On Source Code Optimization for Interpreted Languages using State Models

Jorge López[1,2], Natalia Kushik[2] and Nina Yevtushenko[1]

[1]Department of Information Technologies, Tomsk State University, Lenin str. 36, Tomsk, Russia
[2]SAMOVAR, Télécom SudParis, CNRS, Université Paris-Saclay, 9 rue Charles Fourier 91000, Évry, France

Keywords: Source Code Optimization, Quality of Software, State Models.

Abstract: The paper is devoted to code optimization techniques with respect to various criteria. Code optimization is well studied for compiled languages; however, interpreted languages can also benefit when using optimization approaches. We provide a work in progress of how the code optimization can be effectively performed for the applications developed with the use of interpreted languages. Methods and techniques proposed in the paper rely on the use of formal models, and in particular state models. We propose some code optimization based on two different state models, namely weighted tree automata, and extended finite automata. The problem of extraction of such models is known to be hard, and in both cases we provide some recommendations of how such models can be derived for a code in an interpreted language. All the optimization techniques proposed in the paper are followed by corresponding illustrative examples.

1 INTRODUCTION

As the complexity of information systems increases, novel techniques for software optimization are highly needed. Such optimization criteria can refer to different non-functional properties and requirements ranging from performance to some specific features, such as, for example, energy consumption for a given application under certain environment (Ivan et al., 2007).

Various optimization techniques based on the source code of an application have been well studied during the last decades. Many of these techniques have been implemented in well-known compilers, see for example (Stallman et al., 2009), and nowadays it is even possible to choose specific optimizations performed when producing the binary code.

On the other hand, interpreted languages such as PHP (Bakken et al., 2000), Perl (Wall et al., 2000), etc., have grown in popularity (especially in web applications). However, optimization techniques for compiled languages are not directly applicable for interpreted languages, given the fact that interpreted languages execute statements (one by one) as they appear in the code, i.e. there is no compilation stage of the source code as a whole. The latter therefore, motivates researchers to study methods and

techniques for optimizing applications written in interpreted languages. More precisely, the problem of interest is stated as follows: given a source code in an interpreted language, one should replace this code with another sharing the same functionality, but being 'better' with respect to some non-functional properties. In order to solve this problem we utilize appropriate formal models and propose to perform all the optimization procedures over the formal model rather than on the code itself.

In this paper, we discuss how various classes of Finite State Automata (FSA) can be applied to code optimization for interpreted languages. In particular, we study weighted tree-like automata (Fülöp and Vogler, 2009) and extended finite automata (Smith et al., 2008) as relative code representations. In the first case, we assume that each instruction of the code under optimization is augmented with appropriate weights, that can represent the normalized time needed for the instruction execution, the percentage of a disk load while it is being executed, energy consumption or any other non-functional countable feature. Correspondingly, we provide an idea of how a code can be optimized such that the total cost / weight of the automaton is (close to) minimal. For extended finite automata, we discuss how such model can be extracted from the code under investigation and which analysis can be

performed using the available flexibility. In particular, we discuss certain heuristics, to identify properties of the model that allow providing some benefit for the source code under optimization.

In fact, this short paper presents a work in progress of using state (trace) models for interpreted code optimization. Therefore, the main contribution of the paper is the discussion of how the above models can be derived and which kind of optimization techniques can be applied when using these models. Both cases of weighted and extended automata are followed by small (due to space limitation) examples of illustrative code in PHP.

The structure of the paper is as follows. Section 2 presents approaches of how the previously mentioned state models can be extracted from source code, and the optimization criteria for such models. Section 3 is divided into two parts regarding the use of weighted tree-like automata and extended automata for the code optimization. For each subsection, a discussion of how to perform the corresponding optimization is provided. Section 4 concludes the paper.

2 USING STATE MODELS FOR SOURCE CODE OPTIMIZATION

2.1 Source Code Representation

Source code is considered as a text listing of computer commands (or instructions). This text representation of source code has a formal language definition, designed to supply instructions to the computer; instructions that it must execute. We propose the notion of a state model representation of a given source code. In this paper, we use the previously mentioned state models, i.e., Extended Finite Automata (EFA) and Weighted Tree-like Automata (WTA) for code representation and optimization. In this subsection we discuss how to extract both models from the source code and how to come back to source code from each model.

The basic idea of source code extraction comes to the well-known Abstract Syntax Tree (AST) representations generated by parsing code (Jones, 2003). For interpreted languages there exist a number of open source implementations developed for such purpose, see for example (Popov, 2016). We propose to derive the appropriate state models by traversing the AST data structure that is obtained from a given source code.

Given the fact that a WTA and an AST both have a tree-like structure, the generation of a WTA is mostly straightforward. Starting from the root node of the AST and the initial state of the WTA, we traverse the AST and for each member of the current AST node structure we create a corresponding state in the WTA. The AST nodes' data structures have different members. For example, a *while loop node* has an expression (for the corresponding condition) and a list of statements (the statements to execute form the loop's body). For each of those members we add a transition with the action labeled as the object type of the node member. The successor of a current WTA state is created with respect to the corresponding AST state, and this generation process is done recursively for the successor state and current node structure member.

In order to produce source code from a WTA representation, the pre-order tree transversal algorithm can be used. With this simple algorithm, source code can be obtained from the WTA representation.

On the other hand, the generation of an EFA from an AST data structure is different. The approach also starts from the root of the AST data structure and the initial EFA state. Later on, for each expression or assignation statement we add the corresponding context variables and updating functions for the corresponding expressions. New states are created when control-flow statements are encountered (function calls, loops, conditions, etc.). A function call, however, is processed in a special manner, since the added state can be unrolled (expanded) in accordance to the proper function statements (sub-machine). When adding new states with control-flow, the control flow conditions are added to the predicates of the corresponding transitions. Intuitively, inputs and outputs are encoded (labeled) in the EFA respectively.

The process of generating source code back from an EFA representation follows the reverse procedure from the generation.

Even if the EFA structure does not appear to be close to an AST, the correspondence between the source code and the EFA is very close as it can be seen in Figure 3.

2.2 Source Code Optimization Criteria

Typical criteria for source code optimization include reducing the number of lines in the code. Since we use state models to represent the source code, this is equivalent to reduce either: i) the number of states;

ii) the number of transitions; or iii) the number of context variables in the model.

Other non-functional optimization criteria can also be considered, such as, for example, avoiding the use of costly operations or specific function calls. As an example, one might prefer using three additions of a variable instead of multiplying it by three or avoiding input/output operations in favor of in-memory operations. This will result in less expensive operations which can imply less energy consumption, for instance. Calling reliable or secure functions can be another non-functional requirement, for example, one can consider sanitizing inputs (e.g., *htmlspecialchars* function in PHP) to avoid potential Cross-Site Scripting (Nentwich et al., 2007) or SQL Injection attacks (Halfond and Orso, 2005). These requirements provide other optimization criteria that can be also modeled by an addition of 'big' weights to all non-reliable instruction representations (transitions, for instance) of the corresponding state model.

3 OPTIMIZATION METHODS

3.1 Code Optimization for Tree-like Automata

Different sets of instructions written in the same language can be equivalent. However one of them can be better than others, w.r.t. specific non-functional properties or requirements of the corresponding software. For example, equivalent (from the functional point of view) applications can differ in terms of their performance, energy consumption, disk load, etc. Correspondingly, for the source code, there is a list *Sub* of substitutions which can replace some instructions / sequences of actions. For example, a sequence *ab* can represent the code that is equivalent to the code represented by an action *d*; however, the weight of *d* can be bigger than the sum of weights of *a* and *b* and thus, the *d*–code can be slower or can be more energy consuming than the code represented by the sequential execution of *a* and *b*. We can also use regular expressions, which in fact, correspond to the replacement of a sub-tree in the initial weighted tree automaton. Such regular expressions are well known (Smith et al., 2008) and we do not describe them in detail due to the page limit of this work.

Given a list *Sub* of possible substitutions of traces over the alphabet *A* with traces over alphabet *B*, a trace $\alpha = \alpha_1 \dots \alpha_l$ over alphabet *A* is *equivalent* to trace $\beta = \beta_1 \dots \beta_l$ over alphabet $A \cup B$ if for each $i = 1, \dots, l$, $\alpha_i = \beta_i$ or β_i is a possible substitution for α_i of the list *Sub*. The list of substitutions *Sub* contains only possible substitutions that guarantee that after applying any of such substitutions to a given WTA S, an equivalent WTA S′ is obtained.

An equivalent WTA S′ refers to a WTA obtained after applying a sub-set of substitutions in the *Sub* list to the original WTA S. S′ represents the source code, which is equivalent w.r.t. the overall functional specification of the source code that is represented by S.

Example. Let $A = \{a, b, c\}$, $B = \{a, d\}$ and $Sub = \{aba \rightarrow bc; bcb \rightarrow d; bab \rightarrow cd\}$. Consider the trace *abababa*. If we consider *abababa* as *aba bab a* then we get an equivalent trace *bc baba* that can be considered as *bcb aba*. Thus, the trace *dbc* is equivalent to *abababa*. If each action in the corresponding automaton has a weight one, then the sequence of the weight 7 can be minimized to the one of weight 3. However, if we partition the trace *abababa* in another way as *a bab aba* then an equivalent trace *acdbc* will be obtained that cannot be minimized w.r.t the given set *Sub* of available substitutions.

An important semantic note on the regular expression usage is in the context of replacements. The regular expression replacement actions represent the set of original actions matched by the regular expression over the alphabet *A*. As an example, for the replacement *a.*b* → *db.** the matched sequence of actions is *axazb*, the replacement is *dbxaz*, due to the fact that *xaz* was the matched set of actions in the replacement.

Given a finite list *L* of traces over the alphabet *A* where each action is a weighted action, the problem is to derive traces of minimal weight using substitutions of a list *Sub* over alphabet *B*.

WTA Code Optimization Example. Consider the following simple *PHP* code that outputs the perimeters for the first 1000 circles with integer radius (later referenced as the perimeters code):

```php
<?php
$PI = 3.141592654;
$n = 1000;
for($i = 0; $i < $n; $i++)
 {
  $p = 2 * $PI * $i;
  print("R = $i, P = $p \n");
 }
?>
```

A WTA model shown in Figure 1 was derived manually based on the code listed above.

The weights of the automaton S presented in

Figure 1 are assigned under the assumption that many assignations and expressions of PHP source code have the same cost, which equals one. Therefore, for each node of the corresponding tree, the weight of the incoming edge equals the sum of those for all the outgoing edges plus a penalization constant. The penalization constant, in the running example, adds five to each function call, three to each loop, and one to multiplications.

Weights can favor certain instructions, for instance, to prefer shift left operations instead of multiplications by a power of two.
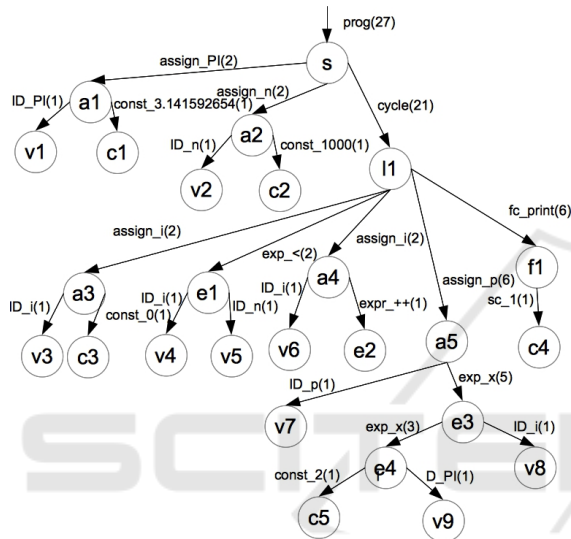
Figure 1: A WTA representation of the perimeters code.

Let the substitution list *Sub* be as follows:

{assign_PI(2)A* → ε(0); assign_n(1)A* → ε(0); ID_n(1)→ const_1000; exp_x(3)exp_(2)A* → const_6.283185308(1)} .

By an application of the substitutions in *Sub* to the WTA in Figure 1 an optimized WTA in Figure 2 is obtained. By following the pre-order tree transversal algorithm for code generation the following optimized code is obtained:

```php
<?php
 for($i = 0; $i < 1000; $i++)
 {
  $p = 6.283185308 * $i;
  print("R = $i, P = $p \n");
 }
?>
```

Please note that, the minimization of the overall weight of the edges at the first level of the corresponding tree, which is the sum of weights of the outgoing edges from the root can be another optimization criterion. If each such outgoing

instruction has weight one, then for replacing we can choose an equivalent tree with the minimal number of root transitions.
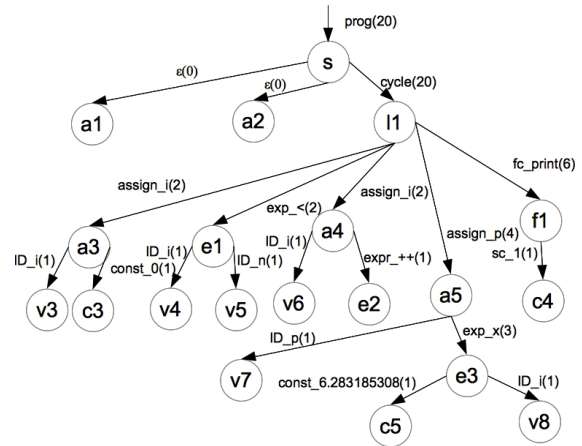

Figure 2: An optimized WTA representation of the perimeters code.

3.2 Code Optimization using Extended Finite Automata

The EFA representation is widely used for code representation since such representations are compact and intuitively very close to code instructions. We have identified some heuristics over the components of the EFA such that after applying these heuristics, an optimized and functionally equivalent EFA is obtained. As an example, consider the heuristic for finding and suppressing a **non-significant** sub-automaton.

Given an EFA *E* extracted from the code, we propose to decompose the automaton *E* into two parts *E*1 and *E*2, such that, for a sub-automaton *E*1 it holds the following: i) each transition of *E*1 does not contain any input or ii) an output. The latter means that the automaton *E*1 is "responsible only" for the internal calculations, i.e. for updating the context variable values. If it holds that for each context variable c_i, that is updated in one of the transitions of the sub-automaton *E*1, c_i does not appear in any predicate included into the sub automaton *E*2 nor in any function that calculates the value of an output parameter, then the sub automaton *E*1 is not significant for the whole behavior of the composition of the machines *E*1 and *E*2. In this case, the automaton *E*1 can be ignored, i.e., can be deleted from the composition.

Consider an example of representing a source code as an EFA. We further illustrate how the EFA can be optimized with the help of the 'non-significant' sub-automaton described above.

285

Extended Finite Automata Code Optimization Example. For a better understanding of how an EFA model can be used for code optimization, consider the following PHP function:

```
function mean($iarr, $n)
{
 $max = $iarr[0];
 for($j=1; $j<$n; $j++)
  if($max < $iarr[$j])
    $max = $iarr[$j];
 for($i=0; $i<$n; $i++)
  $avg += $iarr[$i];
 return $avg / $n;
}
```

The function input parameters are an array, and its length. The output is the mean value of the array. The EFA representation of the code listed above was manually derived and it is shown in Figure 3. We note that the following set of context variables is considered: {*j*, *i*, *s*, *arr_1*, …, *arr_n*}; all context variables are assumed to be initialized to 0, except of *i* which is initialized to 1.

In the running example, the automaton *E*1 is shown in Figure 4 enclosed in a box with a continuous line; the automaton *E*2 is depicted in a box with a dashed line. After applying the above heuristic process, the resulting EFA is obtained (Figure 5).

Figure 3: EFA representation of the PHP mean function.

By direct inspection, one can assure that the optimized EFA is simpler than the initial, and the corresponding source code has less number of instructions. The code that corresponds to the optimized EFA is the following:

```
function mean($iarr, $n)
{
 for($j=0; $j<$n; $j++)
  $avg += $iarr[$j];
 return $avg / $n;
}
```
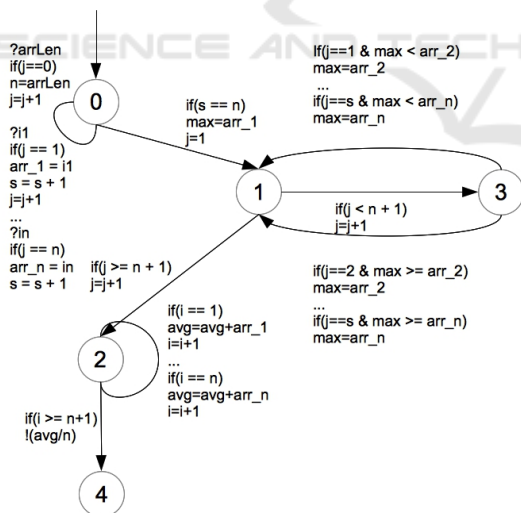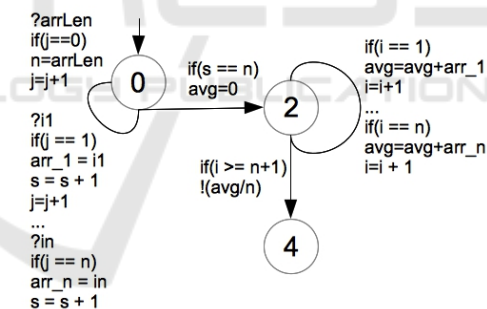
Figure 4: Component identification for the EFA model of the PHP mean function.

Figure 5: An optimized EFA via 'non-significant' sub-automaton heuristics for the PHP mean function.

4 CONCLUSION

In this paper, we have discussed how finite state models can be used for code optimization. As this problem is well-studied and effective methods and tools are well developed in current compilers, we focused on the problem of such optimization for interpreted languages. In particular, we have discussed how state models can be efficiently used to derive another interpreted code, which is better than the original one, according to some criteria (in some sense).

Among the known state models we chose weighted tree-like automata and extended finite automata for code optimization. We provided several heuristics of how the original models can be changed in such a way that substituting / deleting a submachine of the corresponding automaton can lead to a code with a better performance, disk load, energy consumption, etc. As the extraction of original weighted / extended automaton is a hard problem by itself, we briefly describe how the corresponding automata can be derived.

We mention that optimization techniques discussed in the paper are mostly based on the flexibility of the original model and extracting a sub-model that can be substituted or simply, deleted. Such flexibility can be effectively preserved as a largest solution of the corresponding FSM/automata equation (Villa et al., 2015). Therefore, as a future work we would like to apply this theory of equation solving for code optimization. However, such perspective arises new theoretical issues such as solving equations over weighted / extended automata, establishing (necessary and) sufficient conditions for the existence of solutions to such equations, etc. These questions are out of the scope of this position paper and are left for the future work. On the other hand, given the fact that state models are known to be effective for code generation (Giegerich and Graham, 1992), we are also interested in the application of FSM / Automata equation solving for an optimal synthesis of an 'unknown' code component.

Certainly, the efficiency of proposed techniques needs to be experimentally evaluated over larger test cases (larger source code) and this is another direction of our future work. We also plan to experiment in different environments (including compiled languages) given the fact that the proposed approach seems to be applicable for any language parse trees.

ACKNOWLEDGEMENTS

REFERENCES

Villa, T., Petrenko, A., Yevtushenko, N., Mishchenko, A. and Brayton, R., 2015. Component-Based Design by Solving Language Equations. Proceedings of the IEEE, DOI: 10.1109/JPROC.2015.2450937, Volume: 103, Issue: 11, P: 2152-2167, 2015.

Jones, J., 2003. Abstract syntax tree implementation idioms. In the proceedings of the 10th conference on pattern languages of programs.

Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G., 2007. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In the proceedings of the Network and Distributed System Security Symposium (NDSS'07).

Halfond, W. G. J.,Orso, A., 2005. AMNESIA: Analysis and Monitoring for NEutralizing SQL-injection Attacks. In the proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. P: 174-183.

Smith, R., Estan, C.,Jha, S., Kong, S., 2008. Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata. In the proceedings of the ACM SIGCOMM 2008 Conference on Data Communication, P:207-218.

Ivan, I., Boja, C., Vochin, M., Nitescu, I., Toma, C., Popa, M., 2007. Using Genetic Algorithms in Software Optimization. In WSEAS'07, 6th Int. Conference on TELECOMMUNICATIONS and INFORMATICS, Dallas, Texas, USA.

Stallman, R. M. and GCC Developer Community, 2009. Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3, CreateSpace, Paramount, CA, First Edition.

Bakken, S. S., Suraski, Z., Schmid, E., 2000. PHP Manual: Volume 2, iUniverse, Incorporated.

Wall, L., 2000. Programming Perl, O'Reilly & Associates, Inc. 3rd edition. Sebastopol, CA, USA.

Popov N., 2016 PHP Parser, URL: https://github.com/nikic/PHP-Parser, Last visited: 2016-01-14.

Fülöp, Z., Vogler, H., 2009. Weighted Tree Automata and Tree Transducers, P: 313-403, *Handbook of Weighted Automata*, Springer, Berlin Heidelberg

Giegerich, R., and Graham, S L., 1992. Code Generation — Concepts, Tools, Techniques, the proceedings of the International Workshop on Code Generation, Dagstuhl, Germany, Springer –Verlag London