On source-to-source compilers

Evgeniy Ilyushin, Dmitry Namiot

***Abstract* - Computer technologies like computer languages and hardware have been involving for past few decades. We have a lot of computer programs which need to maintain and rewrite when releasing new equipment or technology. It is too expensive and unreliable to rewrite the whole code from scratch. Also the spread of portable devices, which usually have multi-core processors, increase the demands on the quality of the developed programs, their effective work in relation to the consumption of resources. All the above-mentioned reasons complicate the programs and require a large amount of effort from programmers. Moreover, a widely spread of the Internet and web programming with scripting computer languages like JavaScript or Python raised many new problems associated with the quality and reliability of software packages. Source-to-source compilers, also known as transcompiler or transpiler can help to resolve these problems. In this paper, we will describe principles of working for such compilers and consider some of them.**

***Keywords* – source-to-source compiler, source-to-source compilation, optimization, program translation, transcompiler, transpiler.**

I. INTRODUCTION

The first source-to-source compiler was developed in 1981. It translated .ASM source code for the Intel 8080 processor into .A86 source code for the Intel 8086. After the advent of multi-core computing devices was developed automatic parallelizing compilers, e.g. PIPS [1], PLUTO [2], Polaris [3], ROSE [4]. Then began the era of the Internet and scripting computer languages such as Python and JavaScript. They are becoming very popular. But these languages were developed for nonprofessional programmers so computer programs developed in these languages usually has a lot of errors, un-optimized and redundancy code. These causes have made optimizing source-to-source compiler popular also in nowadays, e.g. Google Closure [5], UglifyJS [6], Esmangle [7]. Also, transpilers are used not only for the translation of imperative programming languages, but declarative such as Sass, Less, programs are written in that are translated in CSS, since browsers are able to handle CSS only.

Thus, the main problems that source-to-source compilers solve are:

- translating source code which is written in one language to other approximately the same level of abstraction;
- translating source code to another version of a language;
- automatic parallelization for a sequential source code;
- source code optimization.

In this article, we will focus in more detail on the each of the above tasks, as well as give examples of how compilers solve them, and how these decisions are effective.

Manuscript received Apr 10, 2016.
Evgeniy Ilyushin is a researcher at Lomonosov Moscow State University (e-mail: john.ilyushin@gmail.com).
Dmitry Namiot is a senior scientist at Lomonosov Moscow State University (e-mail: dnamiot@gmail.com).

II. THE ARCHITECTURE AND REQUIREMENTS

As with the traditional compiler, an architecture of transpiler can be divided into two parts – front-end and back-end. The front-end translates the source language into an intermediate representation. The back-end works with the internal representation to produce code in the output language.

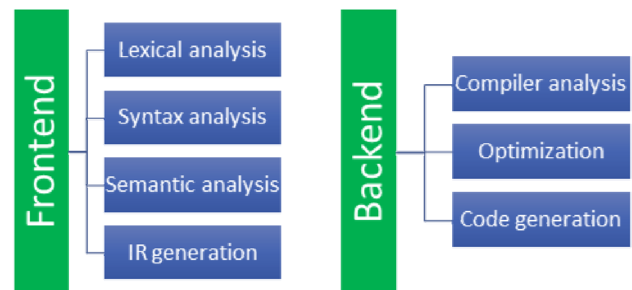

Fig. 1. The architecture of transpiler.

In general, transcompiler takes the source code of a programming language as its input and outputs the source code into another programming language approximately the same level of abstraction or the same language it depends on purpose it. This is a difference between a compiler and a source-to-source compiler because a compiler translates source code to machine code.

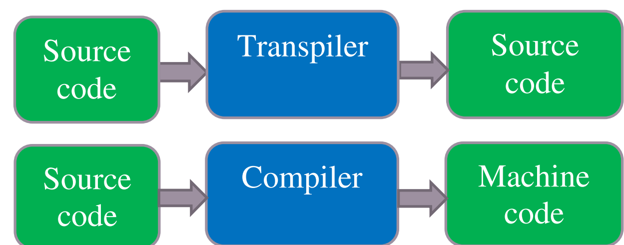

Fig. 2. Principles of working.

Requirements for a typical transpiler are:

1. The resulting program should be close to correct.
2. A result of execution of the translating program has to be exactly the same as a result of execution of source program.
3. The process should have minimal user interaction and fewer user efforts.

III. TRANSLATING FROM ONE LANGUAGE TO OTHER

Source-to-source translation of programs from one high-level language to another has been shown to be an effective aid to programming in many cases. By the use of this approach, it is sometimes possible to produce software more cheaply and reliably.

Cases of using this type of transpiler are:

1. Translation of small programs.

2. Translation of large programs. The automatic translation of large programs usually is not possible. This is due to the fact that they often contain many legacy codes from previous versions, which in turn may contain errors, security vulnerabilities and are not well understood. As an example of such a program, we can mention an operating system. Programs of this class contain millions of lines of code and any attempt to translate the entire program at once will fail, so it should be divided into small areas and translates in stages.

3. Translation of libraries which are planning to do widely spread.

4. Translation within the multi-language runtime. As an example of such translation we can mention the .NET platform by Microsoft [8]. The working scheme of it is presented in the figure below.

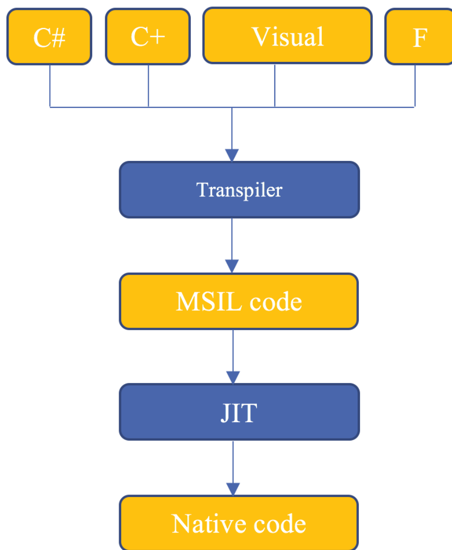

Fig. 3. compiling process of .NET

The source code is written in one of the languages supported by the .NET platform could be converted by transcompiler into intermediate code MSIL (Microsoft Intermediate Language), after that the MSIL code could be translated into Native code.

IV. Translation to another version

This type is used to raise or lower version of computer language for a given source code program. This type of translation could be used in the following cases:

1. Lowering version of the code. After the release of the new standard of a computer language, we want to use new syntax, right now without waiting for compilers support. In this case, we translate our source code written by a new version of a language to an older version. For instance, Babel [9] turns your ES6 code into ES5 friendly code (listing 1).

ES6	ES5

class Test {
 	
getItems(){
return [];
}
 	
saveItem(item){
}
}
	var _prototypeProperties =
	function (child, staticProps, instanceProps) {
	if (staticProps)
	Object.defineProperties(child, staticProps);
	if (instanceProps)
	Object.defineProperties(child.prototype,
	instanceProps); };
	
	var _classCallCheck = function (instance,
	Constructor) {
	if (!(instance instanceof Constructor)) {
	throw new TypeError(«Cannot call a class
	as a function»); } };
	
	var Test = (function () {
	function Test() {
	_classCallCheck(this, Test);
	}
	
	_prototypeProperties(Test, null, {
	getItems: {
	value: function getItems() {
	return [];
	},
	writable: true,
	configurable: true
	},
	saveItem: {
	value: function saveItem(item) {},
	writable: true,
	configurable: true
	}
	});
	
	return Test;
	})();

Listing 1. Example of ES6 class that is transpiled to the ES5 equivalent

2. Raising version of the code. At that rate, similar to the above-mentioned example, we got a new version of a language, but we don't want to use a new syntax and continue to program using the previous version or want to translate the legacy code to the new version. For instance, 2to3 [10] reads Python 2.x source code and applies a series of fixers to transform it into valid Python 3.x code.

As an example of effective using of this type of compiler, we can mention the conversion for more than 80 000 lines unit testing code of four open-source Java applications to use the latest version of the popular JUnit testing framework (is one of a family of unit testing frameworks which is collectively known as xUnit that originated with SUnit) [11].

V. Automatic parallelization

Multicore processors are very commonly used. The industry trend suggests that the number of cores is still going to rise in the coming years. This multicore paradigm shift has forced the software industry to change the way applications are written. To utilize the available cores to their highest potential parallel programs are needed. Similarly, legacy application codes need to be re-written or parallelized so that the new multicore hardware is exploited fully. Writing parallel programs manually is difficult, cost and time consuming and hence there is a need for tools that can aid to convert legacy sequential codes to parallel codes. Such tools are auto-parallelizing transcompilers. The major challenges involved in design and implementation of such a tool include finding alias variables, dependencies between statements, side-effects of function calls etc. Best of these tools exploit task parallelization, loop parallelization and some of them can perform code transformation.

Stages of Automatic parallelization are:

1. Detecting sections of code that can be executed concurrently. The analyzer uses IR provided by the Frontend part of transcompiler. The analyzer will first find out all the functions that are totally independent of each other and mark them as individual tasks. Then analyzer finds which tasks are having dependencies and trying to dispose of its dependencies.

2. A scheduler lists tasks and their dependencies on each other in terms of execution and start times. This stage will produce an optimal schedule in terms of number of processors to be used or the total time of execution of the application.

3. The stage of code generation will insert special construction (instructions for OpenCL, OpenMP, CUDA or other) and can make transformations in the code according to data of the schedule.

Notice that not all transpilers of this type can detect sections of code that can be executed concurrently. Some of them such hiCUDA, SkePU or PGI Accelerator require a programmer to place the constructions in the code by hand (listing 2) [13].

Source code:

```
int N = 512*512;
for(i = 1; i < N; i++)
    B[i] = 3*A[i-1] + 4*A[i] + 3*A[i+1];
```

Source code with directives:

```
int N = 512*512;
#pragma acc region
{
    #pragma acc for independent
    for(i = 1; i < N; i++)
        B[i] = 3*A[i-1] + 4*A[i] + 3*A[i+1];
}
```

Listing 2: Example of using the PGI Accelerator

One of the important issues here is a way we can evaluate the efficiency of these compilers. For example, we can use the following criteria.

Criteria of performance [13]:

- Performance and scalability. The output for a parallel program should give better performance in terms of execution time compared to serial time.
- Memory and time complexity. The output program should be more efficiency in terms of run time and memory usage.
- Parallelization overhead. Parallelization overhead shouldn't kill benefit of using parallel code.

Conclusions about the efficiency of such compilers we can do using results of research in the paper [13]. Authors of the paper chose Par4All [14], CETUS and S2P transcompiler for testing. All these systems are automatic parallelizing compilers. The authors used NAS Parallel Benchmark for testing parallelization tools. These benchmarks were designed to compare the performance of highly parallel computers and are widely recognized as a standard indicator of computer performance. Other than the above NAS benchmark code, they have used a standard matrix multiplication code for benchmarking. As a result, we can say that the performance of parallel code will increase when all the threads are mapped to physical cores. For task level parallelization, the task should have optimal size and fewer dependencies. These tools should try to skip the loops that have smaller execution time.

VI. SOURCE CODE OPTIMIZATION

Nowadays, it is one of the most popular types of transpilers. The main goal of them is translating source code to compact, effective and unmistakable code by the same language. Such popularity transpilers of this type obtained thanks to the wide spread of the Internet and web applications. The vast majority of electronic devices such as PCs, laptops, mobile phones, smart watches etc., provide their users the ability to connect to the Internet and run web applications. A special feature of web applications is that they use in one way or another JavaScript programming language and the user can start the application with the help of a large number of different browsers, which in turn are supported by a particular implementation the language. Each language implementation has its limitations and its own set of optimizations. Some of the ways language optimizations require a long execution time, which adversely affects the user experience and may require considerable expenditure battery power. These problems are typical for all portable devices.

As an example of optimizing transpiler we chose one of the most popular Google Closure by Google, as well as the programs that it has optimized, have been used JavaScript benchmarks Sunspider 1.0.2 and Ubench. We used JavaScript engines V8 and JavaScriptCore for performance analysis.

Table 1. Results of testing

Test's name	decreasing size(%)	decreasing time v8(%)	decreasing time JSC(%)
3d-cube.js	61	-3	-5
3d-morph.js	85	-45	-34
access-binary-trees.js	73	17	14
access-fannkuch.js	70	-32	-20
access-nbody.js	62	13	0
access-nsieve.js	72	-33	-43
bitops-3bit-bits-in-byte.js	77	-20	0
bitops-bits-in-byte.js	70	-3	-6
bitops-bitwise-and.js	91	11	0
bitops-nsieve-bits.js	63	-33	-22
controlflow-recursive.js	55	19	-7
crypto-aes.js	55	3	-6
crypto-md5.js	59	10	6
crypto-sha1.js	75	-14	7
function-closure.js	71	27	40
function-correct-args.js	82	21	5
function-empty.js	61	-3	0
function-excess-args.js	78	18	5
function-missing-args.js	76	6	82
function-sum.js	75	3	6
loop-empty-resolve.js	42	10	-5
loop-empty.js	42	8	3
math-partial-sums.js	66	-33	-6
math-spectral-norm.js	46	21	7
string-fasta.js	24	-9	-13
string-unpack-code.js	2	-4	11
Average:	**62**	**-2**	**1**

As we can see in Table 1, the amount of code has dropped on average by 60% and the performance has dropped in some cases.

From the results as shown in Table 1, we observe the following:

- These tools we can use in the case when we need to reduce a size of the code. For instance, web application each time sends a library to clients. If a size of our library is huge, it will slow down web pages loading speed.

- They aren't effective as optimizing compilers in the case of dynamic programming languages. This is because they can't perform static optimization without knowledge of types. Thus, we should compare the performance of un-optimized and optimized versions of code [15].

Moreover, we checked each pass of Google Closure and calculate their efficient, how many times they called (Table 2):

Table 2. The most efficient passes of Google Closure

Pass name	Number of times	Average reducing code amount	Average execution time (milliseconds)
renameVars	21	737	1
peepholeOptimizations	20	523	1
collapseVariable Declarations	19	34	0
renameProperties	15	148	1
smartNamePass	6	56	14
collapseAnonymousFunctions	4	5	0
coalesceVariable Names	3	134	15
flowSensitiveInlineVariables	3	27	23
exploitAssign	3	18	1

renameVars - renames all the variables names into short names, to reduce code size and also to obfuscate the code.

peepholeOptimizationstyAssignments – consist of:

- PeepholeCollectProper - a pass that looks for assignment to properties of an object or array immediately following its creation using the abbreviated syntax.
- PeepholeFoldConstant – a peephole optimization to fold constants.
- PeepholeMinimizeConditions - a peephole optimization that minimizes conditional expressions according to De Morgan's laws.
- PeepholeRemoveDeadCode – a peephole optimization to remove useless code such as IF's with false guard conditions, comma operator left-hand sides with no side effects, etc.
- PeepholeReplaceKnownMethods - just to fold known methods when they are called with constants.
- PeepholeSimplifyRegExp - simplifies regular expression patterns and flags.
- PeepholeSubstituteAlternateSyntax - a peephole optimization that minimizes code by simplifying conditional expressions, replacing IFs with HOOKs, replacing object constructors with literals, and simplifying returns.

collapseVariableDeclarations - tests for variable declaration collapsing.

renameProperties – renames properties (including methods) of all JavaScript objects. This includes prototypes, functions, object literals, etc.

collapseAnonymousFunctions - collapses anonymous function expressions into named function declarations.

coalesceVariableNames - Reuse variable names if possible.

flowSensitiveInlineVariables - inline variables when possible. This pass attempts to inline a variable by placing the value at the definition where the variable is used.

exploitAssign - tries to chain assignments together.

Below is an example of source-to-source optimization performed on controlflow-recursive.js (Listing 2).

Unoptimized	Optimized

*function ack(m,n){
if (m==0) { return n+1; }
if (n==0) { return ack(m-1,1); }
return ack(m-1, ack(m,n-1));
}*
	*function c(a,b){
	return 0==a?b+1:0==b?c(a-1,1):c(a-1,c(a,b-1))
	}*

*function fib(n) {
if (n < 2){ return 1; }
return fib(n-2) + fib(n-1);
}*
	*function d(a){
	return 2>a?1:d(a-2)+d(a-1)
	}*

*function tak(x,y,z) {
if (y >= x) return z;
return tak(tak(x-1,y,z), tak(y-1,z,x), tak(z-1,x,y));
}*
	*function e(a,b,g){
	return b>=a?g:e(e(a-1,b,g),e(b-1,g,a),e(g-1,a,b))
	}*

*var result = 0;
 	
for (var i = 3; i <= 5; i++) {
result += ack(3,i);
result += fib(17.0+i);
result += tak(3*i+3,2*i+2,i+1);
}*
	*for(var f=0,h=3;5>=h;h++)
	f+=c(3,h),f+=d(17+h),
	f+=e(3*h+3,2*h+2,h+1);*

*var expected = 57775;
if (result != expected)
throw "ERROR: bad result: expected " + expected + " but got " + result;*
	if(57775!=f)throw"ERROR: bad result: expected 57775 but got "+f;

Listing 2: Example of optimization

VII. CONCLUSION

This technology could lead to significant increases in productivity and reliability of software. The potential benefits include faster coding and more reliable software, though testing, debugging, and hand coding would still be necessary. In the case when we are planning to use an auto-parallelizing transpiler, we should remember about problems associated with detecting dependencies between statements and be able to write code minimizing these dependencies, understanding dependency between possibilities of hardware and software. Furthermore, one should not forget that optimizing compilers can sometimes reduce the performance of a program and we should compare the performance of both unoptimized and optimized version of the code.

REFERENCES

[1] PIPS http://pips4u.org/
[2] PLUTO http://pluto-compiler.sourceforge.net/
[3] POLARIS http://polaris.cs.uiuc.edu/polaris/polaris-old.html
[4] ROSE http://rosecompiler.org/
[5] Google Closure https://developers.google.com/closure/compiler/
[6] UglifyJS https://github.com/mishoo/UglifyJS
[7] Esmangle https://github.com/estools/esmangle
[8] Source-to-Source Translation and Software Engineering http://dx.doi.org/10.4236/jsea.2013.64A005 Retrieved: Jun 2013.
[9] Babel https://babeljs.io/
[10] 2to3 https://docs.python.org/2/library/2to3.html
[11] W. Tansey and E. Tilevich, “Annotation Refactoring: Inferring Upgrade Transformations for Legacy Applications,” Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA ’08), Vol. 43, No. 10, 2008, pp. 295-312.
[12] C. Nugteren and H. Corporaal «Introducing ‘Bones’: A Parallelizing Source-to-Source Compiler Based on Algorithmic Skeletons»
[13] A. Athavale, P. Randive and A. Kambale «Automatic Parallelization of Sequential Codes Using S2P Tool and Benchmarking of the Generated Parallel Codes» http://www.kpit.com/downloads/research-papers/automatic-parallelization-sequential-codes.pdf
[14] Par4All https://github.com/Par4All/par4all
[15] I.O. Zolotareva, O.O. Knyga «Modern JavaScript project optimizers»