Compiler, Optimization, Data-flow analysis, OutSystems, Live variable analysis, Inter-procedural optimization.

Resumo

Idealmente, os compiladores deveriam produzir código gerado que fosse tão bom quanto é possível ser escrito à mão. Infelizmente, esta meta não pode ser atingida no caso geral, e é tarefa do optimizador fazer o seu melhor para aproximar-se da solução óptima.
Os optimizadores tradicionais já avançaram ao ponto de serem satisfatórias as optimizações de um único procedimento. Desta forma, a investigação na area de optimização começa a focar-se na análise e optimização inter-procedimentais. E, para a OutSystems, a falta de um optimizador inter-procedimental representa um ponto fraco ao suportar aplicações de larga escala, com interações inter-procedimentais complexas, e com requisitos de modularidade.
Neste trabalho, apresentámos um caso de estudo de optimizações no âmbito do compilador da plataforma OutSystems. Apresentam-se os fundamentos teóricos por detrás de técnicas de optimização, and é dado foco no desenho de um optimizador inter-procedimental para o compilador OutSystems actual.

Keywords: Compiladores, Optimização, Análise de fluxo de dados, OutSystems, Live variable analysis, Optimizações inter-procedimentais.

Index

Index of Figures

Acronyms Index

API – Application Programming Interface

IDE – Integrated Development Environment

IIS - Internet Integration Services

IT – Information Technology

JIT – Just In Time

OML – OutSystems Markup Language

OSHEComp – OutSystems Hub Edition Compiler

SMS – Short Message Service

SQL – Structured Query Language

URL – Universal Resource Location

WAP – Wireless Application Protocol

WYSIWYG – What You See Is What You Get

1 Introduction

As the complexity of software grows, developers need more powerful tools to help them build applications. Such tools generally provide a set of high level primitives for application building, in order to minimize the effort of the developer.

A good tool for creating applications would provide primitives as simple as possible for the developer to use. We can take as example the SQL language which enables to manipulate data through the `SELECT`, `UPDATE` and `DELETE` statements. This trend in simplifying primitives also applies for any development tool, such as frameworks and components, or even IDE's. But those simplifications, necessary to increase development efficiency of modern applications, can often decrease the runtime performance of the same applications, since they are usually not expressive enough to be used optimally in every situation. There are alternatives to balance the oversimplification:

- Provide alternatives for the high level primitives. For example, in network applications the developer usually can choose the adequate level of abstraction to use, selecting a desired protocol and using its primitives to design the application. Different protocols are available, and each of them has its own advantages and limitations.
- Provide primitives that are able to be automatically optimized. For example, the already cited querying language SQL provides primitives that are aimed to be transparently optimized by the database engine. Modern programming languages provide primitives for memory allocation that are able to be optimized both in compile time, and in runtime through garbage collection technologies.

The first alternative requires support for both high level and low level primitives, which requires additional costs in designing and maintaining the primitives. It also adds a burden to the developer, which needs to be aware of additional primitives, and needs to decide which set of primitives to use in a particular application.

The second alternative is only possible if the primitives were designed with care, to allow their automatic optimization. It uses a technology called **optimizing compilers**, which aims to reduce automatically the runtime inefficiencies, even if the developer is not aware of this process.

In this thesis we explore the optimizing compiler of the OutSystems platform, in which the main concern is the optimization of the performance of web applications: mainly optimizing the database access times, and optimizing the size of the data transferred between the browser and the application server.

1.1 Motivation

As we have seen, the role of the optimizing compiler is to reduce runtime inefficiencies that arise in the compiled applications. These inefficiencies can be caused either by bad programming practices, or can be inherent to the design of the programming primitives.

At first sight, it can seem contradictory to have a programming language, by design, impact negatively the runtime performance. It could also seem pointless to develop a new optimization technology, just

to make up for the deficiencies in the programming language. But there are many benefits that arise from this point of view. These are:

- **Dissociation between application logic and performance concerns**. If an optimization process is added to the compilation of the application, the development can focus on the application logic, and leave the performance concerns be addressed by the automatic optimizer. This approach requires less development resources, produces clearer code, and with a lower maintenance cost.
- **Benefit from optimizer evolution**. As the optimizer process is improved, because of advances in the optimization research, all applications would benefit of it, without extra development effort.
- **Simplification of the programming language**. Programming languages are used to develop applications which will run in computers with limited resources. When designing a programming language, the limitations in the current computers should be kept in mind, and the primitives should be flexible to be used without performance losses in a variety of scenarios. But if an optimizer compiler is provided for the programming language, the language primitives can be simplified, hiding from the developer the optimizations that are automatically handled. Ideally, the language could be designed for its logic behavior, ignoring completely the limitations of the runtime environment.

It should be clear that completely freeing the developer from the performance concerns is not the aim of the optimizer compiler. For instance, the choice of the most suitable data structure or algorithm will always be an issue handled by the developer of the particular application. But having automatic optimization techniques allows raising the abstraction level of a language, and helps reducing the development effort.

Section 2.2 further motivates the reader, by analyzing the particular case of the OutSystems compiler, and the particular inter-procedural optimizations.

1.2 Objectives

The scope of this work is to extend the existing OutSystems optimizing compiler, implementing inter-procedural optimization techniques.

An algorithm is designed to solve the inter-procedural optimizations relevant to OutSystems. In this paper, we will study the properties of the algorithm, implement it, and investigate its static effects when compiling real-world applications. Although it would be much more relevant to know the runtime effects of the inter-procedural optimizer – such as gains in runtime application memory usage, or increased performance – it would require us to analyze not only the application itself, but the users and the processes which interact with the application. Thus we follow a more pragmatic measure of the gains of an application, by analyzing it statically.

We also point directions that could be followed as a continuation of this work.

1.3 Structure of the Document

The paper begins with chapter 1, as an introduction to the problem being solved, giving emphasis on a motivation for approaching the problem.

The OutSystems enterprise is introduced in chapter 2, which places the reader in the context of the OutSystems business, and unveils the product developed by the company.

A brief introduction to the state of the art in optimization techniques lies in chapter 4, which presents the theoretic foundations to the following chapters.

In chapter 5, it is formally defined the problem of the inter-procedural optimizations in the OutSystems compiler. Based in the foundations built on chapter 4, we present a solution that allows inter-procedural optimizations in OutSystems compiler. Finally, the measured results of the implementation are shown.

Chapter 6 proposes future work that should be done to improve the current implementation, together with ideas that could complement it.

At last, the conclusions of the paper are stated in chapter 7.

2 OutSystems Enterprise

OutSystems is an international software company created in March 2001. OutSystems main product is a platform for developing and deploying web and mobile built-to-change applications. While the product supports mobile applications in WAP and SMS, the main business focus of the enterprise is on web applications.

The enterprise focuses on product excellence and total customer satisfaction. These values are reflected in high-quality teams and processes and a lean international organization supported by a state-of-the-art informational system [1].

2.1 The OutSystems Business

The main customers are IT companies that need to develop web applications that have a great amount of evolutionary components, requiring their applications to be easy to maintain and to change.

The goal of OutSystems is to reduce the code written by the developers, allowing them to focus on application logical flow rather than in syntax and integration between several interfaces. The product accomplishes this task by providing visual primitives that can be combined to build the applicational flow, and a WYSIWYG editor for layout of the presentational screens of the application.

While the visual components that make up the logic flow can be easily recognized by a human, rather than lines of code, they are also tailored for the recurrent patterns in the web applications development. There are available primitives such as querying a database, invoking a web service, or managing a user and its permissions in the application. The screen editor also has the same presentational patterns that arise naturally in the development of a web application, such as listing records of a database, or editing a given record. Some scalability issues are also tackled by the platform, such as independence of the application from the database driver and schema, and independence of the URL structure of the web site.

With those patterns ready to be used by the programmer, the OutSystems product is a platform for quickly building web and mobile applications from scratch, and easily maintaining them afterwards.

The integration with existing applications, which might be implemented in a different technology, is also an issue addressed in OutSystems Platform. It can be accomplished by extending the visual primitives, by either using web services, or by writing an extension in a regular programming language.

2.2 Motivation: Business Requirements for Optimizations

As previously stated, OutSystems Platform aims to give the developer a set of predefined tools specially tailored for the life cycle of web and mobile applications. It also focuses in the maintainability of the applications developed under the platform, so the developer is kept away from implementation details as much as possible.

In fact, simple web applications can be made entirely in the OutSystems Platform through visual flows, without writing code in any traditional language. This provided independency increases the portability of the applications, because the developer would be only compromising the application with the OutSystems Platform itself. The application can be deployed in any environment supported by the platform.

One of the side effects of raising the abstraction level of the development tools is that the developer has less power on choosing exactly what is under the hood. The less configurable a language is, the more it relies on the compiler to generate quality code.

Fortunately, another side effect of working with higher level languages is that usually they offer more opportunities for automatic optimizations, and these optimizations generally have a great impact in the developed applications. So a constant research in optimization techniques is something that follows every software platform.

Having a good optimizer adds a great value to the product. If an optimizer compiler addresses a given performance concern, the developer can rely in the quality of the code generated, and is free from thinking about that same concern. The automatic optimization translates not only in the increased performance of the application, but also allows a quick and more focused development cycle.

The OutSystems compiler currently has an optimization module, which handles the relevant optimizations local to a given procedure, but does a conservative analysis of calls between procedures. It also causes some local optimizations to be lost, because of pessimistic assumptions about called procedures.

Because of this lack of inter-procedural optimizations, the developers using the OutSystems language prefer a small number of big monolithical procedures, instead of many simple procedures that can be reused in the application logic. The current development practice also disallows the encapsulation of logic inside a procedure, which is needed to define the abstractions of the application, or *objects*.

2.3 OutSystems Platform Overview

The OutSystems Platform is the company main software product. It is composed of a set of three independent components, integrated to deliver an agile development environment for professional web applications. OutSystems Platform was formerly called OutSystems Hub Edition.

The most visible of the components is *Service Studio*, which is used by the designer and developer of the web application. It makes up the front-end of the platform for the developer.

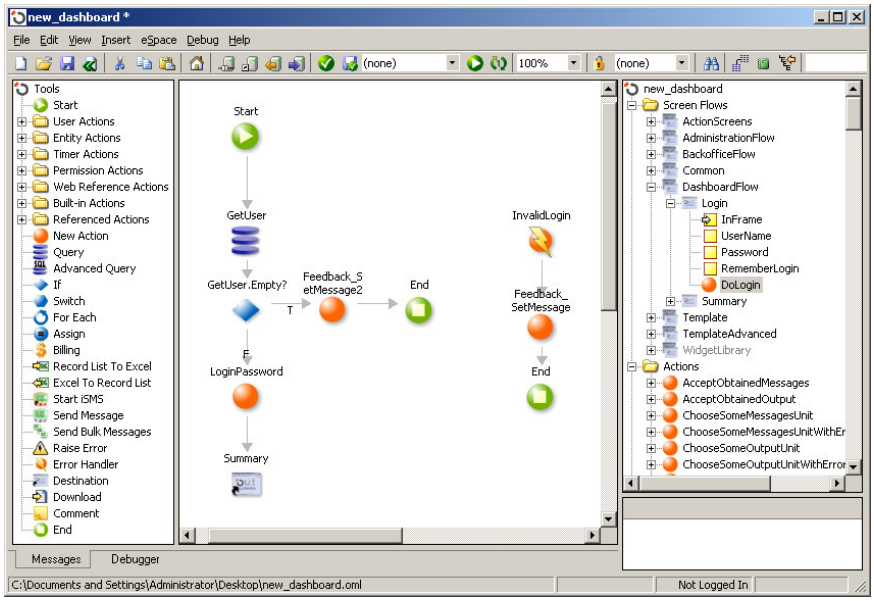


Figure 1 – Service Studio showing a web application.
We can see in the left pane the visual primitives that can be used in the application flow. In the center pane, we can see the logic behind the DoLogin action, with primitives such as a query to the database (labeled as GetUser), a conditional branch (labeled as GetUser.Empty?), and exception handlers (labeled as InvalidLogin). In the right pane, we have the designed high-level structure of the application.

The developer can build its application in Service Studio, and save it. The outcome is a single file containing everything needed to successfully deploy the application in a production server, so the application can be easily carried between computers, or shared with another developer. The file that is saved in the OutSystems Service Studio is called an *eSpace*, or more informally *OML*.

A single eSpace can be developed cooperatively by several teammates at the same time. Each teammate can have its own independent personal area for developing the application in the server. When the development reaches a milestone, the result of their individual effort can be joined in the final eSpace by merging the files they worked on.

The final eSpace file can be verified against errors, in a similar way that the compiler checks for syntactic errors. Some common consistency problems between the several layers of the web application are also checked, and warnings are issued if problems are found. If the verification succeeds, it can be published in the server for production, if so desired.

The OutSystems server for deploying applications is called *Hub Server*, and is another component of the platform. It is actually the core of the platform, and control the compilation, deployment and execution of the applications. In a gross comparison, we can relate Service Studio with an IDE, and Hub Server with the compiler, linker and operating system altogether.

The Hub Server is composed of:

- a traditional web application server for hosting the web applications;
- a database to store the persistent data for the applications;
- and a couple of OutSystems services which are invoked to compile and deploy the applications.

Currently, there are actually two kinds of Hub Servers, supporting distinct environments. The *.NET Hub Server* supports compiling an application into .NET code, and deploying it in Microsoft IIS web

server[1]. The second version is a *J2EE Hub Server*, which supports compiling an application into Java code, and deploying in JBoss web server.

One Hub Server can map to one or more hardware machines, called *Hub Nodes*, allowing the production servers to rely on hardware redundancy for high availability. The database can also be kept in a dedicated server, which is a common practice in production environments.

Publishing an eSpace to a Hub Server involves compiling the OML description of the application into the source code of the application, be it .NET code or Java code. The source code can then be deployed in the web application server, for each active Hub Node. The source code can also be downloaded at any time, for auditing or porting it to another technology.

A basic versioning mechanism is used in the Hub Server, where each upload of an eSpace increases its version number. Reverting to a previous version is allowed in the platform at any time.

The Hub Server component comes with a web application called *Service Center*, which can be used to manage the eSpaces, and monitor the runtime properties of the platform. Service Center is the portal for any developer, or project administrator.

Finally, the third component of OutSystems Platform is *Integration Studio*, which is a Windows application that can be used to create extensions that can be used inside any eSpace. Extensions can be as complex as desired, and the programmer has the power of a traditional programming language at hands. The extensions can be used to achieve complex tasks, or to create custom connectors to integrate the OutSystems web applications with other applications, possibly implemented in other technologies. The extensions can be published to a Hub Server, and then further used inside an eSpace.

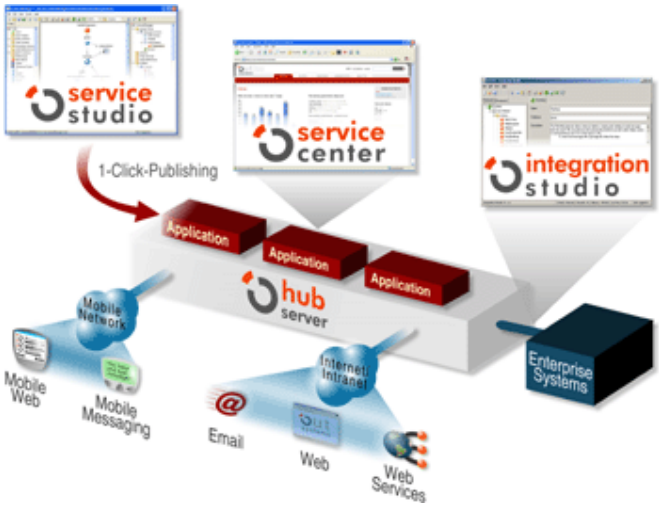


Figure 2 – OutSystems Platform architecture, highlighting the main components and features.
This picture shows the Hub Server as the center of the system. Service Studio does *1-click-publish* of applications to the Hub Server. The applications lie on top of Hub Server infrastructure, and are possibly integrated with other enterprise systems through Integration Studio extensions. Service Center is highlighted as an administration frontend for Hub Server. Finally, the picture also shows the types of supported interfaces: mobile web (WAP), mobile messaging (SMS), email, web, and web services.

[1] In the OutSystems Platform 4.1 it was added support for a modified version of the XSP web server, which is a lightweight .NET web application server, part of the open source Mono project.

3 Compiler Source and Target Languages

In the previous chapter, we have described the main aspects of the platform, highlighting their alignment to the business mission of the OutSystems enterprise. From now on, we will focus on the compilation task performed by the OutSystems compiler, which is an integrated part of the Hub Server component.

Informally, the compiler is called *OSHEComp*[2]. OSHEComp applies compiler techniques to the OML description of the application, and generates the source code of the web application, concretely .NET or Java code. We can therefore identify its source and target languages, which are the main subject of this chapter.

3.1 Source Language Definition

The source language is the eSpace definition, created in the Service Studio. The eSpace file, also called OML, contains everything necessary to compile and deploy the application, except its external dependencies, if any. In this section, we will describe the relevant aspects of the eSpace definition. We will introduce terminology specific to the OutSystems Platform [2], and where applicable we will map it to the terminology used in the literature.

We begin the description of the language by analyzing its procedures. A procedure in OutSystems Platform is mainly called an *action*. There are other procedures which behave basically the same, such as web services. An action can have input parameters and output parameters, and also can declare local variables.

The action code is actually a flow of control, designed with some available primitives such as conditional branches and for each cycles. In Figure 1 we can see an example of such a flow graph in the central pane.

The user interface of the application is designed through *screens*, which are some content sent to the client browser. OutSystems Platform supports web screens, WAP screens and SMS screens, which behave in the same way from the compilers point of view. We will speak of *screens* generally, meaning any supported screen. As we will see, a screen is also another procedure disguised in OutSystems Platform.

A screen can have input parameters, and local screen variables. Each screen, when requested from a user, creates its own execution context. The execution context is made of the actual parameters and the local variables, and the execution context remains active while the user is interacting with the screen. This interaction can last several requests from the browser, each one invoking an action inside the screen context.

The context of a screen is kept between requests through the notion of *viewstate*. The *viewstate* is a collection of every relevant variable of the application, that needs to persist across requests. As long as the user interacts with a screen, the *viewstate* is sent to the client browser on the end of a request,

[2] OSHEComp stands for OutSystems Hub Edition Compiler, and is also the name of the compiler executable bundled with the platform.

and the next request comes with the *viewstate* again, allowing the application server to restore the previous state.

Given this description, we can view the screens as executable objects, which have its own variables and flow of execution control. We can therefore identify screens as a more structured procedure in the OutSystems source language. We will refer to *screen procedures*, meaning this identification of a screen as a procedure.

Mechanisms of exception handling are also provided in the platform through special nodes for throwing and catching exceptions.

The basic data types of the source language cover the common scalar types, such as integer, numeric, boolean, and text. Other types are domain specific, such as date time, currency, and phone number.

In the OutSystems language, each eSpace also has a set of *entities*. Entities help the relational data modeling for the application. Each entity can have one or more attributes, and at most one identifier attribute. The identifier of a given entity also defines a scalar type in the OutSystems language, forbidding the developer to use an identifier of entity *A* where an identifier of *B* is expected.

Entities are mapped into the relational database, so that for each entity there is a table in the database, and for each attribute of the entity, there exists a column in the entity table. The entire database modeling is performed through entities.

For the memory resident objects, the language provides *structures*. A structure can have one or more attributes, of any type. The only restriction is that recursive types are not supported.

Entities and structures can be combined in *records*. A record is a compound type in the OutSystems language.

Records can be stored in *record lists*, which is another compound type in OutSystems.

A *record list*, another compound type, is an array of records. It provides an embedded *current* pointer, and can also be indexed randomly through integer indexes, just like arrays in traditional programming languages.

All assignments in the platform are made by value copy, and not by reference, the only exception being record list assignments. An assignment of a variable of type record list does not copy the list itself, but creates a copy of its *current* pointer. A value copy of record lists is possible through a built-in function, called `ListDuplicate`.

It is also worth of mentioning the available primitives for database querying, as they will be our main target for optimizations.

The OutSystems language provides two primitives for database querying: *simple query* and *advanced query*. They both have input scalar parameters, and a single output parameter of type record list. The only difference between the two primitives is the editor for building the query. Simple queries can be constructed through a simplified editor, which abstracts from the final SQL query. Advanced queries require the developer to write the exact query, and can be used whenever the simple query is not expressive enough.

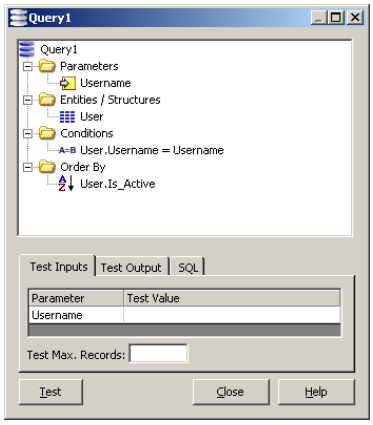


Figure 3 – Simple query editor.
The simple query editor, a querying primitive in the OutSystems language. The query retrieves the users with a given username, ordered by their `Is_Active` attribute. The input parameters are visible in the Parameters folder, and the output parameter is implicitly defined as a record list of the entities and structures present in the query. In the above case, the output parameter is a record list of the entity User.

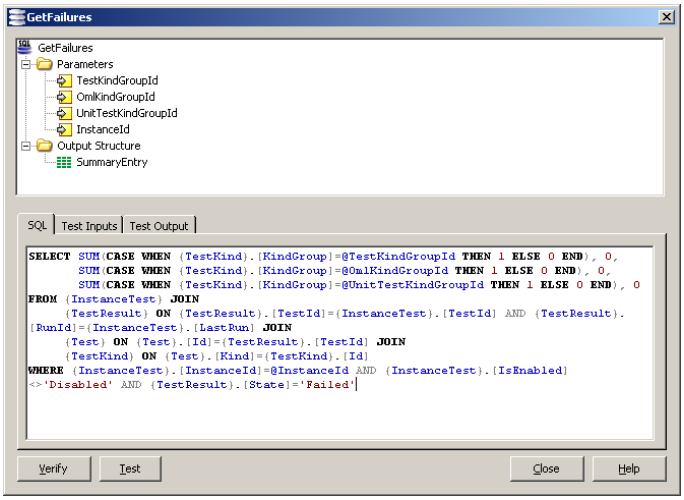


Figure 4 – Advanced query editor.
The advanced query editor, a querying primitive in the OutSystems language. The query is specified in SQL language. The input parameters are visible under the Parameters folder, and the output parameter is a record list of the entities and structures present in the Output Structure folder. In the above case, the output parameter is a record list of the SummaryEntry structure.

In Figure 3, the query is automatically created as a SQL `SELECT` statement. In Figure 4, the `SELECT` statement needs to be defined by the developer.

We should now clarify some limitations imposed by the design of these primitives, and what the automatic optimizer can do to overcome those limitations.

- A simple query outputs every entity and structure which is used in the query. It is a usual case for a query to have many tables joined, but only a few of them is relevant for the application. In SQL, we could specify the complete join in the `FROM` clause, and project only a few of the tables in the `SELECT` clause. This is a manual optimization, and in SQL it would be required to reduce the overhead in data transferring between the database and the application. In a simple query, the

optimizing compiler should decide which entities are relevant for the application, and automatically optimize the underlying SQL query.

- The outputs of both simple queries and advanced queries are record lists, which contains entire entities and structures. It is also a recurrent situation where only a few of the attributes of an entity are required for the application, and the technique used in SQL would be to project only the relevant columns. In the OutSystems language, it should be the optimizer compiler to decide which are the relevant attributes, and automatically perform the projection.
- Both querying primitives does not specify which is the method used for retrieving the result dataset. For instance, there are two main methods: fetching each row on demand, or fetching all rows and storing them in memory. If the query result is iterated only once, the best method is to fetch each row on demand, as it requires less memory. But if the result is iterated more than once, we need to store every row in memory. The optimizing compiler should decide the best method to use.

3.2 Target Language Definition

The target language is .NET [3] source code generated to be deployed in the Microsoft® IIS. The J2EE Hub Server is built as a transformation of the .NET Hub Server, and thus we will omit it in the remaining of the paper.

.NET is a language-neutral environment for writing programs that can easily and securely interoperate. Rather than targeting a particular hardware and operating system, programs can be written for .NET, and will run wherever .NET is implemented as an execution host environment.

The .NET Framework has two main components: the common language runtime and the .NET Framework class library.

The common language runtime is the foundation of the .NET Framework. We can think of the runtime as an agent that manages code at execution time, providing core services such as memory management, thread management, and remoting, while also enforcing strict type safety and other forms of code accuracy that promote security and robustness. In fact, the concept of code management is a fundamental principle of the runtime. Code that targets the runtime is known as *managed code*, while code that does not target the runtime is known as *unmanaged code*.

The runtime is designed to enhance performance. Although the common language runtime provides many standard runtime services, managed code is never interpreted. A feature called JIT[3] compiling enables all managed code to run in the native machine language of the system on which it is executing. Meanwhile, the memory manager removes the possibilities of fragmented memory and increases memory locality-of-reference to further increase performance.

The class library, the other main component of the .NET Framework, is a comprehensive, object-oriented collection of reusable types that you can use to develop applications ranging from traditional

[3] JIT compiling, sometimes referred as JIT – abbreviation for Just-In-Time compiling. The term refers to a compilation technique which converts, at runtime, code from an intermediate code into machine code. The compilation in runtime allows several optimizations that can arise when considering statistics collected from the running process.

command-line or graphical user interface applications to applications based on the latest innovations provided by ASP.NET, such as Web Forms and XML Web services.
For the OutSystems compiler output, the screen layouts are done in ASP.NET [4], and the application logic is targeted as C# [5] source code. This choice of target language was influenced by the market, in which those technologies are currently well accepted by the IT market.

3.3 Optimization Strategy

The choice of the target language as .NET source code (as opposed to a compiled binary) gives us the .NET compiler optimizations without effort. Therefore, the OutSystems optimizer doesn't have to worry about some minor local optimizations, such as constant propagation or dead code analysis. We assume that the .NET compiler used when deploying a web application does a good optimization job.
But there are some other optimizations that can be done in the source language level, that are not possible in the target language. Those optimizations are the best to be addressed by the OutSystems compiler. These are mainly database retrieval optimizations, which aim to reduce database access times. When applied to web applications, these kinds of optimizations are highly desired, mainly because web applications perform database intensive tasks, and usually have little algorithmic spectra.
We are also encouraged to perform optimizations in the *viewstate* parameter kept within the several requests, since a large *viewstate* also degrades the application performance, because a larger bandwidth is needed. These optimizations consist on finding the dead variables at the end of each request, and removing them from the *viewstate*.

4 Optimization Theory

Optimizations of a program can be viewed as transformations that are applied to the program iteratively, yielding benefits in a given metric, such as running time, allocated memory, or binary size.

An important aspect of the optimization transformations is that they must be conservative on the output of the program. An optimization should be completely transparent to the user of the program. The rule of thumb is to never risk changing the meaning of a program.

The best optimizations are those that yield the most benefits for the least effort. The desired benefits should be carefully specified *a priori*, so the optimizer can be designed to meet its requirements.

We saw in the previous chapter that the main requirements for the OutSystems optimizer are database optimizations. We thus describe in this chapter some theoretic tools for optimization techniques, and we evaluate their benefits for the case of OutSystems optimizer.

4.1 Control Flow Analysis

The control flow analysis is, for a given source program, determine its control flow structure. The nodes in the flow graph represent computations, and the edges represent the flow of control.

Control flow analysis is useful to collect topological information about the source code, which can be used to detect hot spots for optimizations. If we can detect that a given region of code is possibly executed many times in a row, we can focus on optimizing this region of code, rather than trying to optimize sections of code that run only a couple of times. This selection of optimizations can reduce the optimization time, and still perform valuable increases in the program performance.

Through control flow we can also infer the order of execution of the program statements, and this information can be further used for more complex data flow analysis, as we are going to see.

We will use as example the following code, which computes the factorial of an integer *n*.

Example of a factorial algorithm.

```
program fact(n)
  i := 1;
  f := 1;
  while (i <= n) do:
    f := f*i;
    i := i+1;
  end while
  output f;
end program
```

Below we enumerate the statements of the preceding program. Note that the expression inside the `while` construct is also a statement itself, which computes the boolean value needed by the conditional branch of execution.

Statements of the program `fact`.

```
S1  i := 1
S2  f := 1
S3  if not (i <= n) goto S6
S4  f := f*i
S5  i := i+1
S6  output f
```

> Definition 1 – **Basic blocks.** A *basic block* is a sequence of statements in which execution flow enters at the beginning and leaves at the end, without halt or possibility of branching, except possibly at the end of the block [6].

Note that a basic block does not have to be a maximal sequence of statements satisfying the mentioned property. In fact, each individual statement can be regarded as a unitary basic block, although this is of little interest because it yields a graph with more nodes and edges.

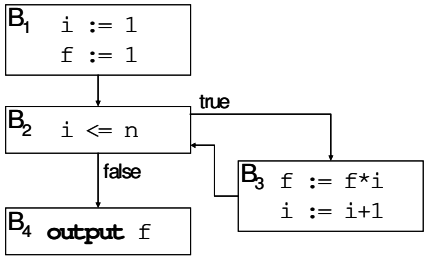


Figure 5 – Control flow of the program `fact`.
Basic blocks are enclosed in rectangles, and arrows are drawn for the possible execution paths. The while can be visually identified, as the loop involving B_2 and B_3.

In Figure 5 we show the basic blocks of the program `fact`. Note that the only branch in the program is made at the end of B_2, which corresponds to either entering the body of the `while` statement, or terminating the cycle and jumping to the `end while` statement.

A basic block is a set of statements which can be considered atomic by the optimizer. Basic blocks can be optimized individually, and the only requirement for the optimizations made is that the program state at the end of the block is preserved. This is a good opportunity for the optimizer to remove

redundant and unreachable code, which are often introduced by the compiler. These optimizations are called *local optimizations*, because they consider only the set of statements that belong to a basic block [6].

> Definition 2 – **Flow graph.** The flow graph is a directed graph, whose nodes are the programs basic blocks, and the edges are all the possible flows of execution between the blocks [7].

The Figure 5 is also the flow graph of the program `fact`, with B_1 being the start node. In OutSystems platform, the flow graph is given directly by the source description language, since there's already a graph that describes the flow of execution. Exceptions should be considered with special care, since every node can possibly jump to an exception handler by throwing an exception.
A flow graph provides us a great tool to perform analysis of the source code. The main analysis that can be done is the detection of loops in the flow graph.

> Definition 3 – **Dominate relationship.** We say that node *d* in the flow graph dominates node *n* (or *n* is dominated by *d*) if every path from the initial node to *n* goes through *d* [6].
> For example, in the `fact` flow graph, B_1 dominates all nodes; B_2 dominates all nodes excluding B_1; B_3 dominates only itself, as does B_4.

> Definition 4 – **Natural loop.** A natural loop must have a single entry point, called the *header*, which dominates all nodes in the loop.

Natural loops can be detected by searching for back edges, which goes from a dominated node to a dominator. When such edge is found, its nodes must be part of the loop; the dominator is the header of the loop and we can travel through the predecessors of the dominated node to build the loop. In the `fact` program, the only back edge is $B_3 \rightarrow B_2$, and we can infer that $\{B_2, B_3\}$ is a natural loop.
Natural loops have the interesting property that they are either disjoint, or completely nested within another [6]. Thus, we can find *inner loops* by searching for loops that doesn't contain any other loop. These loops are the most benefited from optimizations, and should be the focus of the optimizer.

4.2 Data Flow Analysis

The data flow analysis concentrates in the operations that are applied to variables, and the flow of information in the application. This kind of analysis can be used to collect information about the relationships between variable values at each point of execution of the program.
For data flow problems, we concentrate on the manipulation of the program information by a given set of consecutive statements $\Sigma=[S_i \ldots S_j]$. The set Σ can be any set of statements, although for simplicity we generally assume that Σ is a basic block.
We can specify the relevant manipulations according to the problem we are trying to solve. There are four standard problems which we will describe in this section, and each one focuses in different manipulations of data. For each problem, we will define equations for the transformations that take

place in Σ, and solve those equations to find the interesting relations that we are searching for. The equations are called *data-flow equations* [6], and share the form presented in Equation 1.

$$out[\Sigma] = gen[\Sigma] \cup (in[\Sigma] - kill[\Sigma])$$

Equation 1 – Generic data-flow equation.
The equation should be read as "the information at the end of Σ is either generated within Σ, or enters at the beginning and is not killed as control flows through Σ".

We are now able to enumerate some data flow problems, after introducing a few more concepts.

> Definition 5 – **Definitions of a variable.** We say that a statement *S* defines a variable *v* if some execution of *S* might change the value of *v*.

The usual definitions of variables are assignments to them. However, in the general case when the language allows pointers to variables, for example, we may define a variable *v* by an indirect assignment through a pointer to it.
We say that a definition *S* of *v* is killed by *T* if there is a valid execution path from *S* to *T*, and *T* defines *v*.

> Definition 6 – **Availability of an expression.** We say that an expression *x+y*[4] is available at the statement *T*, if it was computed in a previous definition *S*, and the variables *x* and *y* could not be changed between *S* and *T*.

The common case of availability of expressions is when the same value is stored in many variables in sequence, such as in the following example:

Example of availability of expressions.

```
S1  wheel1_radius := tan-1 (k*pi/2)
S2  wheel2_radius := tan-1 (k*pi/2)
S3  wheel1_circumference := 2*pi * tan-1 (k*pi/2)
```

In the example given above, the expression `tan-1 (k*pi/2)` is used three times. But the optimizer can avoid recomputation by detecting that this expression is still available at S_2 and S_3[5], and replacing the repeated expressions with a reference to `wheel1_radius`. This optimization task is also nice for the developer, since he does not need to worry about creating temporary variables to optimize by hand such things.

> Definition 7 – **Live variables.** We define a variable *v* being live at the point *S* in the program, if it can be used in some execution path starting at *S*. On the other hand, we say that the variable *v* is dead at *S* if it would never be used after the control reaches *S*.

Variables which are dead at a point *S* can be deallocated from memory when control reaches *S*, since if *S* is reached, the variables are not needed anymore.

---

[4] Following the same strategy of [6], we use the expression *x+y* as a particular case for the sake of simplicity, but the same definitions can be easily extended to any arbitrary expression, using any combination of operators and variables.
[5] The case of S_3 would require the optimizer to match against sub-expressions of the right hand side of assignments.

Definition 8 – **Use of a variable.** A variable *v* is used at the statement *S* if its *r*-value may be required. For example, the statement `a[b] := c` uses both variables `b` and `c`, but not `a`.

4.2.1 Data Flow Problem: Reaching Definitions

In the reaching definitions problem, we are interested in discovering which is the last definition of a variable *v* in a given point of the programs execution.

For example, in the `fact` program, we have four definitions: S_1, S_2, S_4 and S_5. If we are interested in knowing which is the last definition of the variable *f* in the statement S_6, we can solve the system of equations that arise when applying Equation 1 to each basic block. Note that, in this particular case, after the first iteration on the loop $\{B_2, B_3\}$, the $gen[S_3]$ is propagated to $in[S_2]$, and since they are not killed in S_2, they propagate further to $out[S_2]$.

For reference, the solution to the problem for the `frac` program is:

$$
\begin{aligned}
&in[B_1] = \varnothing \\
&out[B_1] = gen[B_1] = \{S_1, S_2\} \\
&\text{in}[B_2] = out[B_1] \cup out[B_3] = \{S_1, S_2, S_4, S_5\} \\
&out[B_2] = in[B_2] = \{S_1, S_2, S_4, S_5\} \\
&in[B_3] = out[B_2] = \{S_1, S_2, S_4, S_5\} \\
&out[B_3] = gen[B_3] \cup (in[B_3] - kill[B_3]) = \{S_4, S_5\} \\
&in[B_4] = out[B_2] = \{S_1, S_2, S_4, S_5\} \\
&out[B_4] = in[B_4] = \{S_1, S_2, S_4, S_5\}
\end{aligned}
$$

Equation 2 – Solution to the reaching definitions problem for the `frac` program.
Result of applying the Equation 1 to the basic blocks of the `frac` program.

As we can see from $in[B_4]$, both S_2 and S_4 which are definitions of *f* might reach B_4, so we cannot tell at compile time which of the two results will be printed by the `frac` program when the execution takes place.

But if we hypothetically had realized that S_4 could not reach B_4 in the `frac` program, we could just replace the value of *f* in S_6 with the right-hand side of the only definition that reached the **`output`** statement, resulting in S_6 becoming **`output`** `1`. We could then infer at compile time that the program always outputted the value 1, regardless of the execution paths it takes.

4.2.2 Data Flow Problem: Available Expressions

In this data flow problem, we are interested in finding what sub-expressions are available at a given point in the programs control flow. If we manage to retrieve this information, we can then try to match the available expressions with uses of them, and avoid re-computing them by replacing the repeated usages with its previously computed value.

We can define that an expression *x+y* is killed by a sequence of statements Σ if Σ defines *x* or *y*, and does not re-compute *x+y* afterwards. The same expression is generated by Σ if Σ evaluates *x+y* through its execution path, and does not kill it afterwards. The system of data flow equations can be

solved for *in*, which represents the set of available expressions at the beginning of each sequence of statements.

4.2.3 Data Flow Problem: Live-Variable Analysis

In the live variable analysis, we would like to know if a given variable *v* is live at the point *P* in the program.

In this kind of problem, we should define the sets *in*[Σ] to be the set of variables live at the beginning of Σ, and *out*[Σ] to be the set of live variables at the end of Σ. Let *def*[Σ] be the set of variables assigned values in *B* prior to any use of that variable in Σ. And, finally, let *use*[Σ] be the set of variables whose values may be used in Σ prior to any definition of the variable. Then the data flow equations for this problem becomes:

$$in[\Sigma] = use[\Sigma] \cup (out[\Sigma] - def[\Sigma])$$

Equation 3 – Data flow equations for the live variable analysis problem.
A variation of the Equation 1, which is used to solve the live variable analysis problem.

These equations are basically the same as the originals, with the exception that *in* and *out* have exchanged roles, and *use* substitutes *gen* as well as *def* substitutes *kill* [6].

4.2.4 Data Flow Problem: Use-Definition (UD) Chains

The ud-chaining problem is to compute for a point *S* the set of uses of a variable *v*, such that there is at least one path from *S* that does not redefine *v*.

Solving this problem, we are allowed to infer, for a definition *S* of the variable *v*, which statements might use the value computed at *S*.

Just as with live-variable analysis, we use the backward version of the data flow equations:

$$in[\Sigma] = up[\Sigma] \cup (out[\Sigma] - def[\Sigma])$$

Equation 4 – Data flow equations for the use-definition chains problem.
A variation of the Equation 1 in its backward form, which is used to solve the use-definition chains.

4.2.5 Solving Data Flow Equations

We present now an algorithm described in [6] to iteratively solve the system of data flow equations given in Equation 1, in the general case of arbitrary control flows. The other problems can be solved by the same mechanism, since their set of equations are equivalent.

We begin by calculating *gen*[B] and *kill*[B] for each basic block B in the program. We then start with the estimative of *in*[B] = ∅ for each basic block B, and iteratively estimate *out*[B] based on the pre-calculated values of *gen*[B] and *kill*[B].

The iteration step is given by the two following equations, which has to be applied for each block B of the program.

$$in[B] = out[P_1] \cup out[P_2] \cup \ldots \cup out[P_n]$$
$$out[B] = gen[B] \cup (in[B] - kill[B])$$

Equation 5 – Iterative method for solving data flow equations in the general case.
This method solves the data flow equations for both *in* and *out*, given *gen* and *kill* for every basic block.

In Equation 5, $P_1 \ldots P_n$ are the predecessors of B in the flow graph.

Intuitively, the algorithm propagates the definitions as far as they will go without being killed, in a sense simulating all possible executions of the program [6].

4.3 Variable Aliases

In data flow analysis, we commonly assume that every variable is isolated from the others. This assumption means that any definition leaves all variables unchanged, except perhaps the defined variable. This might not be the case, when the source language allows *aliases* for variables – also called *pointers* – which consists in two variables *a* and *b* sharing the same memory location. In those cases, a definition of *a* also defines *b*, and a use of *a* is also a use of *b*.

When the language allows generic pointers, we cannot infer properly which variables are aliased, so we must assume the worst case. In such cases, a definition of a pointer is also a definition to any other variable, and its use is possibly a use of any variable [6].

In languages which allows only variables within a same array to be aliased, we say that a pointer *a* could be an alias of *b* only if they both point to the same array.

Fortunately, the case in the OutSystems language is the later. The only notion of arrays in OutSystems is accomplished by record lists, which are typed linked lists. When a record list is assigned to a variable, it is not copied by value, but by reference instead, leaving a list with two independent pointers to its elements. Those two pointers may point to the same memory location, and thus constitute aliases of one another.

4.4 Inter-procedural data flow

When compiling the application, we usually have all the information about its procedures that we can use to achieve optimizations. For instance, a procedure might not use one of its input parameters. While this can sound a bad design for a procedure, it's often not.

Particularly in OutSystems applications, a procedure might receive a record, or record list, as an input parameter, and might not use all of its attributes. If the record or record list comes from a database query (which is often the case), we may optimize the query not to fetch the attribute value, reducing the data transfer between the application and the database management system.

Several inter-procedural optimizations can be made, such as inlining procedure calls, and optimizing call overhead in some cases. Although these optimizations are a good topic for research, we think that they are not worth optimizing in the class of web applications. We are inclined to the database optimizations, which could lead to greater runtime savings in time, memory and network bandwidth.

Inter-procedural optimizations also can use the extension of data flow to inter-procedural problems. We can construct a *call graph*, whose nodes are procedures, and edges $P \rightarrow Q$ represent a procedure Q call inside procedure P. For each procedure Q, we can define its *gen*[Q] and *kill*[Q], and we can further use this information in the call sites within P to gather inter-procedural data flow information [8].

4.5 Inter-module optimizations

When an application can refer to outside modules, such as compilation units managed independently, we often cannot use optimizations because we have no information about the external module at compile time, and we need to consider the most conservative case.

In OutSystems Platform, in order for an eSpace to be fully successfully published, it must be compiled when every needed dependency is available. Thus, we can gather in compile time information about the external modules. We thus can use this information to optimize the interactions between the application and the external module.

Although this problem relates to inter-procedural analysis, they are not equivalent because the modules are managed independently from the application. However, we can bypass this as an issue by doing inter-procedural optimizations when the entire program is presented to the compiler at once, or by constructing the call graph iteratively as the modules are linked together [8].

5 Inter-procedural Optimizations in OutSystems Compiler

In this chapter, we investigate the problem of optimizing procedure calls in the OutSystems language. We propose a solution to the problem, and its implementation. Finally, we show the results obtained by the chosen solution in real applications.

5.1 Objective of the OutSystems Compiler

In previous sections, we have motivated the reader for the data transferring bottleneck in traditional web applications. These data transfers can become a bottleneck mainly in two distinct cases.

- Data being transferred from the database to the application, as a result of a query.
- Rendered data transferred from the application to the client browser.

We have already mentioned that the OutSystems language defines two data querying primitives, and that both have design limitations that require an optimizing compiler to overcome. Let's rephrase what concerns the optimizer about database transferring.

- The optimizer should decide the optimal projection to be used in the underlying `SELECT` SQL query. To accomplish this task, it needs to retrieve information about the usage of the output of the query in the application, and decide what are the relevant tables and columns in the projection.
- It is desirable to differentiate between a query whose output is never iterated, or iterated only once, or multiple times. Knowing about the iterations of a query helps the optimizer to choose whether to fetch the rows one at a time, or all at once.

To optimize the rendered data, we should optimize the *viewstate* storage, by serializing only the data that is needed. Thus, the compiler needs to know if a given variable can be used after the request is sent to the browser.

5.2 Architecture of OutSystems Compiler

In the OutSystems language, a program is composed of procedures, which can be screens or actions. Both screens and actions have a flow graph, with statements as the graph nodes. The following illustration shows a flow graph for the `frac` program introduced in Section 4.1, implemented in the OutSystems language as an action. It is interesting as the application is designed at the control-flow level, as can be seen by comparing Figure 6 to Figure 5.

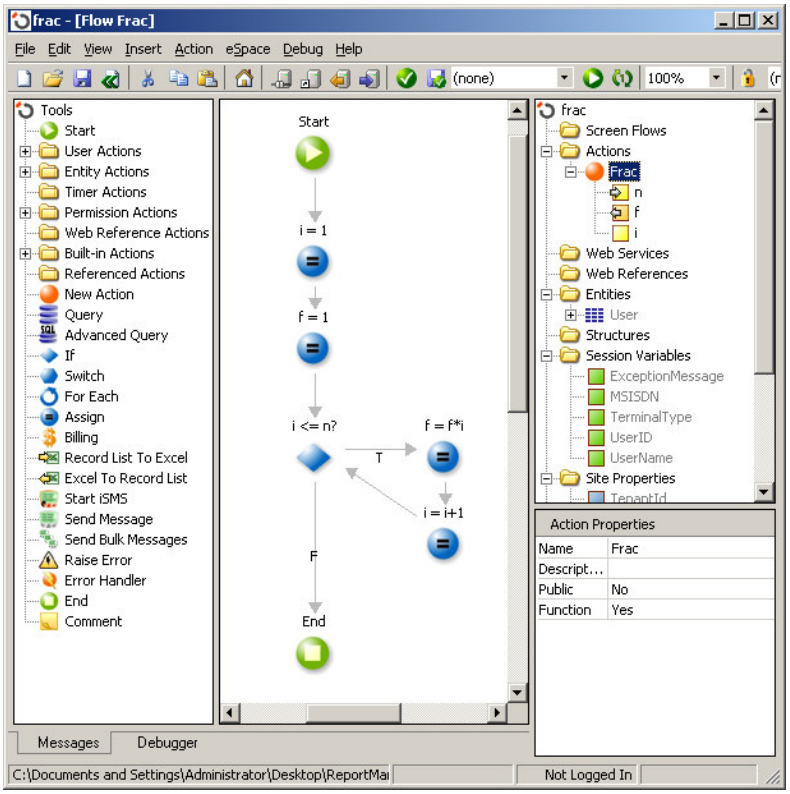


Figure 6 – The `frac` program implemented in OutSystems Service Studio.
In the central pane, we see the implementation of the `frac` program. In the right pane, we can see the Frac action signature, with an input parameter *n*, an output parameter *f*, and a local variable *i*.

In OutSystems language, the flow graphs nodes can be assignments, queries, or procedure calls. Each node has its set of used variables, and a set of defined variables. For example, query nodes have input parameters, which may use variables defined earlier, and an output record list, which is a variable defined by the query node. Nodes for procedure calls also have input parameters and output parameters, which behave exactly in the same way.

In fact, a variable of type record list, as in the case of a query result set, can be viewed just as a compound of its attributes. We can assume that the attributes can be defined and used individually. For this assumption, we define the notion of identifier, which the main data structure handled by the optimizing compiler:

> Definition 9 – **Identifier.** An identifier is an atomic memory location, which can be defined or used. Because recursive types are not supported, every variable of any type has a finite number of identifiers.

The advantage of manipulating identifiers, instead of whole variables, is that we can compute the usage of each identifier individually. The usages of the attributes of a query output can then help optimize the SQL projection used.

The algorithm used for the OutSystems optimizing compiler is a modified solver for the live variables data flow problem, in which we compute liveness of identifiers instead of variables. It optimizes each

procedure at a time, and for each procedure returns the set of identifiers that are used in the given procedure.

When optimizing a given procedure, the optimizer acts conservatively, and considers every output parameter of the procedure as potentially used. In every procedure call, it also considers every input parameter passed to the called procedure as potentially used.

For a given query node, the liveness of each of its attributes is computed. If a given attribute *A* is not live at the end of the query node, it surely is not used at all in the procedure. The optimization which arises from this situation, is that we can exclude the column of attribute *A* from the underlying SQL projection.

The iteration count of every query is also computed. In case the output of a query is used as input to a procedure call, or when it is used as an output parameter, the compiler assumes it is iterated multiple times.

For the *viewstate* optimization, we can compute which are the live identifiers at the point where the screen is rendered and sent to the browser. Each live identifier at this point can potentially be used in subsequent requests, and thus must be serialized into the *viewstate*.

5.3 Use Case for Inter-procedural Optimizations

During the motivation presented in Section 2.2, we stated that the lack of inter-procedural optimizations implied that the application abstractions could not be optimized. In this section, we present a use case where a given abstraction could benefit from inter-procedural optimizations.

Suppose we have an application to manage users. The `User` entity is a special entity in OutSystems language, and is included in every application by default. The following illustration shows the definition of the `User` entity.

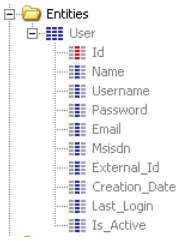


Figure 7 – `User` entity.
The `User` entity, and its attributes. Since it is a system entity, its attributes cannot be changed in the application.

Suppose we want to create an abstraction for the user to use in the application. We could define the following operations:

- `GetUserById`(*UserId*) -> **Record**<*User*>
  Retrieves the record of a user, given its id.
- `GetUserByName`(*Username*) -> **Record**<*User*>
  Retrieves the record of a user, given its username.
- `GetEnabledUsers`() -> **RecordList**<*User*>
  Retrieves the list of all users that are enabled in the application.

- `DisableUsers`(**RecordList**<*User*>) -> *void*

  Receives as input parameter a list of users, and disables them all in the application. Returns nothing.

The set of these operations define an API to manipulate users.

The illustration that follows shows the implementation of one of such operations.

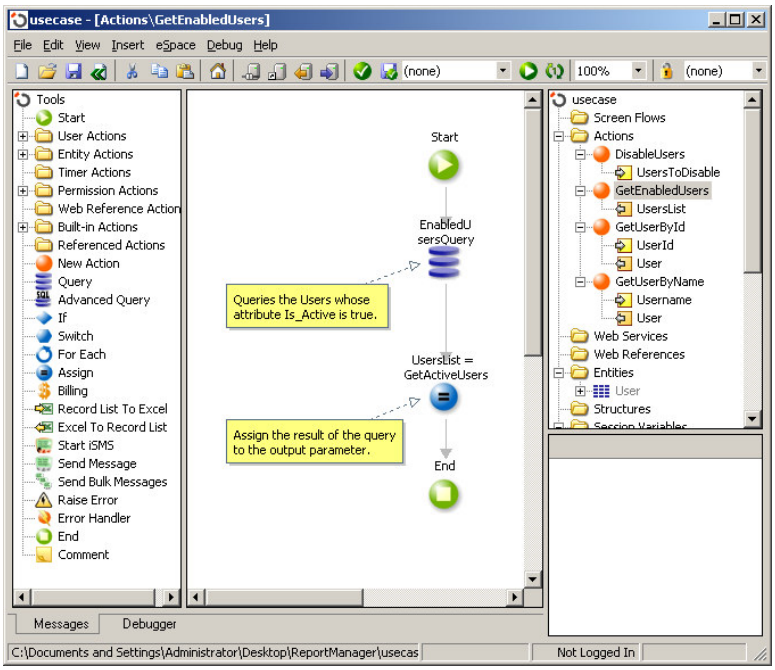


Figure 8 – Implementation of the `GetEnabledUsers` operation in OutSystems Service Studio.
The implementation uses a database Simple Query node, labeled EnabledUsersQuery, to retrieve the enabled users, and outputs the retrieved record list.

The `GetEnabledUsers` action outputs a record list of the `User` entity, which has several attributes. If no inter-procedural optimizations are performed, the worst case needs to be assumed, and every attribute from the `GetEnabledUsers` output needs to be considered potentially used. The lack of such optimizations would imply that the query `EnabledUsersQuery` will not be optimized.

Enabling inter-procedural optimizations would allow the compiler to compute the actual usages of the output of `GetEnabledUsers`, and not just react pessimistically, but realistically.

Suppose now that, in our hypothetical application, the output of the `GetEnabledUsers` is used to display a list of their names. This use case would tell an inter-procedural optimizer that not every identifier outputted by `GetEnabledUsers` is actually used by the application. It would tell even more, that the only used attribute was the `Name` attribute, and perhaps the `Id`. Knowing this information, the compiler could optimize the `EnabledUsersQuery`, and use a underlying SQL projection to only retrieve the `Name` and `Id` attributes.

5.4 Alternatives for Inter-procedural Optimizer Implementations

The class of data-flow problems can be extended to the corresponding inter-procedural problem. We could build a data flow graph for the entire program, in which calls from procedure P to procedure Q are replaced with assignment of actual parameters, and a flow of control to the start of Q.

To solve the data-flow equations in this connected graph we could use the general iterative method in Equation 5.

However, connecting all the graphs of every procedure of a program raises scalability issues. When the number of procedures of a program grew, the OutSystems compiler has always scaled with approximately constant memory, because each procedure is optimized individually. Solving the data-flow problem for the whole program would not scale in memory. Additional memory would be required to compile the existing applications, which would represent a necessary upgrade cost for OutSystems clients. Although we haven't measured the relative difference in memory requirements, it was clear that this approach would impose unwanted scalability constraints in the OutSystems compiler, and we had to research for other alternatives.

The alternative of inlining procedures could not be considered in this case. It consists in replacing the call to a procedure with a code equivalent to the called procedure, which is usually accomplished by duplicating the procedure code, and injecting it in every call site. It was attractive, because it would imply no changes in the optimizer besides the inlining. It would make the application size inevitably grow, requiring more memory by the compiled application. And the growth of the application size would happen with the growth of the number of procedure calls, which is even worse than the previous alternative.

Other alternative which we found was to extend the existing optimizer, in order to make it aware of the inter-procedural optimizations. This ideal could be accomplished by iterative refinement of the inter-procedural usages of variables.

We slightly modify the procedural optimizer, in order to provide to it information about the inter-procedural usage. The procedural optimizer then optimizes a procedure, and making realistic decisions upon the inter-procedural boundaries. On the other hand, the optimization of the procedure yields refined information about the inter-procedural usages that occur inside it.

This approach seemed to have a good trade-off between architectural complexity, memory requirements, compilation time, and impacts in the compiled application. It could be implemented as a layer on top of the existing optimizer, thus having a reduced implementation risk. Since every procedure would still be optimized independently, the memory requirements would be almost the same, with the only difference being the need to store the inter-procedural usages.

Unfortunately it would require a longer optimization time, since every procedure would need to be optimized many times, until the refinement was complete. Although a small compilation time is a desirable feature of a compiler, it is not a critical factor since it does not imply additional hardware costs. Users usually accept the compilation time to slightly grow with a newer release of a compiler, and are willing to wait for a longer compilation at the exchange of an optimized application.

5.5 Inter-procedural Algorithm

In order to introduce the algorithm used by the inter-procedural optimizer, we first start by some notation and definitions.
The following definitions describe the structure of a program, in what concerns the optimizer.

> Definition 10 – **Program ℙ.** A program $\mathbb{P}$ is a set of procedures, $\mathbb{P} = \{P_1, P_2, \ldots, P_N\}$.
>
> Definition 11 – **Variables of a program 𝕍.** By $\mathbb{V}(\mathbb{P})$ we denote the set of all variables of a program $\mathbb{P}$.
>
> Definition 12 – **Variables of a procedure.** Given a program $\mathbb{P}$ and a procedure $P \in \mathbb{P}$, we denote $var(P) \subset \mathbb{V}(\mathbb{P})$ as the set of variables local to procedure P. The output of query primitives, and the outputs of procedure calls are also variables.
>
> Definition 13 – **Input parameters of a procedure.** Given a program $\mathbb{P}$ and a procedure $P \in \mathbb{P}$, we denote $in(P) \subset \mathbb{V}(\mathbb{P})$ as the set of input parameters of P.
>
> Definition 14 – **Output parameters of a procedure.** Given a program $\mathbb{P}$ and a procedure $P \in \mathbb{P}$, we denote $out(P) \subset \mathbb{V}(\mathbb{P})$ as the set of output parameters of P.
>
> Definition 15 – **Call graph.** For a given program $\mathbb{P}$, we can construct a call graph, which is an directed graph with a node for each procedure $P \in \mathbb{P}$, and one edge $P \rightarrow Q$ if P calls Q.

We should clarify that the set of variables of two procedures are not generally disjoint. When a procedure P calls another procedure Q, the output variables of Q become part of the variables of P. Lets state this in a corollary.

> Corollary 1 – **Inter-procedural scope of variables.** Input parameters of P are available only in the scope of P. Output parameters of Q are available in the scope of P if and only if P calls Q. Formally, we have $in(Q) \subset var(P)$ if and only if $P = Q$, and $out(Q) \subset var(P)$ if and only if P calls Q.

The procedural optimizer algorithm, which deals with optimizations local to procedures, is responsible for finding the set of used variables inside a procedure P. It evaluates the liveness of each variable in every point of the procedure. We now define the properties that are relevant for the variables handled by the optimizer.

> Definition 16 – **Used variables.** A variable $v \in var(P)$ is said to be used inside procedure P if it is live in at least one point after its definition in P.
>
> Definition 17 – **Local usage predicate.** For a given procedure P, we define a function called local usage predicate φ_P: $var(P) \rightarrow \{\texttt{true}, \texttt{false}\}$, defined as to have $\varphi_P(v) = \texttt{true}$ if and only if v is used inside P. We might omit the subscript when the procedure P can be clearly inferred from the context.
>
> Definition 18 – **Inter-procedural usage predicate.** For the program $\mathbb{P}$, we define a function called inter-procedural usage Φ: $\mathbb{V}(\mathbb{P}) \times \mathbb{P} \rightarrow \{\texttt{true}, \texttt{false}\}$, defined as follows: for

every two procedures P, Q ∈ $\mathbb{P}$, and a variable v ∈ Q, we have $\Phi(v, P)$ = `true` if and only if v is used inside P. Moreover, we define this function only for input and output parameters of the procedures.

Definition 19 – **Total inter-procedural usage predicate.** For the program $\mathbb{P}$, we define a function called total inter-procedural usage $\overline{\Phi} : \mathbb{V}(\mathbb{P}) \rightarrow \{\texttt{true}, \texttt{false}\}$, defined as $\overline{\Phi}(v)$ = true if and only if v is used in at least one of $\mathbb{P}$'s procedures. The function $\overline{\Phi}$ can also be defined as $\overline{\Phi}(v) = \sum_{P \in \mathbb{P}} \Phi(v,P)$, where the Σ operation stands for the boolean OR.

With the definitions in hand, we are now able to characterize the procedural optimizer algorithm. We do not present its implementation, as it is outside the scope of the work, but it follows a modification of the data flow solving algorithms described in Equation 5. Nevertheless, we describe its relevant properties, which it needs to satisfy in order to be used in the inter-procedural optimizer.

Function `procedureOptimize`
Inputs: `Procedure P; Total inter-procedural usage` $\overline{\Phi}$
Outputs: `Local usage` φ

The function `procedureOptimize` optimizes a given procedure P, and outputs the computed local usage predicate for the procedure P. It additionally uses the total inter-procedural usage predicate $\overline{\Phi}$ in order to be able to optimize the inter-procedural boundaries. It should also obey the following corollary:

Corollary 2 – **Dependencies of the local usage predicate.** The local procedure usage φ_P for a given procedure P depends only of the P structure, the inter-procedural usage of its output parameters, and the inter-procedural usage of the input parameters of the called procedures.

Corollary 2 follows directly from the definition of liveness. In the scope of a procedure, the definitions of variables flow either to its output parameters, or to the input parameters of other procedures. Thus Corollary 2 is not an additional requirement for `procedureOptimize`.

We now present the inter-procedural optimizer, which uses the function `procedureOptimize` iteratively to refine the inter-procedural usages of the program.

```
Function optimize
Inputs: Program ℙ
Outputs: set of used variables of ℙ
Algorithm:
          L1   call-graph ← buildCallGraph(ℙ)
          L2   initialize Φ(0) such that:
                 v ∈ in(P) ⇒ Φ(0)(v, P) is true
                 v ∈ out(Q) and P calls Q ⇒ Φ(0)(v, P) is true
                 otherwise Φ(0)(v, P) is false
          L3   i ← 0
          L4   while i = 0 or Φ̄(i) ≠ Φ̄(i-1) do:
          L5     i ← i+1
          L6     Φ̄(i) ← Φ̄(i-1)
          L7     for each P ∈ ℙ do:
          L8       φP(i) ← procedureOptimize(P, Φ̄(i))
          L9       for each v in in(P) do:
          L10        Φ(i)(v, P) ← φP(i)(v)
          L11      for each Q ∈ ℙ such that P calls Q do:
          L12        for each v in out(Q) do:
          L13          Φ(i)(v, P) ← φP(i)(v)
          L14  output {v ∈ 𝕍(ℙ) | φP(i)(v) is true for some P ∈ ℙ}
```

In line L1, we build the call graph of the program. The call graph is an important data structure for inter-procedural problems, since it synthesizes the calls between the procedures of a program. In this algorithm, it can be used to efficiently tell if a procedure calls another.

The initialization in line L2 represents the pessimistic assumption that all input and output parameters are potentially used, in the conditions of Corollary 1.

In L8, the procedural optimizer is called for a procedure P, and returns information about the usages inside P. All the following lines, from L9 up to L13, stores into $\Phi^{(i)}$ information about the inter-procedural variables used inside P. They use Corollary 1 to cut down the number of updates to $\Phi^{(i)}$.

Finally, in line L14 after the stabilization of the algorithm, we find all the variables of the program ℙ which are used inside some procedure P. For each procedure, we use its most recently calculated local usage predicate $\varphi_P^{(i)}$, to discover its used variables.

5.5.1 Improvements

When optimizing a procedure P, there could be two distinct cases where an inter-procedural optimization can take place. The first one, is that any variable *v* which is used only by an output parameter *o*, ceases to be live when $\overline{\Phi}(o)$ = `false`. The other case, where a variable *v* is used as input parameter *i* to a procedure Q, and $\overline{\Phi}(i)$ = `false` implies *v* being not live. The optimization of *v* could eventually affect $\overline{\Phi}$, and open opportunities for other optimizations in other procedures.

We can describe this effect as a flow in inter-procedural optimizations, where a single optimization implies a chain of other optimizations.

It is interesting if we could perform the entire chain of optimizations in the same iteration. That becomes possible if the procedures are optimized in the same order as the chain of optimizations. So let's introduce the topology of the graph which holds the possible optimization chains.

> Definition 20 – **Inter-procedural chain.** We say that there's an inter-procedural chain from procedure P to procedure Q, with $P \neq Q$, if a change in $\Phi(P, v)$ from `true` to `false` implies a change in φ_Q for some variable $v \in$ var(P).
>
> Definition 21 – **Inter-procedural chain graph.** The inter-procedural chain graph is a directed graph, with a node for each procedure P, and an edge from $P \rightarrow Q$ if there's an inter-procedural chain from P to Q.
>
> Theorem 1 – **Inter-procedural chain graph topology.** Given a program $\mathbb{P}$, and its call graph G, we define G^{-1} as being a directed graph with a node for each procedure, and an edge $P \rightarrow Q$ if one of $P \rightarrow Q$ or $Q \rightarrow P$ is an edge of G. Then the inter-procedural chain graph is a sub graph of G^{-1}.
>
> *Proof.* Theorem 1 is equivalent to say that there's an inter-procedural chain from P to Q, only if either P calls Q or Q calls P.
>
> In fact, it follows from Corollary 2 that changing the value of $\Phi(P, v)$ for $v \in$ in(P) could possibly impact the callers of P. Also because of Corollary 2, if P calls Q, then changing the value of $\Phi(P, v)$ for $v \in$ out(Q) could have implications Q's local usage. This proves that there's an inter-procedural chain from P to Q if P calls Q, or Q calls P.
>
> On the other hand, if P doesn't call Q, and Q doesn't call P, it follows from Corollary 2 that neither procedure can have influence on the local usage of the other.■

Finding a linear ordering of the inter-procedural chain graph is not always possible, because it is a cyclic graph. But we can at least improve the ordering to meet a subset of the chain. By spanning a maximum acyclic sub graph of G^{-1}, we are able to determine a linear ordering of the procedures, which is coherent with a subset of the possible chains. The criteria for determining the spanning tree can be arbitrary.

Other improvement which can be made to the algorithm is to note that, from Corollary 2, the calculation of $\varphi_P^{(i)}$ in line L8 depends only in the inter-procedural usage of the inputs of the procedures called by P. It follows that, if these usages do not change from iteration i-1 to iteration i, the predicate $\varphi_P^{(i)}$ will be equivalent to $\varphi_P^{(i-1)}$. This property is useful, so we can avoid recalculating the usages of procedures that are already stable, and focus on the procedures that can be further optimized.

5.6 Implementation

The original OutSystems compiler used a data-flow solving algorithm, with a pessimistic strategy for inter-procedural interactions. Internally, the algorithm dealt with finding the liveness of identifiers, because variables in OutSystems are a well-defined concept. Although we speak of identifiers in the context of the OutSystems compiler, this definition is semantically equivalent to the variables referred by the literature.

Upon its initialization, the algorithm calculated every definition and usage for each statement of a given procedure. By definition, the start node of a procedure defined the input and output parameters. For a node which could yield an output of the program – such as storing a value in the database, changing a session variable, or rendering of a widget – the algorithm calculated the used identifiers. The end nodes in a procedure, by definition, used all output parameters of the procedure. This allowed to compute the liveness of every identifier, assuming that all the outputs of the procedure were used when the procedure ended its execution.

Similarly, when calling a procedure, every actual input parameter was considered as used.

The first step in the implementation was to discontinue these assumptions. The algorithm should now receive as input the information about what are the live inputs and outputs of a procedure.

There were two modifications to the original algorithm to benefit from this information. The end node of the procedure sets the used variables according to its total inter-procedural usage $\overline{\Phi}$. Also, when a procedure is called, we do not set as used the expressions used as input parameters whose total inter-procedural usage is `false`.

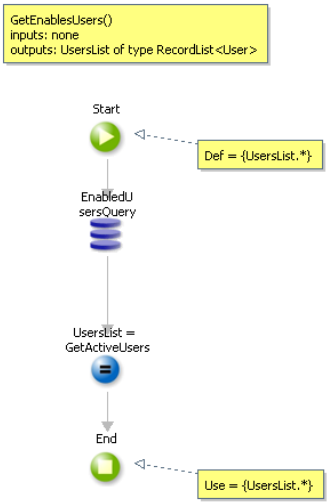


Figure 9 – `GetEnabledUsers` definitions and uses without inter-procedural optimizations.
The procedure `GetEnabledUsers`, showing the definitions in the start node, and the uses in the end node. Without inter-procedural optimizations, every output is considered to be used after the procedure execution.

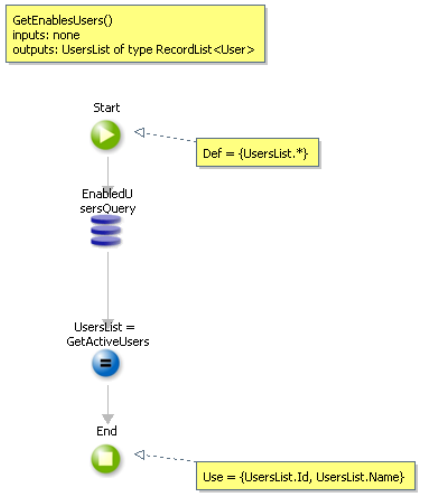


Figure 10 – `GetEnabledUsers` definitions and uses with inter-procedural optimizations.
The procedure `GetEnabledUsers`, showing the definitions in the start node, and the uses in the end node. With inter-procedural optimizations, the end node now holds the realistic assumptions about the usage of the output identifiers.

With the implementation of `procedureOptimize` in hands, we could finally implement the inter-procedural optimizer.

The call graph of the program is built, and a maximum spanning tree algorithm is used to obtain an acyclic graph. To calculate the maximum spanning tree, the call graph was labeled with negative values, in our implementation proportional to the number of call sites between the procedures. Then a minimum spanning tree algorithm was applied, and computing a forest instead of just a tree, to allow disconnected call graphs that arise in applications with many entry points.

The topological order of the forest obtained is used to improve the convergence of the inter-procedural chains. We have performed tests to validate this choice, by running the algorithm without any special order, and running it with the topological order, and its inverse.

For the predicate Φ, we used a composed structure called `InterProceduralUsage`. For the input parameters, the structure stored in a hash-table the set of live identifiers, with the procedure as key. For the output parameters, we had to use a matrix, that for each pair of procedures (P, Q), stored the set of output parameters from P which were used in Q.

The interprocedural optimizer, at the end of each procedure optimization, updates the usages of the input parameters and output parameters in the InterProceduralUsage structure, corresponding to lines L9 to L13 in the algorithm.

When every procedure has been optimized in a given iteration, we store the number of used inputs and outputs for each procedure. Since the algorithm is monotonically convergent, this number can only decrease. We compare these counts with the counts from last iteration, and if they are equal we declare the procedure as being stabilized. The iteration has any changes if at least one procedure is not stable.

Our implementation also checks for redundant optimizations. It does not optimize a procedure again, if it already has stabilized, and every of its called procedures are also stable. This is a consequence of Corollary 2.

The implementation also dealt with several architectural issues, such as having two distinct ways of calling a procedure (as an ExecuteAction node, and as a function call), public procedures, and assignments of record lists creating aliases.

During the implementation of the algorithm, it was made three deliverables, to be tested later and compare their differences. They were:

- The *naïve* implementation of the algorithm, that is to say, without any improvement mentioned in 5.5.1. This executable was called OSHEComp v1.
- The optimizer improved to skip unnecessary procedure optimizations. This binary was called OSHEComp v2.
- The final deliverable, which additionally has the improved ordering of the procedures. It was called OSHEComp v3.

5.7 Results

In this section, we summarize the results gathered throughout this work, and the methodology for obtaining them. The raw data supporting the results hereby claimed are available in Annex.

All results were obtained by using a PC with processor Intel Pentium 4 at 2.8GHz, 2GB of RAM memory, and running Microsoft Windows XP with Service Pack 2. The system had OutSystems Platform 4.0 .NET installed, with Microsoft SQL 2005 installed locally to support the applications.

To measure the results of this approach in the OutSystems applications, it was taken a sample of 52 real world applications, both for OutSystems internal use, and provided by clients. We ran the original compiler, and chose the 15 most complex applications to participate in the tests. Our complexity measure was the peak memory during the compilation of the application. The following table and chart shows the preliminary selection of these 15 applications.

Application	*Application Size (Compressed MB)*	*Compiler Peak Memory (MB)*	*Compilation Time*
EMS_PV.oml	1.766	588	0:00:54
issues.oml	2.773	452	0:00:42
ECHO_UI.oml	3.813	368	0:02:04
Billing.oml	2.079	343	0:00:44
reg_dashboard40.oml	2.223	323	0:00:26
ServiceCenter.oml	3.256	314	0:00:36
SLSRetail.oml	1.689	294	0:03:34
EMS_Customers.oml	1.855	274	0:00:24
EnergyMeasures.oml	1.359	216	0:00:16
regressiontool.oml	1.739	207	0:00:24
EMS_Assets.oml	1.131	187	0:00:13
EnterpriseManager.oml	1.301	164	0:00:16
EMS_Tennet.oml	0.858	157	0:00:10
IssueManager.oml	1.017	147	0:00:18
EMS_ProdProfile.oml	1.057	142	0:00:13

Table 1 – Preliminary selection results.
The table shows the 15 applications chosen to participate in the results measurement. The other applications were omitted in the table, but appear in the graph in Figure 11.

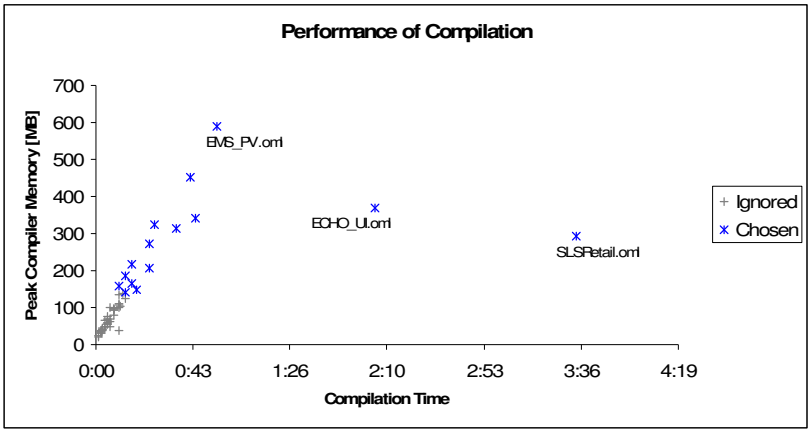


Figure 11 – Graph for the performance of compilation of the available applications.
Applications are plotted with the compilation peak memory against the compilation time. The chosen applications are distinguished from the ignored applications, and it's clearly visible the low relevance of the ignored applications.

In the graph in Figure 11, we can distinguish three applications which seem to be interesting to evaluate in performance terms. They are EMS_PV.oml, ECHO_UI.oml and SLSRetail.oml. The EMS_PV.oml application is the one we consider to be the most complex one, since it's the one that takes up more memory. The later applications seem to have a superior compilation time, although they do not use as much memory as EMS_PV.oml.

Given the chosen applications, we proceeded to the measure of the overall compilation performance, in different stages of development. We measured the compilation time, and peak memory, aiming to compare:

- the official released compiler (OSHEComp 4.0);
- the *naïve* inter-procedural compiler (OSHEComp v1);
- the inter-procedural optimizer improved to skip unnecessary optimizations (OSHEComp v2);
- and the final inter-procedural optimizer, with the ordering of the procedures (OSHEComp v3).

Every application was compiled four times, for every tested version of the compiler, and the average was taken. The peak memory difference between OSHEComp 4.0 and OSHEComp v3 is 20MB, which represents 7% of increase in memory requirements. The compilation time was risen by 5 seconds, which is 10% of the average compilation time.

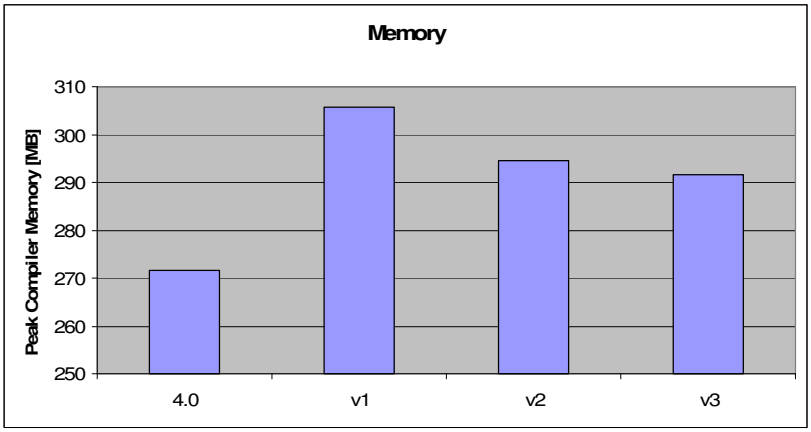


Figure 12 – Peak memory comparison.
The graph shows the differences between the peak memory consumption of several versions of the compiler. The values represent the average of the peak memory for all the tested applications.

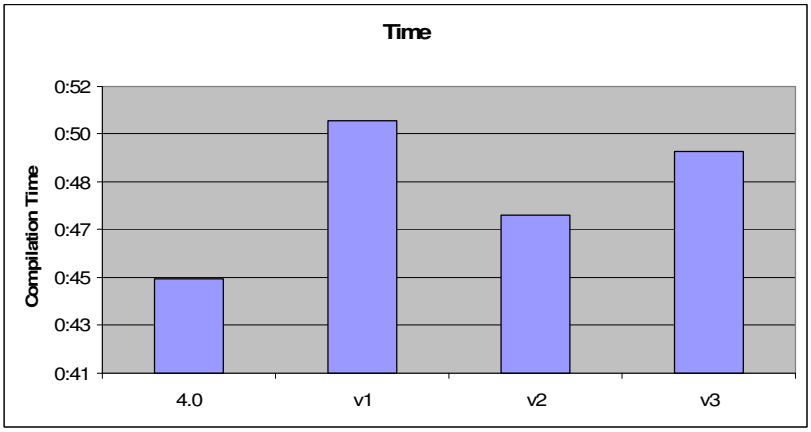


Figure 13 – Compilation time comparison.
The graph shows the differences between the compilation time of several versions of the compiler. The values represent the average of the compilation time for all the tested applications.

For each application, we have performed its compilation with the final inter-procedural compiler, OSHEComp v3. We have gathered information about the application itself, and the optimization stage, such as the number of iterations, and what are the individual gains at every iteration. We have compared the results obtained, with the original algorithm from OSHEComp 4.0.

The results for every individual application are available in Annex. The query optimizations gains, when compared to the old algorithm, vary from 0% in EMS_Tennet.oml to 25,75% in EMS_ProdProfile.oml. The average static gain in the applications tested was 7%.

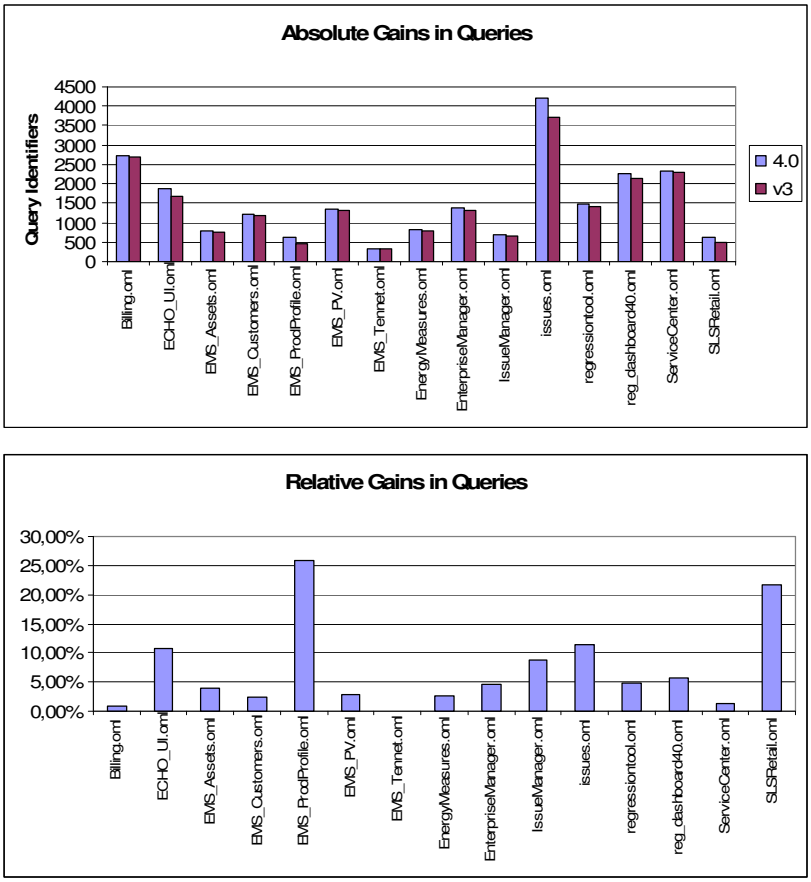


Figure 14 – Comparison of the gains in query identifiers between OSHEComp 4.0 and OSHEComp v3.
The figure shows the absolute and relative gains of the tested applications, when a comparison is made between the official compiler OSHEComp 4.0, and the inter-procedural compiler OSHEComp v3.

The inter-procedural optimizer, as would be expected, is slower than the procedural optimizer. The following chart compares the optimization times of the old optimizer in OSHEComp 4.0, and the inter-procedural optimizer in OSHEComp v3.

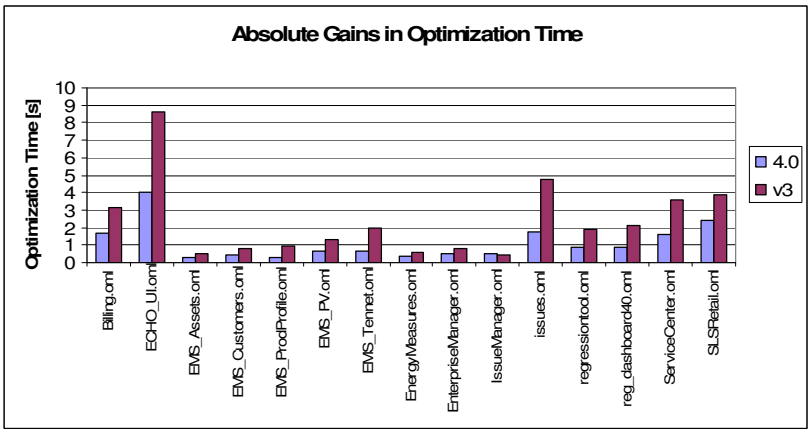


Figure 15 – Comparison of the optimization times between OSHEComp 4.0 and OSHEComp v3.
The figure shows a comparison between optimization times of the official compiler, and the inter-procedural compiler.

The chart in Figure 16 shows the convergence of the applications in the inter-procedural algorithm. Each application took a maximum of 4 iterations. It is interesting that, on the average, 66% of all inter-procedural optimizations of an application will occur in the first iteration.

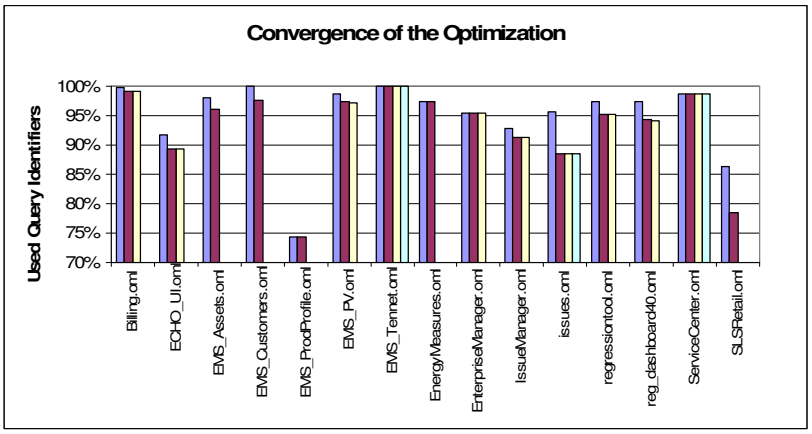


Figure 16 – Normalized convergence of the individual applications for the OSHEComp v3.
The chart presents the iterations of the applications, and the percentage of used query identifiers in each iteration. The reference value, 100%, is the result which would be obtained by OSHEComp 4.0.

6 Future Work

The current implementation of the algorithm in the product lacks support for partial compilation, which is a feature of the OutSystems Platform. It allows compiling only the modified procedures of an application and maintains the unmodified procedures from a cache.

With inter-procedural optimizations, this feature needs to be reviewed, since modifying a given procedure could invalidate the generated code for an unmodified procedure, because an inter-procedural optimization no longer applies in the newer version. This is a work beyond the scope of this thesis, but should be done before this kind of optimizations is released with the product.

Related work already exists in the area of partial compilation. The authors of [9] use a framework to store the inter-procedural information upon compilation, and to track the changes to the procedures source code. If a given module is changed, the framework is capable of knowing which the procedures, if any are, that needs recompilation. It is only required that the compilations of a program be stored in a database, and that the editing of the source code can only be done through a tool which is integrated with the framework.

In fact, their work is very suitable to be applied at OutSystems. We do have access to a database, where we can store inter-procedural information. And every supported editing operation is done by Service Studio, to which we have complete control.

The current implementation also needs several treatments before it is officially a part of OutSystems platform. For instance, it lacks testing and quality assurance evaluations.

We could also perform better optimizations if we specialize the procedures for some frequent usages. Suppose we have a procedure P which has outputs {A, B, C}. One typical usage of the procedure uses output A, but ignores outputs B and C. But scattered through the same application, there are other usages of the same procedure. Sometimes the output B is used, sometimes C, and sometimes the three outputs are needed in a given usage.

In such a situation, our algorithm cannot optimize P, because all of its outputs are live in a given point of the application. To be able to tackle this problem, we should specialize the procedure P for its typical usages, following an approach similar to procedure inlining.

We compile twin copies of P, such as P_1 and P_2. When the procedure P is called and only the output A is used, we replace it with P_1. In every other usage, we replace P with P_2. It follows that our algorithm now can perform a relevant optimization in P_1, because by definition A is its only live output.

For such modification of the algorithm, we should be able to determine statically which are the most profitable specializations. We could come up with a heuristic to cluster the usages of a procedure. Whenever a procedure P is called, we determine what is the subset O of outputs used in the call site. Then we determine the N most frequent subsets in the whole application, and specialize those procedures.

The specialization as we described could be performed after the stabilization of the pure algorithm, without any modification. At this point, we could easily determine the usages in every call site of the application, choose the most profitable specializations, modify the call graph, and continue the algorithm until it stabilizes again. The second time the algorithm runs, we expect that most of the

procedures will remain stable, and only the specialized procedures and their callers would need to be iterated.

Another way to deal with specialization is to dynamically select which specialization is preferred when invoking the procedure in runtime. Suppose we have a procedure P, and we compile some arbitrarily chosen specializations P_1, P_2, …, P_N, optimized for output usages O_1, O_2, …, O_N. When a procedure calls P, it provides the set of needed outputs in that particular call site. In runtime, the OutSystems Platform could decide which specialization fits in the requirements of the call being made. In the worst case, it would fall back to the procedure P compiled without inter-procedural optimizations.

With this approach, the partial compilations would become trivial to implement. It would provide an invariant interface to the procedure, supporting transparently any call, and deciding which optimizations to use for each particular case. This interface could provide optimizations even to calls from outside of the application, from code which was unknown at compile time.

Such a unified interface would also allow the evolution of the inter-procedural optimizer. Two applications compiled with different versions of the inter-procedural optimizer would benefit from inter-procedural optimizations when communicating, as long as they respect the interface defined.

OutSystems is inclined to research in the inter-procedural optimizations, with the main focus on the specialization of procedures, and with the objective to achieve optimizations even from cross-application calls.

We could also use statistical data for an application to base our optimization strategy. For example, we could store analytical information about an application in runtime, and use it to make static decisions when compiling newer versions. We could concentrate the optimization effort in the most executed queries in the history of the application.

There is also a topic of research, which is to use a runtime profiler to collect information about the current execution of the application, and use it to perform dynamic transformations to the code. We could follow a JIT strategy, and specialize the procedures in runtime, as they are needed. This would require dynamic compilation of code.

7 Conclusion

The main objective of this work, which was to implement inter-procedural optimizations in the OutSystems compiler, has been achieved successfully.

We have presented concrete results, which show that the inter-procedural optimizer performs better than the original compiler, according to our static measures, by providing 7% more optimizations over the query fields. It was also shown that, for the average of the tested applications, the proposed inter-procedural optimizer uses 7% more memory than the original, and in average spends 10% more on the compilation time.

Because it was built upon the old procedural optimizer, it involved very few architectural changes, highly reducing the risk and cost of the project. Given the current dimension of the compiler, which has more than 70.000 lines of C# code, its modularity is becoming a concern.

As we have said in section 2.2, an inter-procedural optimizer adds a great value to OutSystems product, because now the users of the OutSystems platform are able to create structured applications, without the performance issues it would incur without inter-procedural optimizations. This work is also a strategic step for optimizing the interfaces between two OutSystems applications.

We are also aware of the bad practices building the OutSystems applications because of lack of inter-procedural optimizations. We believe that the optimization rate of 7% we have obtained could be increased if the applications were better planned, and designed following encapsulation and well-defined interfaces.

8 References

[1] Paulo Rosado, CEO OutSystems: *Company Overview*. Available in OutSystems Portal (http://www.outsystems.com/agile_software/Company.aspx) at January 30, 2007.

[2] OutSystem Documentation: *Service Studio 4.0 Help* (2006)

[3] Microsoft Documentation: *.NET Framework Conceptual Overview*. Available in MSDN Library (http://msdn2.microsoft.com/en-us/library/zw4w595w.aspx) at January 30, 2007.

[4] Jesse Liberty, Dan Hurwitz: *Programming ASP.NET – Building Web Applications and Services*. 3rd Edition. O'Reilly (2005)

[5] Jesse Liberty: *Programming C# – Building .NET Applications*. O'Reilly (2001)

[6] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: *Compilers – Principles, Techniques and Tools*. Addison-Wesley (1986)

[7] Randy Allen, Ken Kennedy: *Optimizing Compilers for Modern Architectures*. Morgan Kaufman (2002)

[8] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman (1997)

[9] Michael Burke, Linda Torczon. *Interprocedural Optimization: Eliminating Unnecessary Recompilation*. In ACM Transactions on Programming Languages and Systems, Vol. 15, No 3. July 1993, pages 367-399.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. McGraw-Hill (2002)

[11] Thomas W. Parsons. *Introduction to Compiler Construction*. W H Freeman & Co (1992)

[12] Charles Fischer, Richard J. LeBlanc Jr. *Crafting a Compiler with C*. Addison-Wesley (1991)

[13] Mary Wolcott Hall. *Managing Interprocedural Optimization*. Rice University (1991)

[14] Vugranam C. Sreedhar, Michael Burke, Jong-Deok Choi. *A Framework for Interprocedural Optimization in the Presence of Dynamic Class Loading*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Vol. 35, No 5. May 2000.

Annex

	Application	Application Size (Compressed MB)	Compilation Peak Memory (MB)	Compilation Time
Chosen	EMS_PV_v286.oml	1.766	588	0:54
	issues.oml	2.773	452	0:42
	ECHO_UI_v1908.oml	3.813	368	2:04
	Billing.oml	2.079	343	0:44
	reg_dashboard40.oml	2.223	323	0:26
	ServiceCenter.oml	3.256	314	0:36
	SLSRetail.oml	1.689	294	3:34
	EMS_Customers.oml	1.855	274	0:24
	EnergyMeasures.oml	1.359	216	0:16
	regressiontool.oml	1.739	207	0:24
	EMS_Assets.oml	1.131	187	0:13
	EnterpriseManager.oml	1.301	164	0:16
	EMS_Tennet.oml	0.858	157	0:10
	IssueManager.oml	1.017	147	0:18
	EMS_ProdProfile.oml	1.057	142	0:13
Ignored	Customer.oml	0.972	139	0:12
	EMS_CustomerPortal.oml	1.037	135	0:10
	IM_Obj.oml	0.813	123	0:13
	RenewalEngine.oml	0.566	109	0:10
	RenewalBO.oml	0.723	106	0:10
	Sales.oml	0.422	103	0:11
	EMS_Nomination.oml	0.433	100	0:06
	EMS_ITron.oml	0.782	99	0:10
	IM_API.oml	0.534	97	0:08
	EMS_ProfileManager.oml	0.645	96	0:08
	EMS_MessageExchange.oml	0.641	92	0:08
	EMS_ECP.oml	0.580	92	0:08
	MVSConnector.oml	0.599	78	0:08
	PushContents.oml	0.300	77	0:05
	EMS_WorkForceBilling.oml	0.474	69	0:06
	EMS_PowerBid.oml	0.322	68	0:05
	EMS_Portal.oml	0.290	65	0:04
	EMS_KW3000.oml	0.322	64	0:05
	EMS_MobileWorkForce.oml	0.273	63	0:06
	Enterprise.oml	0.230	60	0:05
	EMS_ContractManager.oml	0.372	59	0:05
	EMS_CRM_v218.oml	0.669	54	0:05
	EMS_WorkForceConnect.oml	0.185	54	0:05
	PartnerAPI.oml	0.169	50	0:04
	AuditEvents.oml	0.134	49	0:04
	VOS_CombipuntConnect.oml	0.277	48	0:06
	VOS_AxaptaConnector.oml	0.144	45	0:03
	SyncContentsWS.oml	0.165	43	0:03
	SSIE.oml	0.052	41	0:03
	WidgetLibrary40.oml	0.123	38	0:10
	EMS_ECH.oml	0.126	37	0:02
	EMS_SupplierSync.oml	0.089	37	0:03
	EMS_GridOpSync.oml	0.080	36	0:02
	EMS_EnergyICT.oml	0.059	36	0:02
	EMS_Metering.oml	0.055	32	0:02
	InstallBase.oml	0.051	31	0:02
	EMS_MessageBroker.oml	0.017	24	0:01

Annex 1 – Table with statistics for the preliminary selection of the most complex 15 applications.
The table shows the results of the compilation for the 52 sample applications. The results were sorted by peak memory consumption, and the top 15 applications were chosen to participate in the results gathering process.

Application	*Mem.*	*Time*	*Mem.*	*Time*	*Mem.*	*Time*	*Mem.*	*Time*	*AVERAGE Mem.*	*AVERAGE Time*
Billing.oml	341	0:36	343	0:43	343	0:44	342	0:43	342	0:42
ECHO_UI_v1908.oml	373	2:05	368	2:04	366	2:04	385	2:05	373	2:05
EMS_Assets.oml	129	0:14	183	0:14	130	0:14	138	0:14	145	0:14
EMS_Customers.oml	285	0:24	273	0:24	275	0:24	281	0:24	279	0:24
EMS_ProdProfile.oml	160	0:13	149	0:13	152	0:13	152	0:13	153	0:13
EMS_PV_v286.oml	565	0:54	582	0:55	563	0:54	589	0:54	575	0:54
EMS_Tennet.oml	158	0:10	155	0:11	155	0:11	157	0:10	156	0:11
EnergyMeasures.oml	226	0:16	223	0:16	219	0:15	225	0:16	223	0:16
EnterpriseManager.oml	164	0:16	165	0:16	174	0:15	163	0:16	167	0:16
IssueManager.oml	144	0:18	136	0:17	142	0:17	138	0:18	140	0:18
issues.oml	409	0:41	402	0:42	292	0:41	410	0:41	378	0:41
regressiontool.oml	207	0:24	207	0:24	197	0:24	197	0:24	202	0:24
reg_dashboard40.oml	325	0:26	324	0:26	323	0:26	347	0:27	330	0:26
ServiceCenter.oml	314	0:36	316	0:37	318	0:37	314	0:36	316	0:37
SLSRetail.oml	294	3:33	295	3:35	293	3:35	295	3:35	294	3:35
AVERAGE	*273*	*0:44*	*275*	*0:45*	*263*	*0:45*	*276*	*0:45*	*272*	*0:45*

Annex 2 – Table with the peak memory and compilation time for the OSHEComp 4.0.
The table presents the measures for the compilation of the applications. Each application was compiled with OSHEComp 4.0 four times, and the average was taken in the last column.

Application	*Mem.*	*Time*	*Mem.*	*Time*	*Mem.*	*Time*	*Mem.*	*Time*	*AVERAGE Mem.*	*AVERAGE Time*
Billing.oml	327	0:50	327	0:50	326	0:50	327	0:48	327	0:50
ECHO_UI_v1908.oml	547	2:29	547	2:29	547	2:29	547	2:28	547	2:29
EMS_Assets.oml	163	0:15	217	0:16	223	0:16	217	0:15	205	0:16
EMS_Customers.oml	311	0:26	303	0:26	213	0:26	311	0:26	285	0:26
EMS_ProdProfile.oml	161	0:15	148	0:15	163	0:15	159	0:15	158	0:15
EMS_PV_v286.oml	622	0:58	624	0:58	622	0:59	622	0:58	623	0:58
EMS_Tennet.oml	170	0:17	170	0:17	170	0:17	170	0:17	170	0:17
EnergyMeasures.oml	189	0:18	201	0:18	195	0:18	201	0:18	197	0:18
EnterpriseManager.oml	208	0:18	209	0:18	208	0:18	208	0:18	208	0:18
IssueManager.oml	152	0:19	155	0:19	151	0:19	151	0:19	152	0:19
issues.oml	424	0:58	385	0:57	407	0:57	368	0:57	396	0:57
regressiontool.oml	283	0:31	283	0:30	302	0:32	301	0:31	292	0:31
reg_dashboard40.oml	320	0:35	306	0:36	328	0:36	320	0:35	319	0:36
ServiceCenter.oml	418	0:51	409	0:51	409	0:51	418	0:51	414	0:51
SLSRetail.oml	299	3:20	288	3:17	298	3:19	298	3:20	296	3:19
AVERAGE	*306*	*0:51*	*305*	*0:50*	*304*	*0:51*	*308*	*0:50*	*306*	*0:51*

Annex 3 – Table with the peak memory and compilation time for the OSHEComp v1.
The table presents the measures for the compilation of the applications. Each application was compiled with OSHEComp v1 four times, and the average was taken in the last column.

Application	*Mem.*	*Time*	*Mem.*	*Time*	*Mem.*	*Time*	*Mem.*	*Time*	*AVERAGE Mem.*	*AVERAGE Time*
Billing.oml	379	0:42	359	0:42	359	0:42	359	0:42	364	0:42
ECHO_UI_v1908.oml	620	2:21	620	2:20	619	2:21	619	2:21	620	2:21
EMS_Assets.oml	153	0:14	189	0:14	155	0:14	193	0:14	173	0:14
EMS_Customers.oml	276	0:25	276	0:25	283	0:25	218	0:25	263	0:25
EMS_ProdProfile.oml	128	0:14	128	0:14	126	0:14	125	0:14	127	0:14
EMS_PV_v286.oml	601	0:56	597	0:56	593	0:56	601	0:56	598	0:56
EMS_Tennet.oml	161	0:12	161	0:12	161	0:12	161	0:12	161	0:12
EnergyMeasures.oml	219	0:16	222	0:16	215	0:16	217	0:16	218	0:16
EnterpriseManager.oml	218	0:17	168	0:17	166	0:17	167	0:16	180	0:17
IssueManager.oml	144	0:18	139	0:18	146	0:18	145	0:18	144	0:18
issues.oml	475	0:47	424	0:46	442	0:45	296	0:46	409	0:46
regressiontool.oml	202	0:26	200	0:26	202	0:26	204	0:26	202	0:26
reg_dashboard40.oml	337	0:28	337	0:28	336	0:28	336	0:28	337	0:28
ServiceCenter.oml	322	0:43	310	0:42	310	0:42	310	0:43	313	0:42
SLSRetail.oml	307	3:31	310	3:31	310	3:30	310	3:32	309	3:31
AVERAGE	*303*	*0:47*	*296*	*0:47*	*295*	*0:47*	*284*	*0:47*	*294*	*0:47*

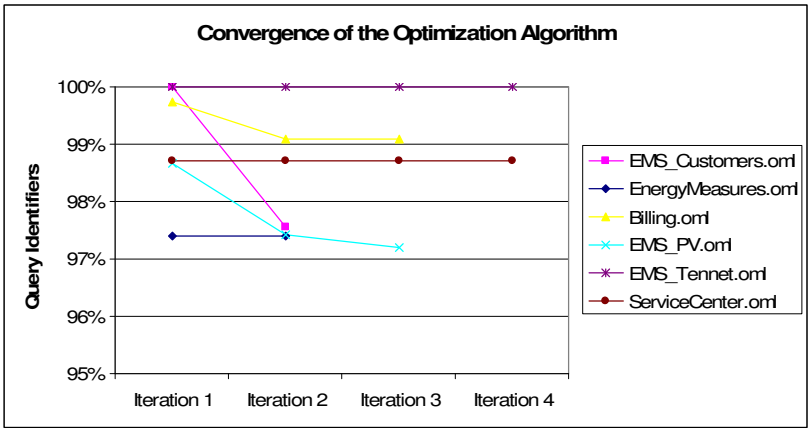
Annex 4 – Table with the peak memory and compilation time for the OSHEComp v2.
The table presents the measures for the compilation of the applications. Each application was compiled with OSHEComp v2 four times, and the average was taken in the last column.

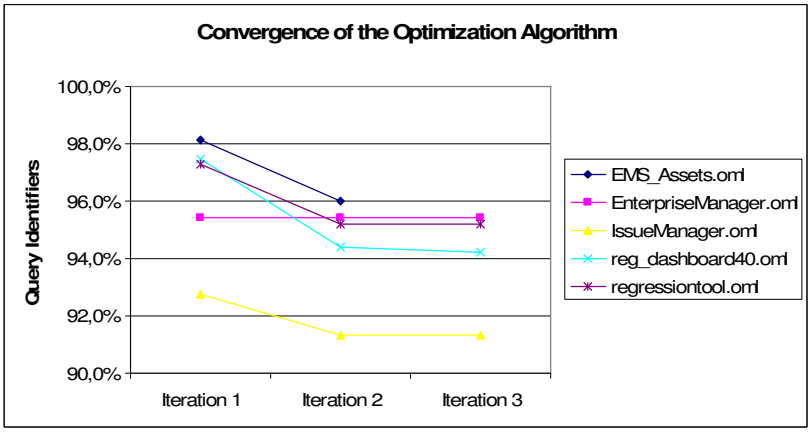
Application	*Mem.*	*Time*	*Mem.*	*Time*	*Mem.*	*Time*	*Mem.*	*Time*	*AVERAGE Mem.*	*AVERAGE Time*
Billing.oml	357	0:42	357	0:44	357	0:42	357	0:41	347	1:24
ECHO_UI_v1908.oml	615	2:19	619	2:19	409	2:16	579	2:18	556	2:18
EMS_Assets.oml	192	0:14	133	0:14	195	0:14	130	0:14	163	0:14
EMS_Customers.oml	286	0:25	232	0:24	285	0:25	198	0:25	250	0:25
EMS_ProdProfile.oml	128	0:13	130	0:14	159	0:13	128	0:13	136	0:13
EMS_PV_v286.oml	596	0:54	605	0:56	601	0:55	590	0:55	598	0:55
EMS_Tennet.oml	148	0:11	163	0:12	161	0:12	151	0:11	156	0:11
EnergyMeasures.oml	219	0:16	219	0:16	218	0:16	219	0:16	219	0:16
EnterpriseManager.oml	226	0:16	171	0:16	180	0:16	180	0:16	189	0:16
IssueManager.oml	153	0:18	147	0:18	148	0:18	149	0:18	149	0:18
issues.oml	392	0:46	425	0:45	467	0:45	437	0:45	430	0:45
regressiontool.oml	202	0:26	202	0:26	207	0:25	207	0:26	205	0:26
reg_dashboard40.oml	338	0:28	337	0:28	334	0:28	336	0:28	336	0:28
ServiceCenter.oml	312	0:42	326	0:42	325	0:42	335	0:41	325	0:42
SLSRetail.oml	316	3:31	316	3:31	316	3:30	316	3:30	316	3:30
AVERAGE	*295*	*0:47*	*292*	*0:47*	*291*	*0:46*	*287*	*0:46*	*292*	*0:49*

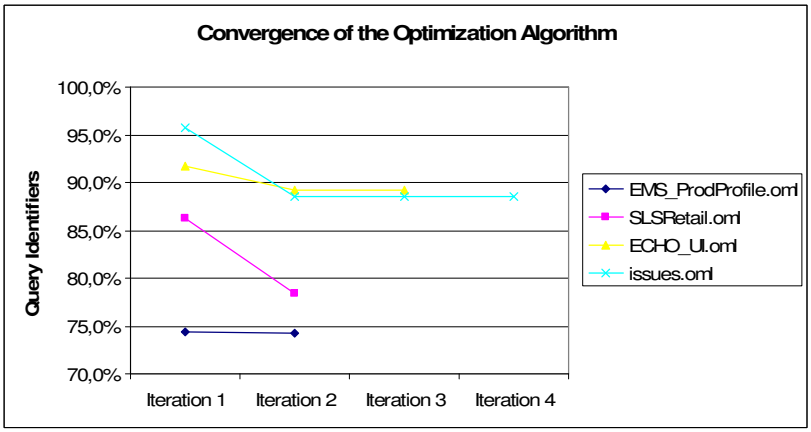
Annex 5 – Table with the peak memory and compilation time for the OSHEComp v3.
The table presents the measures for the compilation of the applications. Each application was compiled with OSHEComp v3 four times, and the average was taken in the last column.

Application	*Static Gain in Query Optimizations*	*Loss in Optimization Time*
Billing.oml	0,92%	88%
ECHO_UI.oml	10,74%	114%
EMS_Assets.oml	3,99%	70%
EMS_Customers.oml	2,44%	70%
EMS_ProdProfile.oml	25,75%	233%
EMS_PV.oml	2,81%	86%
EMS_Tennet.oml	0,00%	195%
EnergyMeasures.oml	2,59%	68%
EnterpriseManager.oml	4,59%	59%
IssueManager.oml	8,66%	12%
issues.oml	11,44%	176%
regressiontool.oml	4,78%	112%
reg_dashboard40.oml	5,78%	140%
ServiceCenter.oml	1,28%	119%
SLSRetail.oml	21,61%	60%
AVERAGE	7,16%	105%

Annex 6 – Table with the main results.
The table shows the gains in query optimizations, and their side-effects exposed as an increase in the optimization time.







Annex 7 – Charts of the convergence of the applications.
The three charts shows the convergence of the optimization of each application. The data was split into three graphs for the sake of legibility. The used query identifiers are plotted relatively to its value without inter-procedural optimizations, which corresponds to the 100% in the graphs.