INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

OutSystems Compiler Optimizations

Leonardo Monteiro Fernandes

Resumo alargado 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

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.

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 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 has its pros and cons.
- 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, by 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.

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.
In chapter 2, it is formally defined the problem of the inter-procedural optimizations in the OutSystems compiler. We present a solution that allows inter-procedural optimizations in OutSystems compiler. Finally, the measured results of the implementation are shown.
Chapter 3 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 4.

2 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.

2.1 Objective of the OutSystems Compiler

In previous sections, we 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.

2.2 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 1 – **Program ℙ.** A program $\mathbb{P}$ is a set of procedures, $\mathbb{P} = \{P_1, P_2, \ldots, P_N\}$.
>
> Definition 2 – **Variables of a program 𝕍.** By $\mathbb{V}(\mathbb{P})$ we denote the set of all variables of a program $\mathbb{P}$.
>
> Definition 3 – **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 4 – **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 5 – **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 6 – **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 7 – **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 8 – **Local usage predicate.** For a given procedure P, we define a function called local usage predicate φ_P: var(P)→{`true`, `false`}, defined as to have $\varphi_P(v)$ = `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 9 – **Inter-procedural usage predicate.** For the program $\mathbb{P}$, we define a function called inter-procedural usage Φ: $\mathbb{V}(\mathbb{P})\times\mathbb{P}$→{`true`, `false`}, defined as follows: for every two procedures P, Q $\in \mathbb{P}$, and a variable $v \in$ 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 10 – **Total inter-procedural usage predicate.** For the program $\mathbb{P}$, we define a function called total inter-procedural usage $\overline{\Phi}$: $\mathbb{V}(\mathbb{P})$→{`true`, `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 the literature. 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 Φ̄
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 $\mathbb{P}$

Outputs: set of used variables of $\mathbb{P}$

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       φ_P^(i) ← procedureOptimize(P, Φ̄^(i))
L9       for each v in in(P) do:
L10        Φ^(i)(v, P) ← φ_P^(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) ← φ_P^(i)(v)
L14  output {v ∈ 𝕍(ℙ) | φ_P^(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 $\mathbb{P}$ 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.

2.2.1 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 11 – **Inter-procedural chain.** We say that there's an inter-procedural chain from procedure P to procedure Q, with P ≠ 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 12 – **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 → 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 → Q if one of P → Q or Q → 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.

2.3 Results

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.

The new algorithm suffers from a 7% of increase in memory requirements. The compilation time was also risen by 5 seconds, which is 10% of the average compilation time.

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%.

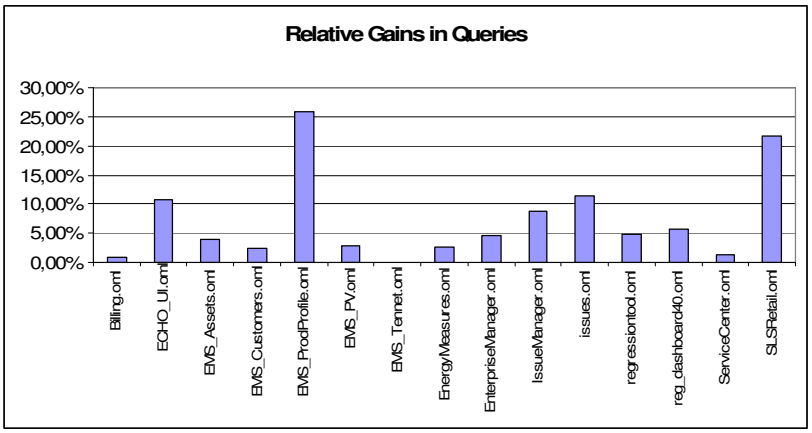

Figure 1 – Comparison of the gains in query identifiers between the inter-procedural optimizer and the original algorithm.
The figure shows the relative gains of the tested applications, when a comparison is made between the original compiler, and the inter-procedural compiler.

The chart in Figure 2 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.

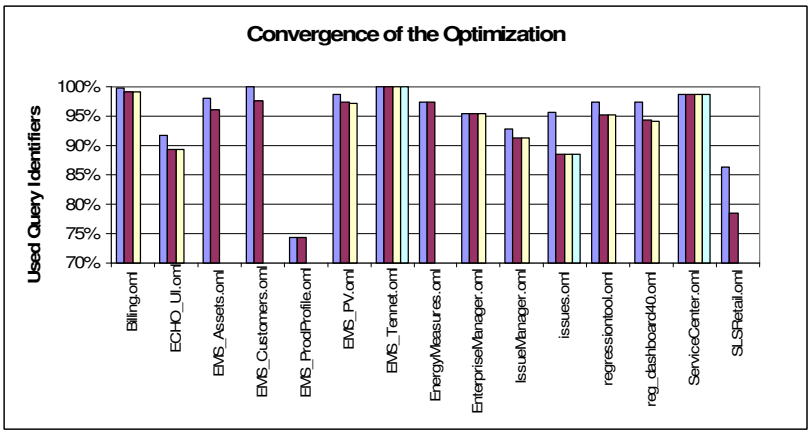

Figure 2 – Normalized convergence of the individual applications for the inter-procedural optimizer.
The chart presents the iterations of the applications, and the percentage of used query identifiers in each iteration. The reference value, 100%, is the result which would be obtained by the original algorithm.

3 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. 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.

We could also perform better optimizations if we specialize the procedures for some frequent usages, following an approach similar to procedure inlining. 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.

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.

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.

4 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 70000 lines of C# code, its modularity is becoming a concern.

The 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.

5 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.