SolidOpt – A Multi-Model Software Optimization Framework

Vassil Vassilev, Alexander Penev, Martin Vassilev

**Faculty of Mathematics and Informatics, University of Plovdiv "Paisii Hilendarski"
Plovdiv, Bulgaria**

Abstract

This paper presents a framework, SolidOpt, which helps the automated and dynamic lifelong optimization of software systems. Usually, optimizations are a virtue of the (optimizing) compilers. We suggest moving out the optimization facilities and making them more accessible even to end-users during the entire program life cycle. In order to achieve better results, SolidOpt provides multiple representations and flow graphs. The work presents some of the main ideas and principles of the optimization framework and the advantages of using multiple representations. We emphasize on the significance of the environment and how it influences the optimal execution of the computer programs. We examine a "continuous optimization" approach, which considers program's environment and perform domain-specific optimizations. We illustrate the concept of these advanced optimizations with examples.

Keywords: *Software Optimization, Multi-model Architecture, Software analysis, IL/bytecode Engineering.*

1. Introduction

It is challenging to achieve system execution optimum in some respect, performance for example, especially when developing large software systems. There are techniques for improving the system optimality in different aspects and at different time of its life cycle. Almost every technique needs an interactive intervention by system developers to introduce the improvements. There are many factors, which influence the system optimality. Few software applications try to address some issues in this field. However, most of them are concentrated in only certain stages of the development life cycle of the software application.

There are many frameworks, which contain elements for software system analysis. In addition, there are frameworks, which contain elements for software system transformation (especially optimization). A good combination between both kinds would enable the application to perform focused, automated code transformations. It would give a mechanism for better optimality control during entire system life cycle.

Informally, software optimization is a performance improvement of the target application (or more generally, the use of less resources of the computer system). Two major approaches can lead to a performance increase:

- Hardware – it consists of replacing the hardware of the computer system with more productive components. It has very clear disadvantages;
- Software – it consists of creation of more optimal computer programs or tuning the performance of the existing ones.

Our focus is on the software optimization. There are two kinds of software optimization, depending on the way of applying them:

- Manual – it is a very labor-intensive task. It includes collecting information of how the system works. Continuous tests for efficiency and correctness of the changes should be made. Most of these kinds of optimizations are in one criterion. One of the reasons is that it is difficult to consider so many parameters, to seek an eventual correlation between them and to test and evaluate the accomplished result;
- Automatic – it includes applying diverse optimization methods on the target program. The methods suppose preliminary proof for semantic equivalency of the transformed program (like an optimizing compiler, for instance).

In some cases, even the optimizing compilers do not produce always-optimal executable code. When compiling large systems, the production of suboptimal code affects negatively system's overall performance. The optimization modules in the compilers, including the Just-In-Time (JIT) compilers, analyze different representations of the source code and apply a set of optimizations. Moreover, application of the optimization methods in JIT compilers is limited by time, because they kick in only at application start up. Some of the limitations of the built-in optimizations in the translators are their:

- Static nature – they are applied once at compile time;
- General nature – specific information for the domain is not used;
- Non-extensible nature – they are an essential part of the translator and they cannot be easily extended by a third-party developer.

A framework delivering synergy between analyses, profiling and optimization of applications during their entire life cycle of software systems would minimize the impact of the described issues. Our main goal is to develop a framework, which contains utilities for lifelong analysis and automated transformation of software systems. The framework enables the development of:

- Self-optimizing software systems – systems, which incorporate optimization methods and apply them on themselves (their design and implementation is done with the knowledge of the framework existence);
- Tools – they can be used for optimizing during system's entire life cycle.

SolidOpt supposes several roles of use:

- A developer role – the developer uses the framework during programming to achieve optimality (with exclusive knowledge of the framework and the program model);
- An integrator role – the integrator uses tools based on the framework to optimize implemented programs (the integrator does not have knowledge of the program model and the concept of the program);
- An end user role – the end user uses a program built on top of the framework or tools, based on the framework and makes extra settings on the optimizing profiles. Statistics during the execution can be used for extra personalization of the optimization techniques.

Transformation methods of the framework use the information, obtained from the analysis tools (static or dynamic) to perform automated transformations. The goal of SolidOpt is to provide building blocks to assemble optimization tools. They should be able to optimize .NET/Mono assemblies given only their binaries. The framework should be able to provide multiple representations (models) of the code and optimizations. They should be implemented in an open, general, loosely coupled and extensible way.

This paper is divided into sections as follows: Section 2, Related Work, describes the related work in the domain; Section 3, Architecture, introduces a common architecture and argues about its usability in a few different cases; Section 4, Implementation, gives a brief overview of the implemented code models, flow models, optimization methods and tools; Section 5, Advanced Optimizations, presents a few research results in the area of continuous self-optimization and program adaptability; Section 6, Conclusion, summarizes the work and gives future perspectives.

2. Related Work

Some of the most often used tools and frameworks for analyzing software systems are FxCop [1], StyleCop [2], Gendarme [3], FindBugs [4], PMD [5], Bandera [6], JLint [7][8], ESC/Java [9]. More detailed information and comparison between some of them is given in [10].

Gendarme is an extensible rule-based tool for finding problems in .NET applications and libraries. Gendarme inspects programs and libraries that contain code in CIL [23] format (Mono and .NET). It looks for common problems with the code, problems that a compiler does not typically check or have not historically checked.

FindBugs is a bug pattern detector similar to Gandrame. This tool analyzes Java bytecode and performs syntactic checks and data flow analysis on program source code. FindBugs uses a series of ad hoc techniques designed to balance precision, efficiency and usability. One of the main techniques FindBugs uses is to match syntactically source code to known suspicious programming practices.

Bandera uses a completely different technique. In order to use Bandera, the programmers annotate their source code with specifications describing what should be checked, or no specifications if the programmer only wants to verify some standard synchronization properties. Bandera checks for deadlocks if annotations are not present. Bandera includes optional slicing and abstraction phases, followed by model checking.

ATOM (Analysis Tools with OM) [11] is a single framework for building a wide range of customized program analysis tools. The user simply defines the tool-specific details in instrumentation and analysis routines. Building a basic block counting tool with ATOM requires only a page of code. Vulcan [12] extends the main ideas of ATOM. It performs both static and dynamic code modifications in heterogeneous and distributed software systems.

BARBER (Binary Refactoring browser for Java) [13] is a tool for bytecode transformation such as Split Class, Glue Class, Inline/Devirtualize Method, Remove Delegate and Remove Visitor. Soot is a framework for optimizing Java bytecode [14]. The framework is implemented in Java and supports three intermediate representations for representing Java bytecode: Baf, Jimple, and Grimp.

The LLVM project [15] aims to provide building blocks for lifelong optimization. It relies on a well-defined intermediate representation called bitcode or LLVM IR. LLVM IR is in static single assignment (SSA) form, which is very suitable for optimizations. Many of the ideas are fundamental but they are oriented in building compilers, which constrain the use of the framework. It does not provide more than one view of the program and the optimizations are constrained by the low level LLVM IR.

Above-described systems do not support unified mechanism for analysis, profiling and optimizing wide range of arbitrary applications. Some of them are suitable for only one programming language. Others are concentrated only on analysis or transformation of the system. Third, cover only part of the entire system life cycle (mostly the development time).

3. Architecture

Programming languages are notations for describing computations to people and to machines [22]. Software can be treated as a prescription, solving concrete real-world problem. Prescriptions or algorithms are models, which describe part of the real world. Usually, that software program is a (execution) model for solving a given real-world problem.

With time the programming languages should get more and more abstract and closer to the natural language, i.e. to turn into a model, clearer to the developer. This opens a semantic gap between the low-level models, which are executable by concrete virtual execution systems (VES). The gap is filled by a complex system reducing the high-level model to a low-level one. The translation process leads to loss of information about the high-level model and often to a creation of suboptimal executable code. The suboptimal translation is due to three main reasons:

- Missing or not very well implemented optimization modules in the translator;
- Not using the full capabilities of the VES;
- Not using the whole information, which is available in the high-level model.

A solution of these disadvantages is to bring the optimization modules and algorithms outside of the translators. This grants a better flexibility and independence, because a third-party developer can extend the optimization methods. In addition, this allows the developers to add extra knowledge of the application domain and perform more aggressive optimization strategies. These strategies may not be valid in the common case, because they are specific for the domain or the concrete application.

During translation to an executable model, the translators fill the semantic gap by constructing internally multiple program models. These help the translation by reducing the difference in the levels of abstraction of the input and output models. Unfortunately, many transformations between models often remain unclear and often irreversible.

Our framework, SolidOpt, uses multiple models with clear and observable mechanisms for transformation, which allows more accurate translation (transformation) and optimization of the computer programs. Figure 1 proposes a multi-model architecture and generalizes the transformation of the high-level software model into an executable. Depending on the goals, the transformation can work at different models.

In terms of SolidOpt, a transformation optimization method (shortly called an optimization method or an optimization) is a module, which transforms the software program. This transformation should satisfy given conditions. Often, these conditions are in the form of metrics, which estimate system quality. The metrics evaluate properties such as efficiency, energy consumption and memory footprint. Some optimization methods can use models of different levels of abstraction because of:

- More efficient operation of the method – there are situations when an optimization can be applied at several levels of abstractions. However, on each level the optimization efficiency can vary. Therefore, it is better to choose a model, upon which the optimization has best result when applied;
- Easier implementation of the method – for example, finding dead code is much easier using a control flow graph model (the problem is reduced to graph connectivity) than at source code level.

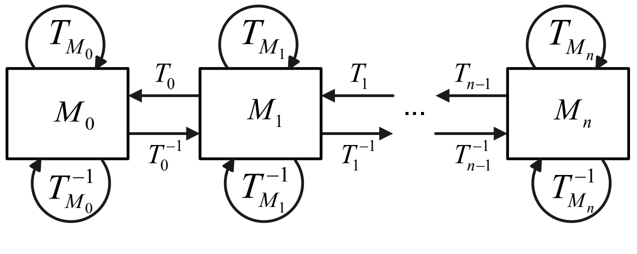
$$\begin{tikzcd}
M_0 \arrow[loop above, "T_{M_0}"] \arrow[loop below, "T_{M_0}^{-1}"] \arrow[r, shift right, "T_0^{-1}"'] & M_1 \arrow[l, shift right, "T_0"'] \arrow[loop above, "T_{M_1}"] \arrow[loop below, "T_{M_1}^{-1}"] \arrow[r, shift right, "T_1^{-1}"'] & \cdots \arrow[l, shift right, "T_1"'] \arrow[r, shift right, "T_{n-1}^{-1}"'] & M_n \arrow[l, shift right, "T_{n-1}"'] \arrow[loop above, "T_{M_n}"] \arrow[loop below, "T_{M_n}^{-1}"]
\end{tikzcd}$$

Fig. 1 Multiple Model Interaction.

Some methods require a concrete model. For instance, the machine-dependent optimizations are made on low level of abstraction (execution level). The implementation of a method replacing multiplication by a power of two with a left shift illustrates a machine-dependent optimization. A key objective of SolidOpt is to give enough models for applying optimization transformations.

Figure 1 demonstrates the relationship between models and transformations. More concretely, it has the following annotations:

- M_i – level of model abstraction, where $i \in [0;n]$. M_0 is the target model, i.e. the machine executable;
- $T_{M_i}, T_{M_i}^{-1}$ – model transformation, where $i \in [0;n]$. The level of abstraction is preserved;
- T_i, T_i^{-1} – trans-model transformation, where $i \in [0;n-1]$. The level of abstraction is changed.

The scheme describes the relationship between models and how to lower or raise the level of abstraction. In addition, it shows the mechanism for transforming models. Moreover, there may be more than one model at the same level of abstraction. We call them models with same level of abstraction. For example, lets assume the model M_0 is designed for a CISC processor. It is clear there can be a model M'_0, that is designed for a RISC processor. We say that M'_0 and M_0 of equivalent level of abstraction.

Forward direction (lowering the abstraction) is known as compilation (denoted it with $T_{M_i}, i \in [0;n-1]$). Increasing the level of abstraction (opposite direction) is known as decompilation (denoted with $T_{M_i}^{-1}, i \in [0;n-1]$). An important part of the discussion is the existence and acceptability of these transformations.

The existence of a generic transformation function between models is not guaranteed. Sometimes using an approximation of the model-transforming function is admissible. For instance, the generation of suboptimal code by the compiler may be partially a consequence of a trade-off with this "satisfying" approximation. In this particular example, the approximation satisfiability is only because the compiler must generate semantically equivalent models. Conversely, that satisfying approximation may be unsatisfying in terms of optimal execution. Models in SolidOpt's model hierarchy are selected to have adequate transformation functions. One of the main goals of the framework is to change the abstraction level of models with no information loss. When there is no way to avoid loss of information, the loss should be minimal.

Let us consider the transformations $T_{M_i}, i \in [0;n]$ and $T_{M_i}^{-1}, i \in [0;n]$. In essence, they are the transformations of a model into the same model by improving it in some aspect. $T_{M_i}^{-1}$ aims to reverse the transformation made by T_{M_i}. In contrast to compilation and decompilation, the existence of such a reverse function is not guaranteed. However, if there was such a function, it could be used for easier manipulation of the model. The functions T_{M_i} are important for the concept of the framework. They are used to change the model, making it better in some aspect. The idea can be described with first order logic. We can define a predicate, which indicates whether the new model is better from some perspective. Let us define the predicate $P(m,m') = \mu(m') < \mu(m)$, which $\mu(x)$ is a function determining the size of the program model $x \in M_i$, $i \in [0;n]$. Each transformation, which satisfies P, improves the model in terms of achieving a minimum amount of programming code.

Let us consider the following scheme:

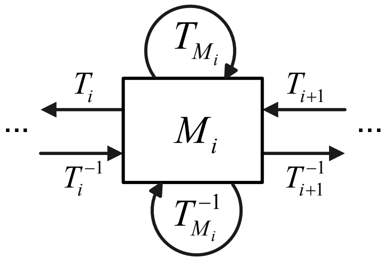

Fig. 2 Refactoring/Obfuscation.

The scheme (Figure 2) represents the ith ($i \in [0;n]$) element of Figure 1. Lets assume that M_i is the source code of the program. The proposed scheme works well for transformation processes such as source code refactoring [16]. If one defines a predicate P, that satisfies the definition of refactoring and apply transformation T_{M_i} and the transformation satisfies the predicate, we can say that we did refactoring on the code. In general, we say that we did refactoring on the model M_i. If $T_{M_i}^{-1}$ exists, we could easily reverse the model without additional overhead into prior to T_{M_i} state, i.e. the applied transformation is reversible. Quite naturally, in this scheme can be fit another well-known technique for transformation of the code – code obfuscation [17]. The technique uses code transformation for other purpose – to make the reverse engineering harder.

The methodology allows users to build optimization methods, operating at different levels. It provides flexibility in the development of sophisticated optimizations such as merging of classes. In order to achieve better results, some optimizations may need to work on several abstraction levels of the model.

Using multi-model architecture decreases the productivity of the system. In addition, it increases the overall complexity of the framework. However, the purpose of our framework is to provide a mechanism for optimizing software applications and lower performance is acceptable for the prototype. Optimizations compensate the lower productivity of the framework: it is more important to achieve greater efficiency of the target system. We have ideas how to introduce self-optimizations to reduce SolidOpt's lower performance effect if it becomes a bottleneck.

4. Implementation

The modularity of the concepts and the framework itself allowed us to work on many different parts in parallel. We can categorize the developments into Flow Models, Code Models, Optimizations and Tools. They are briefly presented in the following sections.

4.1 Flow Models

Control Flow Graph: A control flow graph (CFG) is a graph representing the execution flow [19]. Every graph node contains instructions grouped in basic blocks. Every basic block contains only linear instructions, i.e. instructions that do not change the control flow and that are executed in a row – one after another. There is a branch or a return instruction at the end of each basic block and the next instruction starts a new basic block. The edges of the built graph model are all possible branches between the basic blocks. There are two types of branches in SolidOpt:

- Structural – i.e. branches caused by the "normal" possible changes in the control flow of the program;
- Exceptional – i.e. branches caused by the exception handling in the control flow of the program.

The implementation of CFG in SolidOpt can be divided in two steps: creating basic blocks and connecting them. Each instruction is parsed and checked if it reflects certain conditions. New basic blocks are created if the current instruction is: the first instruction in the method body; an exception handler start instruction; a terminator instruction (instruction for branch, return, break, exception); has an operand – an address pointing to other instruction. After method body is split into basic blocks, they are connected accordingly. Analyzing the last instruction of each basic block produces the connections between the blocks. The implementation of CFG in SolidOpt provides a bidirectional connection between each basic block, i.e. every block knows about the blocks the control flow may jump to (called successors) and the blocks it is pointed to (predecessors).

Call Graph: A call graph (CG) is implemented and it models the connections between method calls in a method body. The call graph contains nodes and edges constructed by the following rules: the analyzed function creates the root node; a function or a method call generates a node; then the nodes are connected to their caller nodes.

4.2 Code Models

Three Address Code: Three-address code (TAC) is an intermediate code representation where each statement contains at most one operator on the right side of an instruction [19]. The TAC is a sequence of instruction in the form of `A = B op C` where `A`, `B` and `C` are identifiers while "`op`" stands for operator [22]. The three address instructions are based on two concepts – addresses and instructions. The addresses can be names; constants; or temporaries. The instructions can be: assignment instructions; copy instructions; unconditional jumps; conditional jumps; procedure calls; return instructions; array manipulation instructions; address and pointer instructions; type casts; etc. TAC instructions are executed in numerical sequence unless forced otherwise by a conditional or unconditional jump. SolidOpt uses a simulation stack to turn the stack-based CIL into TAC.

Abstract Syntax Tree: The Abstract syntax tree (AST) represents the source code as a hierarchical syntax structure [22] in the form of a graph. A node represents each operator in an expression and each operand creates a child. It is true that an AST can be generated not just from an expression but also from any construct and it introduces more formal representation by omitting certain details from the source code model. The AST is a suitable representation for outlining the priorities of the operators in an expression. Currently, SolidOpt supports TAC to AST transformations.

4.3 Optimizations

The optimizations are a key ingredient in the framework but they are only the tip of the iceberg. In order for an optimization to be effective, it needs a lot of other infrastructure. An optimization should work on a well-defined model. It should use the model, which makes the optimization most efficient. The optimization should guarantee correctness. In the cases when it is domain-specific, it should yield correctness boundaries and expected errors. These concerns usually require the optimization to be split into two parts: analysis part and transformation part. The analysis part is responsible for checking the feasibility and the impact of the optimization. The transformation part changes the model to actually apply it. Due to the limited manpower, we have focused more on the implementation of the transformations parts of the optimizations. Usually the driving part is done using annotation attributes available in CLR and thus CLR-based programming languages such as C#. As a proof of concept, we have implemented a few well-known optimizations working on variety of models:

- Method inline – we implemented two versions: one using the AST level and one using the CIL level. We ran into different issues on both models and thus we cannot say on which level the implementation was easier. The performance improvement effects are slightly in favor of the CIL level, because there are less compilation steps and the control of what is generated is better;
- Constant folding and propagation – it was trivial to implement on AST level, because it consists of node visitation and replacement;
- Dead code elimination – it was trivial to implement using a CFG on CIL level;
- Overflow arithmetic removal – CIL offers such target-dependent information explicitly. Its implementation iterates over the instructions in the method body and replaces overflow-checking instructions by their non-checking versions.

4.4 Tools

SolidReflector is a plugin-based tool developed in the context of SolidOpt [19]. SolidReflector uses SolidOpt as a library, in order to build the multiple models of a .NET assembly. A key goal of the tool is to make the multistage compilation a little more comprehensive. Once the model is built, it is extended with a graphical visualization, showing implicit information such as nodes and edges in the flow graphs, for example. Visual and non-visual code models are bound together to create a hybrid graph of models. It allows the models to be changed in flight. Introduced changes to the model can be lowered to an executable and run in simulation mode in a secure environment. For instance, the nodes of method's control flow graph can be visually "rewired" using SolidReflector's graphical user interface. The change can alter the semantics not only of the method itself, but the entire program. After a change in a model by the user, the modifications have to be compiled again into an assembly, i.e. into executable code.

5. Advanced Optimizations

Software applications can be considered as a specimen, from which optimal-working systems are generated. The original source code remains unchanged and a subsystem takes responsibility to generate an optimal incarnation. An important advantage of the approach is that conceptually different software transformations depending on the objectives can be applied. Additionally, this reduces the optimization-specific code in the actual program logic.

The existence of a specialized software subsystem allows the division of the specimen, a blueprint produced by a developer, and a model (for execution), produced by the translator. Hence, in order to get better performance it is not necessary to change the blueprint. In our point of view, this is a key strength, because changing the source code only for performance tuning makes it hard to read, understand and maintain. In order this program specialization to happen, many factors come into play, such as user-system interaction and system's surroundings.

5.1 Operational Environment

Software systems do not live in full independence. Usually, they depend on many external factors such as the operating system, for instance. Operating systems isolate the application into processes. Software systems should be considered as a combination of a process and its operational environment. The operational environment can constrain the process or change its behavior and may be influential to its performance. The processes exist in this environment and depend on its nature. The environment consists of three main elements:

- Inter-process communication – communication between two separate processes is done through data exchange. The communication is two types – direct and indirect. Direct communication happens by sending messages (requests) and receives results. Indirect approach is in communication with the process through a third process (mediator). The type of data and communications determine a great part of the process behavior;
- Hardware – the characteristics of the hardware components define many functional features of the process. In the end, the hardware determines the overall productivity;
- User interaction – the data exchange with the application is usually mediated by a user interface. The user defines user-specific control flow and data flow of the application. The user-specific control flow is a sequence of operations, requested by the user. The user-specific data flow is the sequence of exchanged data with the process. Both, user-specific control and data flow define a user's behavior with an application.

The information about application's operational environment, including user's behavior, determines important factors about the optimal execution. This information varies in the different stages of application's life cycle. It can be qualitatively and quantitatively different in development time, deployment time and production time. This is one of the reasons that we consider optimization as a continuous process spanning throughout entire optimization lifetime. Optimization in some parts of the life cycle can be even of a negative effect overall. All optimization strategies should take into account the collected usage information. Some strategies include "adaptation" to the operational environment, some to the user behavior and some to both. The following example illustrates the idea:

```
if (x == 1) {...}
else if (x == 2) {...}
...
else if (x == 30) {...}
```

From the collected runtime data the value that most often gets x is 30 and we can order the rest by their statistical expectations. Then one can apply the transformation of the model reshuffling the order of the statements into:

```
if (x == 30) {...}
else if (x == 2) {...}
...
else if (x == 1) {...}
```

Thus, the program is going to do (most often) 29 comparisons less, which increases its productivity while keeping its semantics. The transformation changes the program control flow based on the user's data flow.

The behavioral and environmental data broadens the optimization horizon. However, the proper interpretation of the information rather than information itself is more important. Incorrectly interpreted reliable information about the system can lead to much greater trouble than the opposite (if we interpreted the information correctly we could determine whether the information is incorrect). There are several prominent data analysis techniques, which can help with the data interpretation:

- Simple mathematical models and heuristic methods – one can use data fitting in order to find admissible heuristics and construct a feasible mathematical model on top of them. This gives a relatively simple but powerful mechanism for data interpretation and management. Often finding such a "good" function is not an easy task. When specificity of the data is changed, it is necessary to restart the modeling process. Mathematical models and heuristics strongly depend on the problem area. For example, it is easy to find a heuristic for the application size, while it is hard to find such heuristic for a system performance or for a system power saving plan [18]. The human factor is very important in the process;
- Statistical methods – there are many tools and libraries, which perform data analysis [19]. They build sophisticated statistical models, which allow drawing statistics-based predictions. They can be used to form better heuristic functions;
- Expert systems – using them gives flexibility in the predictions. Expert systems can be applied in many cases. They can control the optimization modules, based on the knowledge of experts. They are able to handle large amounts of data automatically. They can be used in combination with statistical methods to perform precise control over the optimization of computer programs;
- Intelligent Systems – the next level is to use fully automated systems that control the entire process of optimization. This has advantages and disadvantages, which are beyond the scope of this paper.

Another example for model adaptation according to the operational environment is the architectural changes of the hardware. For instance, the creation of a productive application run on a VES with no cache memory (a micro controller) suggests a specific transformation of the model. Adding cache memory to the architecture of the VES (for instance an Intel CPU) is an architectural change. On the new architecture, the application may not achieve the expected optimal execution. Considering the operational environment makes optimizations much more effective.

5.2 Continuous Self-Optimization

SolidOpt can be used to create self-optimizing software systems. In addition, they may collect behavioral information and information about their operational environment. We consider such a scenario as a software evolution. The framework provides the building blocks, which can be used to evolve the model's optimality. The evolution of model optimality is usually in a specific direction, e.g. improving of a metric (such as the size of the program, occupied memory, performance, and power saving). We summarize the self-optimization by introducing an architectural pattern shown on Figure 3.

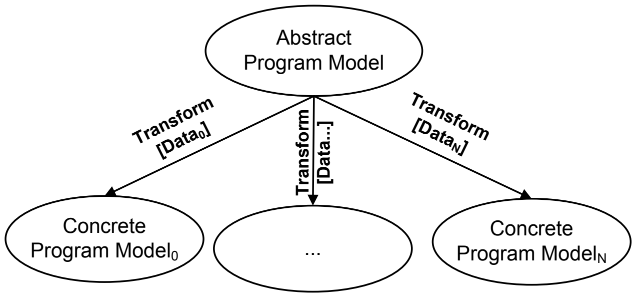

Fig. 3 General Application Specialization.

Based on the collected information from the interaction with the operational environment and program's internal behavior, an optimization strategy for the model can be selected. The optimization of the application model creates essentially a new executable with improved performance with respect to the collected data. Since the data includes user-application interaction, the multi-user applications will have multiple datasets. Thus, either the optimizer will have to produce a user-specific optimized copy or it should find a good trade-off.

For each iteration, the collected data will be about an application different from the initial. There are two major scenarios for optimizations, which rely on the collected data. They could be applied either to the initial application or to the last produced optimal version. The latter case should be better since it is supposed to converge much quicker. Figure 4, illustrates the evolution in optimization. The evolutionary pattern describes an endless optimization loop, based on the relationship between the abstract program model (APM), the concrete program model (CPM) and their transformations.

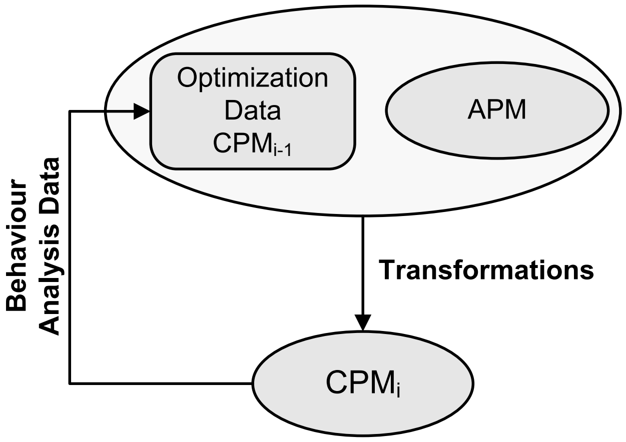

Fig. 4 Application Transformation Evolution.

The initial model serves as a template to construct an optimal concrete program model. The new program model and its execution enable to collect new data for the next cycle of evolution. The new program model can depend on:

- The old model (CPM$_{i-1}$) – helps reducing the time to adapt (achieving system optimal operation);
- The abstract (initial) model – provides a way in case the evolution of a specific model is impossible or the data does not fit well the abstract model can be used to restart the procedure;
- The data, collected from the operational environment – ensures precise tuning of the optimizations.

The length of an evolution period strictly depends on the specifics of the application. In addition, it depends on the time needed to collect sufficient information about interaction with the operational environment. The periods can be of various intervals of time.

Model adaptability is the ability of the software system to be a subject of an evolutionary process. Adaptation is a time-dependent process. In situations where the time for adaptation is years, it would be irrational to think that it is useful. Therefore, an important property of the adaptability is its feasibility. A decisive characteristic of systems in terms of those facts is the adaptability of the model. Highly specialized applications are more difficult for adaptation, because of the specific knowledge, incorporated during development.

5.3 Experimental Results

System configuration is essentially moving constants outside the source code such that if they are changed no recompilation will be needed [21]. However, it introduces a significant overhead because of extra method calls and reading configuration files.

Our approach to address this issue to collect statistics and perform application specialization. Moreover, in the particular case statistics may not be necessary, because big parts of the configuration files change only in exceptional circumstances. For instance, configuration parameters such as system's base folder location are changed only at installation time.

Generation of a copy with inlined configuration parameters speeds up the system dramatically. According to our tests cases (see Table 1 in Appendix) we achieve 568–35725 orders of magnitude between SolidOpt configuration assembly and INI file and 6–520 orders of magnitude between SolidOpt configuration assembly and XML file performance improvement depending on the type of configuration parameters. As a bonus, we guarantee type safety because the algorithm is able to deduct the type based on its usage throughout the program. The comparison is reading between 1 and 100 parameters from:

- INI file format configuration – we used a C# wrapper over Windows's kernel function GetPrivateProfile-String, responsible for "retrieving a string from the specified section in an initialization file";
- XML file format configuration – stored in the app.config file and accessible through application's Properties.Settings.Default namespace;
- Inlined in the program configuration – a C# assembly with constants which is JIT compiled on application's startup.

The domain-specific optimization parses the configuration file and produces an assembly. The assembly is embedded in the application, replacing all configuration file reads. Thus, a new, specialized application is generated, i.e. the original application remains unchanged and a new more efficient one is generated. Following the terminology, the original application is the APM and the specialized one – a CPM. If the configuration file changes, the process is repeated.

The improvement in productivity is evident. Making this module of production quality and integrating it in SolidOpt's mainline is planned. Most applications depend on configuration files and the optimization will have a significant impact on the performance.

6. Conclusion

We presented SolidOpt – a multi-model software optimization framework. We described its theoretical scheme and showed some of its essential implementation aspects. We showed some of its key properties such as generality, openness, extensibility, flexibility and modularity. We examined advanced software optimization techniques and proved their usability. Some of the preliminary tests and research give us enough certainty to claim that the domain is very promising and further research and development should be carried on. Especially interesting is the research and development automated methods for optimization, based on statistical evaluation of the system operability (profiling).

Important next steps to increase the robustness of the framework are the improvement of the AST and its compilation and decompilation to other models. This would enable more complex, close to source code transformations such as code refactoring. In addition, we are working on complex optimizations such as method devirtualization [20] and better support for domain-specific optimizations.

We plan to implement a TAC parser, which can build TAC triplets from a text file. It would help SolidReflector to provide a multi-model integrated development environment. In other words, users could choose to modify parts of the code or optimize it in a closer to the executable way, using the TAC form. This is also true for the rest of the models, however, they are scheduled with lower development priority.

The analysis part of the optimization methods should be enhanced, in order to achieve better automation of the optimization methods for non-experts. More user-friendly tools such as SolidReflector should be developed for more domain-specific optimizations.

Appendix

The computations, present in Table 1 (see also Figure 5), read the configuration parameters between 10^5 to 10^7 times and measure the elapsed time. Then the results are tabulated to 10^5, adjusting the elapsed time. The parameter type is only relevant for the SolidOpt case whereas the reads from XML or INI are strings by design.

Table 1: Domain-specific System Configuration Optimization.

Test Type	***Parameters***	***Elapsed Time (ms)***
SolidOpt Conf.	1 Integer	0.15
INI File	1 Integer	5359
XML File	1 Integer	78
SolidOpt	1 String	3.1
INI File	1 String	5344
XML File	1 String	78
SolidOpt Conf.	1 Integer, 1 String	17.2
INI File	1 Integer, 1 String	11984
XML File	1 Integer, 1 String	156
SolidOpt Conf.	20 String	90.7
INI File	20 String	121656
XML File	20 String	1329
SolidOpt Conf.	50 Integer, 50 String	1062
INI File	50 Integer, 50 String	603406
XML File	50 Integer, 50 String	7218

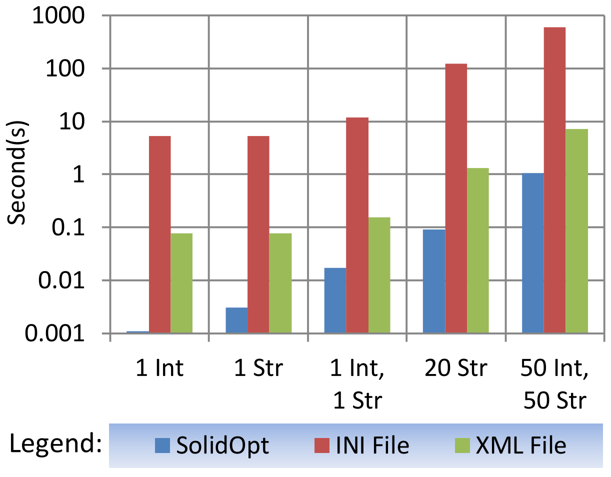

Fig. 5 Comparison of Domain-specific System Configuration Optimization.

References

[1] FxCop, http://msdn.microsoft.com/en-us/library/bb429476.aspx, (visited on 5 December 2014).
[2] StyleCop, http://stylecop.codeplex.com/, (visited on March 2015).
[3] Gendarme, http://www.mono-project.com/docs/tools+libraries/tools/gendarme/, (visited on February 2015).
[4] D. Hovemeyer, and W. Pugh, "Finding bugs is easy", SIGPLAN Notices, Vol. 39, No. 12, 2004, pp. 92-106.
[5] PMD/Java, http://pmd.sourceforge.net, (visited on February 2015).
[6] J. Corbett, et al., "Bandera: Extracting Finite-state Models from Java Source Code", in Proceedings of the 22nd International Conference on Software Engineering, 2000, pp. 439-448.
[7] C. Artho, "Finding faults in multi-threaded programs", M.S. thesis, Institute of Computer Systems, Federal Institute of Technology, Zurich/Austin, 2001.
[8] Jlint, http://artho.com/jlint, (visited on February 2015).
[9] C. Flanagan, et al., "Extended Static Checking for Java", in Proceedings of the ACM SIGPLAN'02 Conference on PLDI, 2002, pp. 234-245.
[10] N. Rutar, C. Almazan, and J. Foster, "A Comparison of Bug Finding Tools for Java", in Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering, 2004, pp. 245-256.
[11] A. Srivastava, and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools", in Proceedings of the SIGPLAN'94 Conference on PLDI, 1994, pp. 196-205.
[12] A. Srivastava, A. Edwards, and H. Vo, "Vulcan: Binary Transformation in a Distributed Environment", Microsoft Research Technical Report, 2001, MSR-TR-2001-50.
[13] E. Tilevich, and Y. Smaragdakis, "Binary refactoring: Improving code behind the scenes", in Proceedings of ICSE'05, 2005, pp. 264-273.
[14] R. Vallée-Rai, et al., "SOOT – a Java Optimization Framework", in CASCON'99, 1999, pp. 125-135.
[15] C. Lattner, and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", in Proceedings of the International Symposium on Code Generation and Optimization, 2004, pp. 75-86.
[16] M. Fowler, et al, Refactoring: Improving the Design of Existing Programs, Addison Wesley Longman Inc., 1999.
[17] D. Low, "Protecting Java Code via Code Obfuscation", ACM Crossroads, Vol. 4, No. 3, 1998, pp. 21-23.
[18] A. Varde, et al., "Comparing mathematical and heuristic approaches for scientific data analysis", ACM Artificial Intelligence for Engineering Design, Analysis and Manufacturing, Vol. 22, No. 1, 2008, pp. 53-69.
[19] V. Vassilev, M. Vassilev, and P. Petrova, "SolidReflector: A multistage, Interactive, Decompilation Framework", in International Conference From DeLC to VelSpace, 2014, pp. 49-58.
[20] K. Ishizaki, et al., "A Study of Devirtualization Techniques for a Java Just-In-Time Compiler", in Proceedings of the 15th ACM SIGPLAN Conference on OOPSLA, 2000, pp. 294-310
[21] A. Penev, D. Dimov, and D. Kralchev, "Acceleration of structured and heterogeneous configuration of the applications", in Proceedings 36th conference of Union of Bulgarian Mathematicians, 2007, pp. 327-331.
[22] A. Aho, M. Lam, R. Sethi, and J. Ullman, Compilers: Principles, Techniques, and Tools (2nd Edition), Addison Wesley Longman Publishing Co. Inc., Boston, 2006.
[23] ECMA Team, Standard ECMA-335 (6th Edition), ECMA International, Geneva, Switzerland, 2012.

Vassil Vassilev is pursuing his PhD degree in Computing at the University of Plovdiv “Paisii Hilendarski”, Bulgaria. In 2010 he completed his MSc degree in “Software Technologies” and in 2009 a BSc degree in “Informatics” at the same institution. Vassil has around 5 years work experience at CERN. Currently, his research interests are in the area of programming languages design and implementation and software optimization. He is a member of the ACM.

Alexander Penev received his PhD degree in Computing at the University of Plovdiv “Paisii Hilendarski”, Bulgaria. In 1996, he completed his MSc degree in “Mathematics – specialization Informatics” at the same institution. Alexander has over 20 years work experience at University of Plovdiv as an assistant professor. Currently, his research interests are in the area of computer graphics, programming languages design and implementation, and software optimization.

Martin Vassilev received an MSc in “Software technologies” in the University of Plovdiv in 2013. He completed his BSc in "Informatics" a year earlier. Martin has a broad set of interests in the field of software technologies. He is a Student member of the ACM.