International Journal of Automation and Computing 15(2), April 2018, 228-238
DOI: 10.1007/s11633-017-1100-0

Improving on Linear Scan Register Allocation

Shahrzad Kananizadeh[1] Kirill Kononenko[2]

[1] Department of Computer Science, Saarland University, Saarbrücken, Germany

[2] École Normale Supérieure/French Institute for Research in Computer Science and Automation (INRIA), Paris, France

Abstract: Register allocation is a major step for all compilers. Various register allocation algorithms have been developed over the decades. This work describes a new class of rapid register allocation algorithms and presents experimental data on their behavior. Our research encourages the avoidance of graphing and graph-coloring based on the fact that precise graph-coloring is nondeterministic polynomial time-complete (NP-complete), which is not suitable for real-time tasks. In addition, practical graph-coloring algorithms tend to use polynomial-time heuristics. In dynamic compilation environments, their super linear complexity makes them unsuitable for register allocation and code generation. Existing tools for code generation and register allocation do not completely fulfill the requirements of fast compilation. Existing approaches either do not allow for the optimization of register allocation to be achieved comprehensively with a sufficient degree of performance or they require an unjustifiable amount of time and/or resources. Therefore, we propose a new class of register allocation and code generation algorithms that can be performed in linear time. These algorithms are based on the mathematical foundations of abstract interpretation and the computation of the level of abstraction. They have been implemented in a specialized library for just-in-time compilation. The specialization of this library involves the execution of common intermediate language (CIL) and low level virtual machine (LLVM) with a focus on embedded systems.

Keywords: Register allocation, just-in-time compilation, code generation, static analysis, dynamic analysis.

1 Introduction

"Just-in-time" compilers during program execution are being increasingly used for the execution of dynamic languages such as Perl and Python, and for semi-dynamic languages, such as Java and C++.

Currently, however, there is one common problem, which arises in the use of the existing approaches of fast just-in-time compilation. In almost every case, the just-in-time compiler is very specific to the used object model, to the library running programs, garbage collection and other features of bytecode for the particular system[1–3]. This inevitably leads to a situation where the effort needs to be repeated, and all the useful work that went into creating and developing a fast and high quality just-in-time compiler for a single virtual machine and a programming language cannot be used a second time for another virtual machine and another programming language.

Just-in-time compilers are not only useful in the implementation of programming languages. They can also be used in other fields of mathematics and programming. For example, graphical applications can achieve better execution if they compile special means of just-in-time execution of the program, specialized for "manual" generation of images, more accurately, achieving a better performance than statically generated general-purpose means. Obviously, such applications do not need object models, garbage-collectors, or large class libraries.

The greatest amount of work during the development of a new just-in-time compiler relates to the support operations of arithmetics, conversion of data types, reading from memory, writing to memory, creation of loops, analysis of the data-flow graph, algorithm of register allocation, and generation of executable machine code. Only a very small fraction of the work is dependent on the bytecode of the virtual machine and the programming language. The purpose of the libjit-linear-scan[4, 5] project consists of providing a necessary and sufficient set of tools that ensure the process of fast just-in-time compilation of the program, liberating the developer from the work on the features of the programming language and operating system. In those parts where the support of common object models is available, this is done strictly in additional libraries of the operating system, and not in the form of code of the kernel of the execution environment.

The libjit-linear-scan library aims to provide developers with more time and opportunity to think about the high-level design of the virtual machine, without focusing on the details of generation and execution of binary code. At the same time, experts in optimization, design, compiler implementation, and native code generation can focus on more low-level problems of binary code execution, instead of working on applied problems.

Super linear complexity algorithms, which are used in compilers, can fail in case of an algorithmic complexity attack. Hence, algorithms that are fast and have a linear complexity of O(n) are often used. When this is im-

Research Article
Manuscript received November 7, 2016; accepted August 2, 2017; published online February 28, 2018
Recommended by Associate Editor James Whidborne

possible, it is expedient to use algorithms that have a higher complexity. The library that we have developed in this study implements functionality via fast algorithms based on just-in-time compilation with a linear complexity. This library was created from the ground up to support just-in-time compilation, independent of any format of bytecode or programming language. Currently, libjit-linear-scan[4, 5] is used as a just-in-time compiler for Portable.NET[4, 6] and low level virtual machine (LLVM)[7, 8]. In order to facilitate just-in-time compilation in common language runtime and embedded systems, libjit-linear-scan as well as the just-in-time compiler of Portable.NET have been created with the original support of the open source automation development lab (OSADL) in the DotGNU project[6]. Therefore, this extension is compatible with all software that already uses its application programming interface (API).

The contributions of our work are two-fold. First, we describe a novel approach for the register allocation based on a virtual execution environment that combines information from static and dynamic program analyses. Second, this paper advances the field of computer science by developing tools for the just-in-time compilation of program code.

The remainder of this paper is organized as follows. The next section presents an overview of the various approaches for the fast register allocation. Section 3 presents our approach. Section 3.1 presents the principles of register allocation via graph coloring. Section 3.2 presents the formulation of the novel mathematical model. Section 3.3 presents the principles for implementing our approach. Section 4 contains the test results, and it presents and analyzes the obtained experimental data in terms of the performance. Section 5 presents a discussion on the results. The final section concludes the paper.

2 Literature review

Register allocation involves four particular problems: the computation of $\langle du \rangle$ chains, the dead-code elimination, the definition of live variables, and graph coloring.

Altogether, these problems can be effectively solved for chordal graphs. The interference graphs of program code in static single assignment (SSA) form are chordal (e.g., Fig. 1). Therefore, we will solve all these tasks for chordal graphs.

The compiler optimization of register allocation in machine code causes a significant increase in code execution performance.

The most well-known approach to global register allocation is via graph coloring[2, 9], as formalized by Cocke[10], Ershov[11] and Schwartz[12] and later implemented by Chaitin[13, 14], Chow and Hennessy[15]. Since the basic graph-coloring problem underlying this approach is nondeterministic polynomial time-complete (NP-complete), however, any practical algorithm must simplify the problem using heuristics[16–19]. Many such heuristic

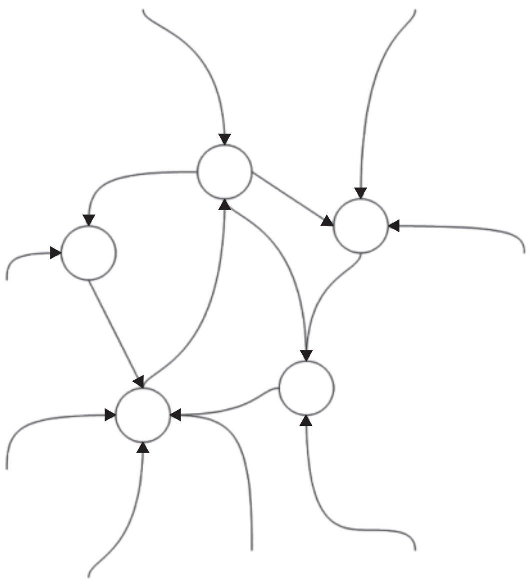
Fig. 1 An example of a chordal graph

variations have been explored, such as linear scan[20–22], integer linear programming[23, 24], puzzle solving[25, 26], repairing[27, 28], trees[29] and traces[30]. Krause[18] proves that given certain restrictions on the input program (e.g., for C a limit on the use of goto labels) graph-coloring register allocation can be done in polynomial time for a fixed number of registers. The approach is not a practical choice for dynamic compilers since the execution time is exponential in the number of registers. The work of Lozano et al.[19] contains a proof that any optimal graph-coloring register allocation algorithms will have execution time exponential in the number of registers. The latest research on graph coloring based register allocation suggests the Callahan-Koblenz algorithm[31, 32] that represents the hierarchical program structure with a tile tree. Such a tree structure isolates the high-frequency and low-frequency code regions and provides a basis for the allocator's overall operation and spill placement decisions. While those heuristics yield polynomial-time algorithms, they do not generally run in linear time, with the exception of the Bouchez, Brisk, Hack et al.[28, 33–37] algorithms for programs in static single assignment (SSA) form[38–40], which partly inspired the present work. We also want to draw attention to useful works[41–43] that develop a similar approach. We make the novel observation that most of the advantages of this linear-time algorithm may be obtained without actually transforming the program to SSA form, since many realistic programs appear to have chordal inference graphs, thereby enabling a much simpler algorithm that can generate code nearly as efficiently.

3 Methods

3.1 Register allocation based on coloring

Graph coloring based on constructing an interference graph of variables is the traditional approach to register allocation. The graph is used for mapping the set of dynamic states of variables into the set of their static states during program execution. Vertices in the graph are variables of the program. Thus, the vertices of the graph also represent symbolic and machine registers that are allocated for temporary variables. The method of register allocation based on graph coloring in the general case, without the use of heuristics, has an NP-complexity. Many approximate solutions of this problem have been studied. However, they are all heuristics. Among the best known is the heuristic of Chaitin based on graph coloring. However, this heuristic works in $O(n^2)$ time, where n is number of variables or the size of the program in a more complex formulation. Just-in-time compilers should compile and distribute registers in $O(n)$ time. The problem of optimal register allocation in an infinite amount of time on making the decision can be solved by exhaustive search. A variant of machine code for a fixed variant of allocation of registers and local memory is generated. The variant of generated machine code is analyzed for optimality by counting the number of instructions addressing memory in the machine code, or counting the number of memory accesses during program execution on the basis of the statistics.

If two variables interfere with each other, then an arc is drawn between the vertices of the interference graph. For non-interfering variables, we can use the same register, thereby reducing the number of registers required. On the other hand, the arc between two variables means that we should not assign them both to the same register, because their liveness at a certain point in time of program execution intersects. This graph can be represented as a matrix of lists, depending on which operations are required.

The basic idea of global register allocation (Fig. 2) can be expressed in five steps, which are as follows:

1) During code generation and optimization (whatever phase has preceded allocation of registers) or, as the first stage of register allocation, allocate unique symbolic registers to objects that can be assigned to registers, unique symbolic registers, for example, s1, s2,$\cdots$, using as many of them as necessary to store all objects (source variables, temporary variables, large numerical constants, etc.).

2) Determine which objects should be candidates for allocation into registers.

3) Build a so-called interference graph. The interference graph is a graph whose nodes represent the allocated objects and real machine registers of the target machine, and whose arcs (or directed arcs) represent interferences. Then, two allocated objects intersect if they are alive at the same time. An object and a register interfere if the object cannot or should not be allocated to this register (e.g., an integer operand or a register for floating-point values).

4) Color the node of the interference graph with R colors, where R is the number of available registers, so that every two adjacent nodes have different colors (this is called coloring with R-colors).

5) Select for each object a register which is of the same color.

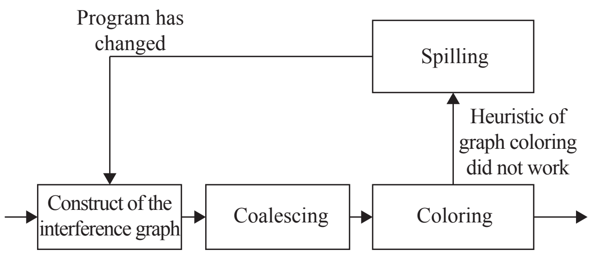
Program has changed
Spilling
Heuristic of graph coloring did not work
Construct of the interference graph
Coalescing
Coloring

Fig. 2 A general description of algorithms based on graph coloring

Fig. 3 shows an example of a program that makes addition of numbers in the variables. All the variables with a common vertex in the interference graph are assigned to different machine registers.

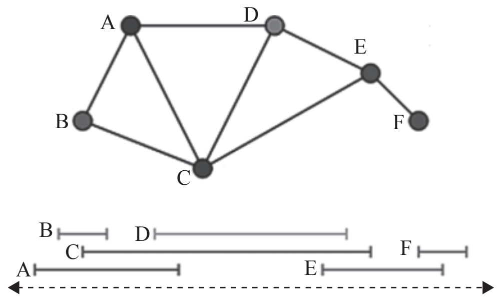

Fig. 3 Register allocation via graph coloring

Many problems can be solved more efficiently in the case of chordal graphs. In SSA form, $\langle du \rangle$ chains are expressed explicitly. Interference graphs of programs in SSA form are chordal. Nonetheless, the Φ-functions themselves are not computable. The problem of optimization of Φ-functions for their transformation to machine code is NP-complete. Thus, the problem of optimal register allocation is NP-complete. Chordal graphs are perfectly orderable, which allows the rapid coloring of graphs. Is it possible to find the given ordering rapidly? It is evident that for interval graphs, the given order is expressed explicitly. It can be assumed that interference graphs of realistic programs are chordal and even interval.

3.2 Mathematical model

If all variants of the execution of a program were deterministic and finite, then of course it would be possible to compute the result of the execution of this program using concrete hardware at any time before its execution. However, some programs can achieve an infinite number of side effects from the wholly deterministic area of access for the program, such as reading and writing into the shared memory. Those side effects can be infinite and even incomputable. Therefore, we propose a new mathematical model in order to take this behavior into account. In the mathematical model that we created, there is a possibility to compute the result of the execution of a program using concrete hardware at any time before its execution if all variants of its execution are deterministic and finite. A good machine code generator supports a few mechanisms for its generation and optimization of register allocation. Thus, in order to make the optimal decision, the infinite variants must be analyzed at an appropriate time. This can be done with the use of domain-specific optimization and an analysis of program behavior. For these computations, we use a linear analysis and the mathematical apparatus of probability spaces based on the Kolmogorov axioms. We define the level of abstraction as the probability of making a correct decision about the properties of a program before its actual execution on real hardware, based on the analysis of a limited set of data performed in linear time. Thus, we formalize the hypothesis that more abstract bytecode can be executed more efficiently than less abstract bytecode. Limited resources are to be allocated to the execution of solutions with a maximum level of abstraction, with the goal of maximizing the expectation of success in the selected path of the execution of the program. In other words, we look at the machine code generated by the dynamic compiler as a finite set of random variables. Each of these random variables has its own distribution. Nevertheless, those distributions can be incomputable or be infinitely divisible.

We look at a sequence of random variables arranged in some geometrical space (e.g., Fig. 4). First, an input bytecode method is verified and is converted into an intermediate form. Each concrete realization of the binary code is a realization of the random value. We define a function that computes the expected values of a certain target function defined on this binary code.

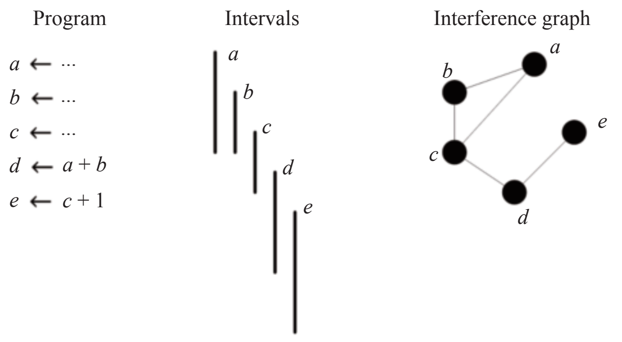

Fig. 4 A sequence of variables

We expect monotonic properties of the target function from applications of the resources of one class to more resources of the second class. The use of first class resources requires much less execution time for a piece of machine code on a hardware processor, and thus for the entire program. On the other hand, their usage highlights the need to commit second-class resources at a certain moment in time. Register allocation is simply the easiest example to demonstrate the efficiency of our model for optimization.

Unless we have fully deterministic control of concrete realizations we believe that such realizations can have various distributions of probability according to the concrete values of some input parameters.

Thus, we analyze this situation using a simple hidden Markov model with a finite number of linearly ordered nodes (e.g., Fig. 5). In other words, there is a description of the properties of the signal that is analogous to wavelet transforms and thus an analogous relation between the apparatus of characteristic functions and the formulas developed by Kolmogorov, Lévy and Khintchine in the computation of the properties of stochastic processes.

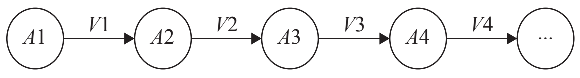

Fig. 5 Markov model

We believe that in the context of uncertainties, nondeterminism and a set of possibilities with equal or variable weights, resources should be directed toward the solution with the maximal probability of obtaining the correct solution based on the ahead-of-time prediction of the program in the given domain of optimization.

3.3 Implementation

We apply abstract interpretation[44–46] in order to predict the properties and behavior of a program before its execution.

We define the goals of register allocation for a given set of properties of a program as, first, the correctness of execution and, second, the minimization of the number of accesses of the local stack memory. We predict the properties of the given program before its execution on real hardware. Each binary property has a certain probability of being obtained. We store each of these binary properties in a bit. Each bit is a degree of freedom. Static analysis creates sets of these bits. Our register allocation algorithms commit the set of bits created by static analysis for optimization and code generation. For a given program there is one deterministic such set of bits.

The probability of predicting the properties of a given program before its execution in a linear time is inversely proportional to the size of the analyzed program. We also describe below the required transformation for the computation of that probability and its properties. We call this probability the level of abstraction. Because it is practically difficult to build a simple deterministic process for the computation of all possible variants of behavior for a defined program, we also look at the defined properties of the generated machine code. Each piece of machine code has an expected weight given by an approximation which depends linearly on the computed level of abstraction.

Properties are defined on a certain domain of optimization. For example, as mentioned above, they can be the number of accesses of the memory, the security of the machine code and/or the size of the generated machine code. In all cases, there should be an interval of machine code for achieving some set (or singleton) of the defined strategies of optimization. Thus, we look at the process of register allocation for just-in-time compilers as the result of the defined process of static analysis. For that defined process, each managed method is mapped onto a set of random variables in a defined probability space. That random variable vector is a set of possible machine codes as well as a set of data for which the source code is compiled. In other words, each compiled object is a realization of an ordered set of random variables. We consider their distributions below.

We define this weight as the difference between the expected optimistic gains and the expected pessimistic losses. In other words, we look at the expected best variant and the expected worst variant. We then define a function of wins and losses, which are defined on primitive chains of events $\langle X\rangle$:

$$\begin{aligned}
&\mathrm{E}(losses(\langle X\rangle)) = \\
&\quad\int_{t=0}^{t=T} abstraction(\langle X(t)\rangle)\mathrm{E}(losses(\langle X(t)\rangle))\mathrm{d}t \\
&\mathrm{E}(wins(\langle X\rangle)) = \\
&\quad\int_{t=0}^{t=T} abstraction(\langle X(t)\rangle)\mathrm{E}(wins(\langle X(t)\rangle))\mathrm{d}t \\
&\mathrm{E}(weight(\langle X\rangle)) = \mathrm{E}(wins(\langle X\rangle) - losses(\langle X\rangle)) = \\
&\quad\mathrm{E}(wins(\langle X\rangle)) - \mathrm{E}(losses(\langle X\rangle)) = \\
&\quad\int_{t=0}^{t=T} abstraction(\langle X(t)\rangle)\mathrm{E}(weight(\langle X(t)\rangle))\mathrm{d}t.
\end{aligned}$$

$\mathrm{E}(weight\langle X(t)\rangle)$ defines the expected weight that a block with index $i(t)$ should have in case there is a correct decision with a probability 1. Moreover, if the block does indeed have that given property, then the information about the properties of that signal has been used correctly to achieve the expected benefits. We expect that when all accessible information has been used, rather than just a part of it, then the correct decision can be made.

We represent the intermediate code by independent sets, in which there are variables. Each variable is assigned a virtual register for all ranges of its life. There is a limited set of resources for the allocation of hardware registers. We optimize the use of this limited resource of hardware registers for register allocation. Below, we show the expected weight that a variable should be assigned a hardware register.

Let us look at the step in which there is already information about the ranges of a live variable. Suppose the distribution of properties of the intermediate code in a certain continuous region that contains the properties for optimization is uniform and the level of abstraction is constant. This is a valid assumption because a general distribution can usually be approximated by various uniform distributions in different regions, and we may even let the number of regions become infinite. Our motivation for this assumption is that the probability of finding properties for a sequence of bits with only a limited number of bits in finite time is inversely proportional to the total number of bits in that sequence and proportional to the total number of bits that can be analyzed in linear time. Of course, it is possible to have an unknown distribution. Nevertheless, we believe that a uniform distribution is a good approximation at this scale. Thus,

$$abstraction(\langle X\rangle) = \frac{1}{Power(\langle X\rangle)}.$$

The power function returns the number of bits that can be used in the representation of the set of all possible variants of the behavior of the machine code and data. Each block of bits can either use a variable or ignore it. We introduce the function $\Delta(t)$ which is equal to the expected number of times a given variable is used in a set of bits to access memory in the time interval from t to $t + \Delta t$. We conclude that

$$\mathrm{E}(weight(\langle X\rangle)) = \int_A^B \frac{1}{Power(\langle X\rangle)}\Delta(t)\mathrm{d}t = \frac{U}{B-A}.$$

Here, U is the total number of usages of the variable. Each variable has a physical geometrical space in the program representation where it is live. A is the coordinate of the appearance of the first-used live range of the variable, and B is the coordinate of its last-used live range. When U is constant for all the variables and there is a high intensity of appearance of new variables and A is much smaller than B, we get the formula for the weights of the original heuristic linear scan.

$$\mathrm{E}(weight(\langle X\rangle)) = \frac{1}{B}.$$

4 Results

This section discusses the experimental results of the use of the developed library. Five levels of optimization are supported. Optimization level 0 uses the default calling convention, global register allocation based on the number of uses, and floating-point operations are using floating point unit (FPU). Optimization level 1 uses only local register allocation in each block of instructions. Optimization level 2 uses linear scan, and fast analysis of the liveness of variables is using the graph of control flow. At optimization level 2, only one interval of liveness of variables for each virtual register is present. Optimization level 3 uses the algorithm of second-chance bin-packing based on multiple intervals of liveness of variables for each virtual register. Optimization level 4 includes dead-code elimination. In addition, the new generator of machine code uses single instruction, multiple data (SIMD) extensions and streaming SIMD extension registers (XMM) registers to represent floating-point operations.

PnetMark[47, 48] was created on the basis of CaffeineMark benchmarks for the Java virtual machine. Conversion of test was made using the transformation of benchmarks from descriptions in Java language into C++. The main purpose of this performance benchmark is to compare different versions of Portable.NET and identify the place that requires improvement. Higher values mean better performance of the runtime in certain particular conditions. The SciMark2 shows the performance in scientific and numeric computations. PnetMark benchmark consists of a set of benchmarks Sieve, Loop, Logic, String, Float, and Method. Linpack shows the performance of the system in solving problems of numeric linear algebra.

Each benchmark was executed 80 times for each variant of the execution environment, shown in Table 1, and the algorithm of register allocation. This number was selected as a result of consultations with the specialists, who develop Portable.NET.

On the *Y*-axis in Figs. 6–12 is the average number of invocations of the function in a given unit of time. The increase of performance in the presented benchmarks was computed as the number, which is determined relatively to the performance of PnetMark benchmark for the mode LS-O4 internal and direct unrolled.

Analogically, the performance values for the SciMark2 and Linpack benchmarks are also computed below.

Fig. 6 determines the time of passing all regression tests of the library of classes.

Figs. 7 and 8 present the results for the PnetMark, SciMark2, and Linpack benchmarks.

Higher values mean better performance of the runtime in benchmarks. The increase of performance in PnetMark is 116%, in SciMark2 it is 239%, and in Linpack it is 136%.

As seen from the estimates of standard deviation, the variance of the results in our environment is substantially less than other variants of the execution environment of .NET from competitors and the previous version of the just-in-time compiler.

Let us look at two examples of functions that we call power, and methods for computation of the expected benefit. The first definition uses the number of operational

Table 1 Programs' execution environments

VM	Variant of the environment	Optimization level	Symbol
1 Portable.NET	Internal ABI	4	LS-O4 internal
2 Portable.NET	Cdecl ABI	4	LS-O4 cdecl
3 Portable.NET	Internal ABI	3	LS-O3 internal
4 Portable.NET	Cdecl ABI	3	LS-O3 cdecl
5 Portable.NET	Internal ABI	2	LS-O2 internal
6 Portable.NET	Cdecl ABI	2	LS-O2 cdecl
7 Portable.NET	Internal ABI	1	LS-O1 internal
8 Portable.NET	Cdecl ABI	1	LS-O1 cdecl
9 Portable.NET	Cdecl ABI allocation of registers using the number of usages of the variable for the operations with numbers with a floating point. For operations with numbers with a floating point, the x87 co-processor is used. Priority is for the variables with the maximal number of uses	0	O0 cdecl
10 Portable.NET	Unrolling of the interpreter loop		Direct unrolled
11 Portable.NET	Interpreter based on pointers on handlers of the byte-code (direct threaded)		Direct threaded
12 Portable.NET	Interpreter		Token threaded
13 Portable.NET	Mode of interpreter of libjit		Libjit interpreter
14 Microsoft.NET Framework 2.0			.NET 2.0
15 Mono	2.4		Mono 2.4
16 Mono	2.2		Mono 2.2
17 Mono	2.0		Mono 2.0
18 Mono	1.1		Mono 1.1
19 Mono	Mint 1.1.11		Mint 1.1

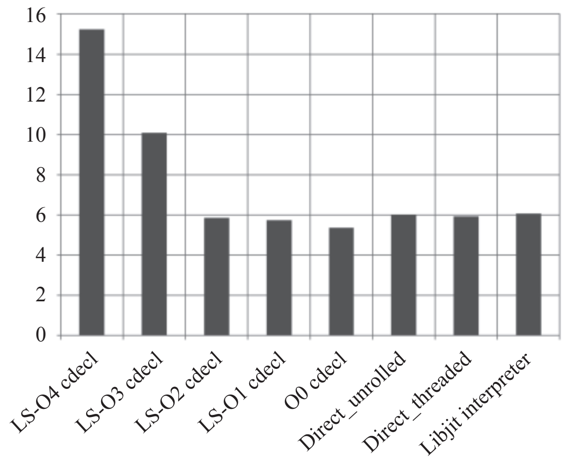

Fig. 6 Time of passing tests of the library of classes of Portable.NET (in seconds)

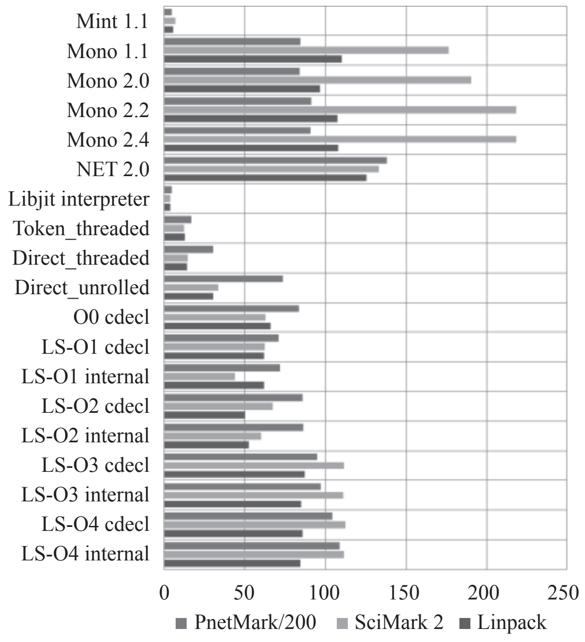

Fig. 7 PnetMark, SciMark2, Linpack benchmarks

codes included in the set of intervals of liveness, and the number of times the variable was used. This register allocator is designated as abstract linear scan (ALS). The second definition uses the size of the machine code and the number of times the memory in the generated machine code was accessed. In other words, each operational code has two elements with two offsets, wherein the machine code starts with the current operational code and ends with it. Computations are performed for the number of accesses to the primitive code generator i486_membase_emit, to which a counter based on a global variable is added. In other words, we computed the number of expected memory accesses in the generated machine code, excluding loops. This register allocator is denoted as an abstract linear scan based on the heuristics of machine code. The allocators of registers are compared to the heuristics of simple linear scan, which saves the register with the farthest end of time of liveness. We denote this register allocator as linear scan (LS). Moreover, another heuristic of register allocation is used, which is based on the number of intervals of liveness of each register. The results of the application of these heuristics are presented in Fig. 8. An analysis of the results demonstrates that the improvement in code speed results in smaller size of machine code.

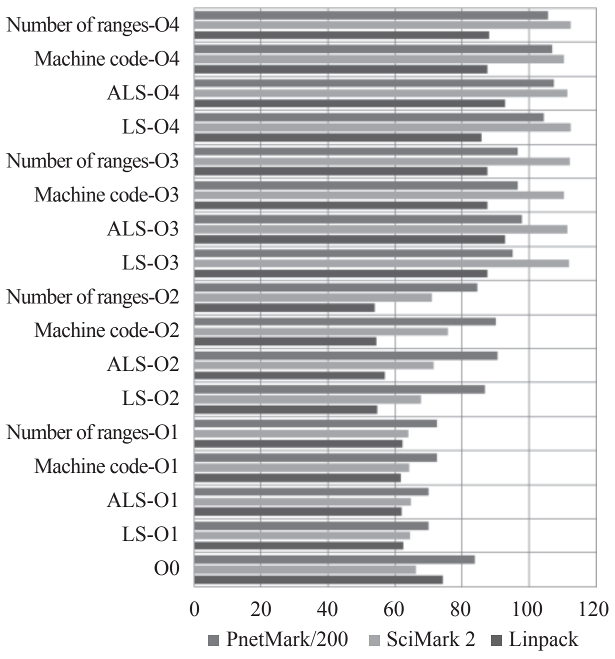

Fig. 8 PnetMark, SciMark2, Linpack benchmarks

The graphs in Figs. 9–12 show the estimation of standard deviation for various performance benchmarks. It is worth noting that the numerical expression of estima-

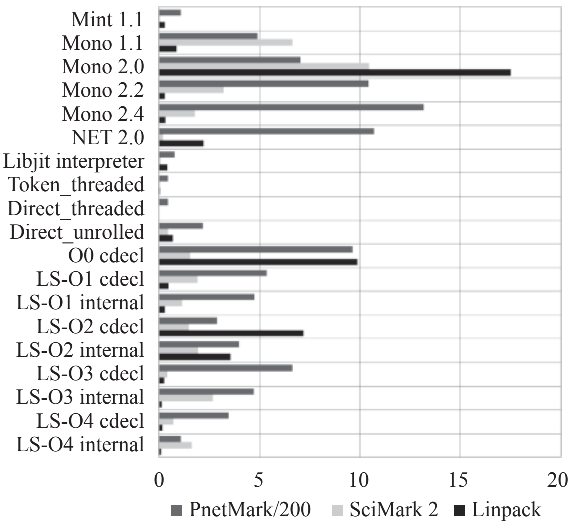

Fig. 9 Estimation of deviation of values of PnetMark, SciMark2 and Linpack benchmarks

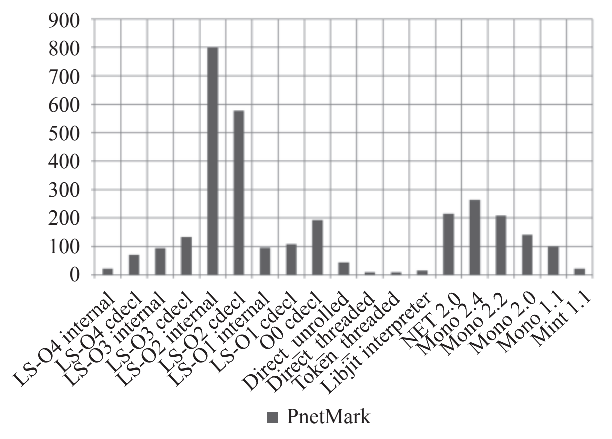

Fig. 10 Estimation of deviation of values of PnetMark benchmark

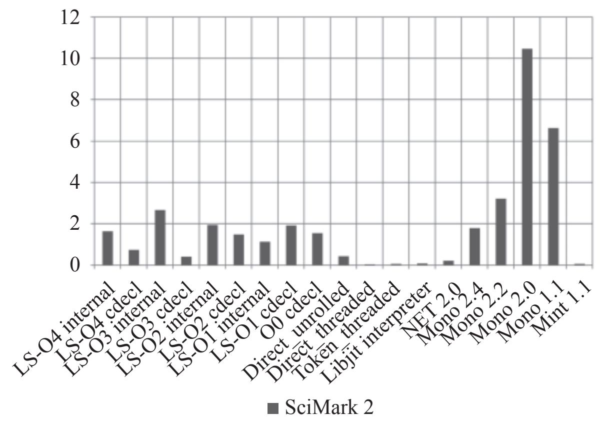

Fig. 11 Estimation of deviation of values of SciMark2 benchmark

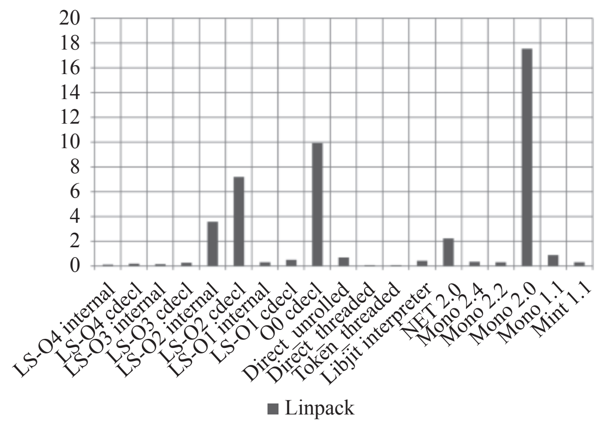

Fig. 12 Estimation of deviation of values of Linpack benchmark

tions of dispersion for these random values has a largely demonstrative and relative character. This is because the obtained results of its numerical expression vary depending on the hardware platform and a variety of other factors. Each point on the graphs took 2 hours of computation and required 80 runs of the benchmark. It has determining value for the analysis of the research results. It tells that objective unbiased statistical studies over a considerable period of time were conducted, a significant sample from realizations of random variables was taken. These results are objective for this hardware platform and operating system. They show the possibility of conducting repeated measurements and are analogous in quality to the results for other platforms and operating systems.

5 Discussions

As follows from the data in Figs. 6–12, the new heuristics of register allocation show improvements in performance benchmarks in comparison with the use of the simple heuristic of linear scan. When using register allocation based on machine code, analysis of the assembly language code shows the inefficiency of the algorithm of local allocation of registers, applied to the chosen path of optimization using the algorithm of global allocation of registers. The algorithm of local allocation of registers is the next object for further research and optimization. When using the heuristic with machine code, there is an improvement of the PnetMark benchmark. However, the performance of SciMark2 and Linpack decreases. Studies show that this appears to be due to the inefficiency of using only two points of optimization. This methodology and further possible improvements are similar to the approach of applying the simplex method for finding the optimal result. In this case, the global algorithm of allocation of registers yields impressive results, both the number of virtual registers that use the local stack and the size of the local stack used for global variables are smaller. However, the local algorithm of allocation of registers adds many accesses to the memory stack. This can be solved by using a larger number of experimental points and a greater number of optimization passes. The experiment confirms that most of the variables in the intermediate representation, which is obtained from the compilation of common intermediate language (CIL), have only one interval of liveness. Most of the intervals of liveness of virtual registers are short and have a length not greater than three operational codes if dead-code elimination is not applied previously. Most intervals of liveness of virtual registers are not above four operational codes if dead-code elimination was applied. Most of these variables are temporary. Thus, the use of the heuristic of saving into memory virtual registers with the largest number of intervals of liveness is justified. Fig. 13 shows the sample distribution function of the lengths of liveness of variables.

The number of intervals of liveness with optimization level 2 is always equal to 1. The distribution of the number of intervals of liveness with optimization levels 3 and 4 is shown in Fig. 14.

The improvement in performance benchmarks is the result of the analysis of variables′ liveness space and

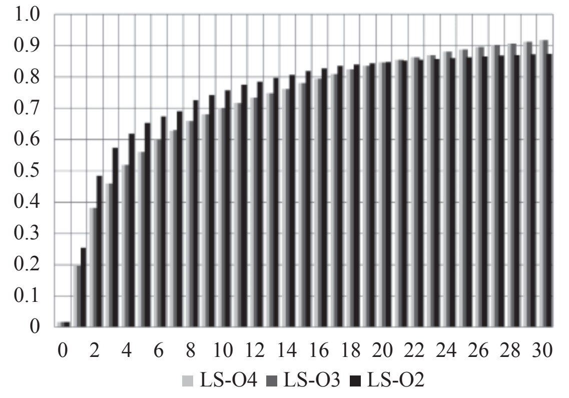

Fig. 13 Sample distribution function of the length of the interval

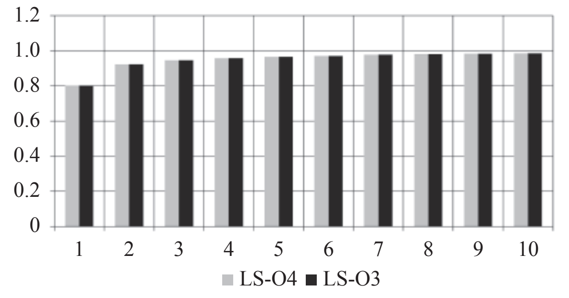

Fig. 14 Sample distribution function of the number of intervals

dead-code elimination. The most productive algorithm variant is the one that uses the intermediate representation for the computation of variables′ weights. In our opinion, the heuristic based on the candidates of machine code has greater potential. It gives results that are close to the heuristic described above with the use of two points. In addition, it requires more computational time, because it can use more experimental points for finding the optimal solution. It can be used to advantage, because not all machine code properties are clearly defined in intermediate code.

It should be noted that the original implementation of the algorithm of linear scan, which uses only one interval of liveness for compilation, worked two times faster than the algorithm of register allocation, based on the number of uses of variables. In order to implement the use of multiple intervals of liveness, it was necessary to implement the more general algorithm. This algorithm slowed down compilation time.

The increased performance of the new algorithms of register allocation happens at all levels, but it is particularly significant on low levels of optimization. In addition, it is much more significant after the use of information provided by static analysis of the space of liveness of variables and after dead-code elimination.

Compared with the interpretation mode, fast compilation with libjit-linear-scan does not increase the time of execution of the virtual machine or the time of compilation, and thus the quality of the machine code improves. The increased performance of the abstract linear scan compared with simple linear scan at optimization level 2 in PnetMark is 5.30%, in SciMark2 – 4.00%, and in Linpack – 12.59%. In addition, improvements were noted in all intermediate benchmarks in PnetMark, SciMark2 and Linpack.

New algorithms do not have any unnecessary computational costs, and additionally use only the information about the number of uses of variables. It should be noted that when simple linear scan is enabled, this information is computed anyway in invocations of the API.

6 Conclusions

The paper studied a new class of rapid register allocation algorithms and presented experimental data on their behavior. Thus, the authors found that the increase of runtime performance for PnetMark is 116%, for SciMark2 – 239%, and 136% for Linpack. As seen from the estimates of standard deviation, the variance of the results in the environment is substantially less than other variants of .NET execution environment of competitors and the previous version of the just-in-time compiler. The results of heuristics application for PnetMark is 105%, 155% for SciMark2, and 93% for Linpack. These results are objective for this hardware platform and operating system. They show the possibility of performing repeated measurements and are similar in quality to the results for other platforms and operating systems. The paper provides the estimation of value deviation of PnetMark, SciMark2, and Linpack benchmarks. The maximum value deviation of PnetMark benchmark is for internal LS-O2. The maximum value deviation of SciMark2 and Linpack benchmarks is for mono 2.0. Thus, the new heuristics of register allocation show improvements in benchmarks performance in comparison with the use of simple heuristics of linear scan. The researchers also found that the performance of SciMark2 and Linpack decreases and this appears to be due to the inefficiency of using only two points of optimization. The conducted experiment confirmed that most of the variables in the intermediate representation of libjit-linear-scan, obtained from the compilation of CIL, have only one interval of liveness.

The results of the performed research may be useful for the development of rapid just-in-time compilers, methods of automatic neutralization of vulnerabilities, converters of bytecodes of virtual machines, and compilers for executing the programs of virtual machines such as Java, Python, Perl, Ruby, LLVM, and Parrot.

We have outlined a theoretical possibility and have demonstrated the possibility of the practical applicability of the proposed approach. We have tested various options and have specified the optimal algorithm implementations, based on the practical use purposes. It is worth noting that the proposed approach is open to improvement, and although we have demonstrated the applicability of the principle and the possibility of the algorithm, the proposed approach at the moment-in a great measure a theoretical possibility. Further research directions could be concerned with various possibilities of dynamic compilation and the dynamic neutralization of data leakages. For example, it is possible to create new domain-oriented algorithms for the optimization of the performance of code and neutralization of threats that target the narrow specifics of the application. The given algorithms can be specialized and optimized for the application in a variety of areas where embedded systems or mainframes can be or are being used. These areas include, for instance, power consumption or various aspects of the secure execution of programs in operating systems for general and specialized purposes.

References

[1] D. D. Niu, L. Liu, X. Zhang, S. Lü, Z. Li. Security analysis model, system architecture and relational model of enterprise cloud services. *International Journal of Automation and Computing*, vol. 13, no. 6, pp. 574–584, 2016. DOI: 10.1007/s11633-016-1014-2.

[2] R. Odaira, T. Nakaike, T. Inagaki, H. Komatsu, T. Nakatani. Coloring-based coalescing for graph coloring register allocation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ACM, Toronto, Canada, pp. 160–169, 2010.

[3] Q. Liang, Y. Z. Wang, Y. H. Zhang. Resource virtualization model using hybrid-graph representation and converging algorithm for cloud computing. *International Journal of Automation and Computing*, vol. 10, no. 6, pp. 597–606, 2013. DOI: 10.1007/s11633-013-0758-1.

[4] K. Kononenko. Libjit linear scan: A model for fast and efficient compilation. *International Review on Modelling & Simulations*, vol. 3, no. 5, pp. 1035–1044, 2010.

[5] K. Kononenko. A unified approach to identifying and healing vulnerabilities in x86 machine code. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, ACM, Istanbul, Turkey, pp. 397–398, 2012. DOI: 10.1145/2348543.2348593.

[6] F. P. Miller, A. F. Vandome, J. McBrewster. Mono (*Software*): *MonoDevelop, Software Patents and Free Software, Novell, Comparison of Application Virtual Machines, DotGNU, Portable .NET, .NET Framework, ⋯ Free and Open Source Software, Ximian.* Alpha Press, 2009.

[7] M. Pandey, S. Sarda. *LLVM Cookbook.* Birmingham, UK: Packt Publishing, 2015.

[8] J. Z. Zhao, S. Nagarakatte, M. M. K. Martin, S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation,* ACM, Seattle, USA, pp. 175–186, 2013. DOI: 10.1145/2499370.2462164.

[9] M. D. Smith, N. Ramsey, G. Holloway. A generalized algorithm for graph-coloring register allocation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation,* ACM, Seattle, USA, pp. 277–288, 2004. DOI: 10.1145/996893.996875.

[10] J. Cocke, J. Markstein. Measurement of code improvement algorithms. In *Proceedings of the IFIP Congress,* Tokyo, Japan, pp. 221–228, 1980.

[11] A. P. Ershov. Alpha-an automatic programming system of high efficiency. ALGOL Bull, France. pp. 19–27, 1965.

[12] J. R. Schwartz. On programming: An interim report on the SETL project. Installment I: Generalities; Installment II: The SETL Language and Examples of Its Use, Technical Report COO-3077-94, New York University, New York, USA, 1975.

[13] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, P. W. Markstein. Register allocation via coloring. *Computer Languages*, vol. 6, no. 1, pp. 47–57, 1981. DOI: 10.1016/0096-0551(81)90048-5.

[14] G. Chaitin. Register allocation and spilling via graph coloring. *ACM SIGPLAN Notices*, vol. 39, no. 4, pp. 66–74, 2004. DOI: 10.1145/989393.989403.

[15] F. C. Chow, J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 4, pp. 501–536, 1990. DOI: 10.1145/88616.88621.

[16] F. Rastello, B. Diouf, A. Cohen. A polynomial spilling heuristic: Layered allocation. In *Proceedings of IEEE/ACM International Symposium on Code Generation and Optimization*, IEEE, Washington DC, USA, pp. 1–10, 2013.

[17] P. K. Krause. Optimal register allocation in polynomial time. In *Proceedings of International Conference on Compiler Construction*, Rome, Italy, pp. 1–20, 2013.

[18] P. K. Krause. The complexity of register allocation. *Discrete Applied Mathematics*, vol. 168, pp. 51–59, 2014. DOI: 10.1016/j.dam.2013.03.015.

[19] R. C. Lozano, M. Carlsson, F. Drejhammar, C. Schulte. Constraint-based register allocation and instruction scheduling. In *Proceedings of the 18th international conference on Principles and Practice of Constraint Programming,* Quebec City, Canada, pp. 750–766, 2012. DOI: 10.1007/978-3-642-33558-7_54.

[20] V. Sarkar, R. Barik. Extended linear scan: An alternate foundation for global register allocation. In *Proceedings of the 16th International Conference on Compiler Construction*, Springer-Verlag, Braga, Portugal, pp. 141–155, 2007.

[21] O. Traub, G. Holloway, M. D. Smith. Quality and speed in linear-scan register allocation. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, Montreal, Canada, pp. 142–151, 1998. DOI: 10.1145/277652.277714.

[22] C. Wimmer, M. Franz. Linear scan register allocation on SSA form. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ACM, Toronto, Canada, pp. 170–179, 2010. DOI: 10.1145/1772954.1772979.

[23] G. Calinescu, M. M. Li. Register loading via linear programming. *Algorithmica*, vol. 72, no. 4, pp. 1011–1032, 2015. DOI: 10.1007/s00453-014-9888-2.

[24] I. H. R. Jiang, G. J. Nam, H. Y. Chang, S. R. Nassif, J. Hayes. Smart grid load balancing techniques via simultaneous switch/tie-line/wire configurations. In *Proceedings of IEEE/ACM International Conference on Computer-aided Design*, IEEE, San Jose, pp. 382–388, 2014. DOI: 10.1109/ICCAD.2014.7001380.

[25] F. M. Quintão Pereira, J. Palsberg. Register allocation by puzzle solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, Tucson, USA, pp. 216–226, 2008. DOI: 10.1145/1379022.1375609.

[26] F. M. Quintão Pereira. Register allocation by puzzle solving, Ph. D. dissertation, University of California, USA, 2008.

[27] R. Barik, J. S. Zhao, V. Sarkar. A decoupled non-SSA global register allocation using bipartite liveness graphs. *ACM Transactions on Architecture and Code Optimization*, vol. 10, no. 4, Article number 63, 2013. DOI: 10.1145/2544101.

[28] Q. Colombet, B. Boissinot, P. Brisk, S. Hack, F. Rastello. Graph-coloring and treescan register allocation using repairing. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ACM, Taipei, Taiwan, China, pp. 45–54, 2011. DOI: 10.1145/2038698.2038708.

[29] H. B. Rong. Tree register allocation. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, New York, USA, pp. 67–77, 2009. DOI: 10.1145/1669112.1669123.

[30] J. Eisl. Trace register allocation. In *Proceedings of ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, ACM, Pittsburgh, USA, pp. 21–23, 2015. DOI: 10.1145/2814189.2814199.

[31] D. Callahan, B. Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, Toronto, Canada, pp. 192–203, 1991. DOI: 10.1145/113446.113462.

[32] K. D. Cooper, A. Dasgupta, J. Eckhardt. Revisiting graph coloring register allocation: A study of the Chaitin-Briggs and Callahan-Koblenz algorithms. In *Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing*, Hawthorne, USA, pp. 1–16, 2006. DOI: 10.1007/978-3-540-69330-7_1.

[33] M. Mohr, A. Grudnitsky, T. Modschiedler, L. Bauer, S. Hack, J. Henkel. Hardware acceleration for programs in SSA form. In *Proceedings of International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, IEEE, Montreal, Canada, Article number 14, 2013.

[34] P. K. Krause. Bytewise register allocation. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, ACM, Sankt Goar, Germany, pp. 22–27, 2015. DOI: 10.1145/2764967.2764971.

[35] F. Bouchez. A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases, Ph. D. dissertation, École Normale supérieure de Lyon, Lyon, France, 2009.

[36] S. Hack, Register allocation for programs in SSA form, Ph. D. dissertation, University of Karlsruhe, Germany, 2007.

[37] B. Boissinot, F. Brandner, A. Darte, B. D. de Dinechin, F. Rastello. A non-iterative data-flow algorithm for computing liveness sets in strict SSA programs. In *Proceedings of the 9th Asian Conference on Programming Languages and Systems*, Kenting, Taiwan, China, pp. 137–154, 2011. DOI: 10.1007/978-3-642-25318-8_13.

[38] Q. Colombet, F. Brandner, A. Darte. Studying optimal spilling in the light of SSA. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ACM, Taipei, Taiwan, China, pp. 25–34, 2011. DOI: 10.1145/2038698.2038706.

[39] B. Boissinot, P. Brisk, A. Darte, F. Rastello. SSI properties revisited. *ACM Transactions on Embedded Computing Systems*, vol. 11S, no. 1, Article number 21, 2012. DOI: 10.1145/2180887.2180898.

[40] P. Brisk, M. Sarrafzadeh. Interference graphs for procedures in static single information form are interval graphs. In *Proceedings of the 10th International Workshop on Software & Compilers for Embedded Systems*, ACM, Nice, France, pp. 101–110, 2007. DOI: 10.1145/1269843.1269858.

[41] A. F. Deon, Y. A. Menyaev. The complete set simulation of stochastic sequences without repeated and skipped elements. *Journal of Universal Computer Science*, vol. 22, no. 8, pp. 1023–1047, 2016. DOI: 10.3217/jucs-022-08-1023.

[42] A. F. Deon, Y. A. Menyaev. Parametrical tuning of twisting generators. *Journal of Computer Science*, vol. 12, no. 8, pp. 363–378, 2016. DOI: 10.3844/jcssp.2016.363.378.

[43] D. E. Knuth. *Art of Computer Programming, volume 2: Seminumerical Algorithms*. Boston, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[44] P. Cousot, R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming*, Dunod, France, pp. 106–130, 1976.

[45] P. Cousot, R. Cousot. Abstract interpretation: Past, present and future. In *Proceedings of Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science*, ACM, Vienna, Austria, Article number 2, 2014. DOI: 10.1145/2603088.2603165.

[46] P. Cousot, R. Cousot, L. Mauborgne. Theories, solvers and static analysis by abstract interpretation. *Journal of the ACM*, vol. 59, no. 6, Article number 31, 2012. DOI: 10.1145/2395116.2395120.

[47] J. R. Dick, K. B. Kent, J. C. Libby. A quantitative analysis of the .NET common language runtime. *Journal of Systems Architecture*, vol. 54, no. 7, pp. 679–696, 2008. DOI: 10.1016/j.sysarc.2007.11.004.

[48] T. Davies, C. Karlsson, H. Liu, C. Ding, Z. Z. Chen. High performance linpack benchmark: A fault tolerant implementation without checkpointing. In *Proceedings of International Conference on Supercomputing*, ACM, Tucson, USA, pp. 162–171, 2011. DOI: 10.1145/1995896.1995923.

Shahrzad Kananizadeh received the B. Sc. degree in cyber security from University of Tubingen, Germany. She is currently a researcher at Saarland University, Germany and works in Robert Bosch, Germany.

Her research interests include programming languages, compilers and program analysis. Her areas of expertise include computer engineering, communication engineering and telecommunications.

E-mail: kananiz_sh@hotmail.com
ORCID iD: 0000-0003-3226-5129

Kirill Kononenko is a mathematician who is interested in the theoretical and mathematical foundations of computer science. He is also interested in linguistics and foreign languages. He developed the libjit-linear-scan library for dynamic compilation. He is a member of ACM and IEEE.

His research interests include theoretical physics, stochastic processes and mathematical logic.

E-mail: kirill.kononenko@acm.org (Corresponding author)
ORCID iD: 0000-0002-7882-6326