# Dynamic Data-dependence Analysis in SAPFOR

Nikita Kataev[1][0000-0002-7603-4026], Alexander Smirnov[1][0000-0002-2971-4248]
Andrey Zhukov[2][0000-0001-9018-2941]

[1] Keldysh Institute of Applied Mathematic RAS, Miusskaya sq., 4, 125047, Moscow, Russia
[2] Lomonosov Moscow State University, GSP-1, Leninskie Gory, 11999, Moscow, Russia
kaniandr@gmail.com

**Abstract.** The possibilities of static program analysis are often insufficient to investigate real-world applications. Complicated control-flow graph and memory access patterns lead to a conservative assumption of loop-carried data dependencies. To make decisions about the possibility to parallelize loops in a program, SAPFOR implemented a dynamic data dependence analysis. This analysis is based on the instrumentation of the LLVM representation of the analyzed programs and can be performed for applications in the C and Fortran languages. The use of static analysis allows to reduce the number of analyzed memory accesses and to ignore scalar variables, which can be explored in a static way. A selective analysis of program functions and loops is also allowed. These features allow us to significantly reduce the overhead of the program execution time, while maintaining the completeness of the analysis. The developed tool was tested on performance tests from the NAS Parallel Benchmarks (NPB) package for C and Fortran languages. The implementation of dynamic analysis, in addition to traditional types of data dependencies (flow, anti, output), allows us to determine privatizable variables and a possibility of loop pipelining. Together with the capabilities of DVM and OpenMP these greatly facilitates program parallelization and simplify insertion of the appropriate compiler directives.

**Keywords:** Program Analysis, Dynamic Analysis, Semi-automatic Parallelization, SAPFOR, DVM, LLVM.

## 1 Introduction

The main goal of SAPFOR [1, 2] (System FOR Automated Parallelization) development is to reduce the complexity of parallel programming. Programming across diverse architectures leads to the complicated combined use of various parallel programming models (MPI, SHMEM, OpenMP, CUDA, OpenACC, OpenCL). The SAPFOR system uses the DVMH languages (Fortran-DVMH and C-DVMH) included in the DVM system as the target programming language [3, 4]. These languages propose an interface which hides features specific to various parallel programming models and facilitates both manual and semi-automatic development of parallel programs. The SAPFOR project combines three main lines of research: program analysis, automatic parallelization of "well-formed" sequential programs and semi-automatic transformation of a program to obtain its well-formed version. Each of these lines, both individually and as a whole, is called upon to assist in the development of parallel programs. All of the three mentioned lines of research are based on the investigation of program properties. Thus, program analysis is an essential part of any parallelization process, not necessarily automatic.

SAPFOR provides both static and dynamic analysis of programs. Separately, each type of analysis is not enough to investigate parallelization opportunities. On the one hand, static approach in many cases is too conservative and it reports the presence of really missing dependencies in order to maintain the correctness of the program. The authors of [5] explore the possibilities of automatic parallelization using Intel and PGI compilers on the example of a set of scientific kernels OmpSRC [6]. They identify at least two main problems that prevent static analysis of these benchmarks: the use of pointers and a complex control flow graph. In both cases, compilers are forced to make conservative decisions about dependencies in the program. On the other hand, dynamic analysis, firstly, analyzes the program for a certain set of input data, and, secondly, it is resource intensive, that is, it may have major time and memory overheads.

In addition to the fact whether data dependence exists it is important to determine the possible ways to eliminate it. For example, anti and output dependencies can be eliminated by data privatization (i.e. creating a local copy of the data for each process or thread). Corresponding static analysis of scalar variables based on data flow analysis is implemented in SAPFOR [7]. However, array privatization requires exploration of the sets of elements which are accessed at each loop iteration. Pointer-based and indirect accesses, the dependence of subscript expressions on function parameters and other values identifiable only at runtime make it impossible to statically determine privatizable arrays. Moreover, the elimination of such dependencies is one of the key transformations required to parallelize benchmarks from the NAS Parallel Benchmarks (NPB) [8, 9]. Another source of parallelism is flow loop-carried dependencies with distance limited by a constant. Pipelined execution of corresponding loops is possible. The DVM system provides *across* directive to reveal such loops and its specification requires the indication of a dependence distances. To overcome these analysis issues, a dynamic data dependence analysis was implemented in the SAPFOR system.

## 2 Existing Approaches to Dynamic Analysis

Dynamic analysis has the following main shortcomings. It depends on the completeness of the input and it also has high memory and runtime overheads. The degree of representativeness of the input affects significantly the reliability of dynamic analysis. The use of poorly selected data sets can lead to the omission of existing dependencies. In this sense, dynamic analysis is possible only under the close control of the user who parallelizes a program, and it is convenient for use in semi-automatic parallelization systems such as SAPFOR.

To obtain the program properties at runtime the following two approaches are widely used: sampling profilers evaluate program execution at regular time intervals, and instrumentation-based analysis allows us to study the program behavior in predefined points. Instrumentation technique inserts calls to some external library into the program code to collect necessary information. Sampling profilers less demand on the consumed resources than instrumentation-based analysis but they may suffer from the loss of information, so sampling profilers are not widely used for data-dependence analysis.

The use of instrumentation is the most suitable approach to determine data dependencies. In turn, it is possible to insert calls in a source code, in some intermediate representation or in compiled executables (binary instrumentation). Instrumentation of programs at the source level requires a separate implementation of tools for each supported programming language (in the case of SAPFOR, these are Fortran and C languages). It is necessary to process separately all possible syntactic constructions of supported languages. In addition, preliminary program transformation may reduce overhead costs in certain cases. These kinds of transformations are convenient to perform on some intermediate representation of the program which is common for different languages (for example, LLVM IR). Binary instrumentation is the most effective in terms of completeness of coverage of the executable program, as it allows instrumentation even in the absence of source codes. For example, binary instrumentation allows us to analyze already compiled modules for which other instrumentation techniques are not applicable. The disadvantage is that to establish relation between the received information and the source code of the analyzed program complete source-level debug information is required. It should be mentioned that these required metadata could be lost during program compilation.

The SAPFOR system uses LLVM intermediate representation (IR) to perform program analysis (both static and dynamic) [10]. LLVM IR is common for various programming languages. LLVM supports the ability to perform additional analysis and transformations to selectively instrument a program, many of which are already implemented in LLVM. The metadata available in LLVM IR can be further extended to fully describe the source program. To achieve this goal SAPFOR establishes a correspondence between certain constructions of the source program represented in the form of a syntax tree (AST) and LLVM IR constructs (loops, variables, functions and their calls). LLVM IR simultaneously exists in three forms: the structure of C ++ 11 language classes, binary and textual representation. A human readable representation provides natural opportunities to debug and visualize the transformations. The disadvantage is that pre-compiled sections of the program cannot be instrumented and must be subjected to conservative analysis.

One of the most widely used tools for dynamic data dependence analysis is Intel Advisor, which is a part of Intel Parallel Studio [11]. This tool allows you to determine three types of dependencies: flow (RAW), anti (WAR), output (WAW). A more detailed classification cannot be performed. For example, this tool cannot reveal variables that can be privatized, it also does not determine the dependence distances. For analysis, the binary instrumentation is used, therefore, in order to correctly correlate the obtained results with the source code, it is recommended to disable optimizations.

To start the analysis of data dependencies, you must first obtain the survey report. Then the loops which are presented in this report could be marked for further analysis. To obtain a survey report, sampling profiler is used, so the number of detected loops depends on the size of the interval that is used to collect the data. But even for the minimum interval, not all loops will be recognized, for example, for the LU program (class S) with the minimum interval only one third of the loops will be detected (about 60 loops out of 187 program loops). This is primarily due to the fact that input of the S class is a very small data set for test purposes on which the program runs for a split second. However, the specified data set describes the whole possible behavior of the program.

Table 1 shows the Intel Advisor slowdown for the LU program (class S), as well as the number of loops available for the analysis with a minimum sampling interval. Slowdown is indicated as the ratio of execution times for programs compiled with different optimization options (-O0 and -O3) and with enabled and disabled data dependence analysis. It is worth noting that the recommended analysis mode requires the use of the -O0 option to ensure analysis of the source program without optimizations and to ensure exact correspondence of the output information to the source code objects. In this case, the slowdown is about 4160 times, despite the fact that only a third of all program loops have been be analyzed.

Test has been executed on Intel Xeon CPU E5-1660 v2, 3.70 GHz, with Turbo Boost disabled. To compile and analyze programs, Intel Parallel Studio 2019 was used based on the GCC 7.4 system libraries.

**Table 1.** Intel Advisor 2019 Slowdown in LU (NPB) Benchmark Analysis (Class S)

| Optimization | -O0 (enable analysis) | -O3 (enable analysis) |
|---|---|---|
| -O0 (disable analysis) | 850 times / 62 loops | 433 times / 27 loops |
| -O3 (disable analysis) | **4160 times / 62 loops** | 2384 times / 27 loops |

The basic idea a dynamic analysis algorithm implemented in SAPFOR is similar to the pairwise method described in [12]. The authors propose improvements to this algorithm to reduce memory and time overhead, but we were not able to access the tool they developed. The authors rely on the binary instrumentation, although they talk about the possibility of using LLVM. In addition, dynamic analysis was performed only on the most resource-intensive loops. Therefore, it is rather difficult to estimate the real costs when analyzing the entire program (the total number of loops in the analyzed programs is not given). Some small loops with low execution time could be omitted in case of parallelization for systems with shared memory. However, SAPFOR needs to make global decisions about the distribution of data and computations for heterogeneous computational cluster. Therefore, it is impossible to exclude any loops from the analysis, since they can access the distributed data (including indirect accesses which could not be tracked with static approach).

## 3 Dynamic Analysis Algorithm Description

The dynamic analysis in SAPFOR should analyze all loops and detect for each loop the type of data dependence and the possibility of its elimination. To determine the possibility of pipelined execution of a loop, the dependence distance (maximum and minimum) should be determined. It is also necessary to identify whether dependence is spurious and whether data privatization allows it to be eliminated. Since it is not necessary to discover the specific operators that produce data dependencies, a list of the whole memory accesses is not required. Instead, it is possible to store a flag which indicates that a memory location with a specified address has been accessed. Some additional information useful to identify the desired properties also should be collected. This means that simple processing of accumulated data should be performed on each memory access. This approach allows us to reduce the memory overhead if at each loop iteration multiple accesses occur at the same address. The time of dynamic analysis changes in an unobvious way. On the one hand, at the moment of memory access there is a data processing, which slows down the execution. On the other hand, when the control flow exits this loop the number of processed data is decreased. Hence, the analysis time greatly depends on the structure of the analyzed program.

For each nested loop there is a separate data structure which describes memory accesses in this loop. On each memory access its description is updated for the closest loop nest. After the loop exit the accumulated information is used to update appropriate data structure for the outer loop and dependencies found for the inner loop are also stored in the global storage.

The dynamic analyzer uses two main data structures: a global storage of analysis results and stack of execution contexts. The global storage contains list of all memory accesses in the loop and a description of all dependencies found for each loop. The description comprises of dependence kind (flow, anti, output), the dependence distance (maximum and minimum) and a flag which indicates the possibility of data privatization. An execution context is a set of memory addresses in conjunction with additional information necessary to identify properties of interest. Upon entering the loop or function, a new empty context is created and pushed to the stack. The context stores the number of the current iteration of the corresponding loop. In the case of a function the iteration number does not matter and has undefined value. At the beginning of loop iteration a special function from the analyzer runtime library is called. This function changes the number of iteration in the appropriate context. On each memory access the dynamic analyzer looks for the description of the accessed address. If there is no any description then the new one is created in the top context. For example, in case of reading there is a description of an accessed address in the top context, and this description indicates that the last reading is made at the current iteration. If there was a record that indicates write access at another iteration, then the fact of flow dependence is established and the distance between iterations is calculated. To calculate the distance, it is necessary to store not only the iteration number of the last read/write, but also the iteration number of the first read/write.

When exit from a loop or function occurs, the top context is removed from the stack. For each address from the removed context, if there was a read or write access to this address, the corresponding operation is fixed at the new top of the stack. For a loop related to a removed context information about dependencies is added to the global storage.

At the time of program finalization, information from the global storage is printed in the specified format.

## 4 Implementation Details

The runtime library of dynamic analyzer is implemented in C ++ 11, to make instrumentation LLVM pass has been implemented. This pass inserts in LLVM IR function calls to the runtime library. For each instrumented object, its description based on debug information is passed to the dynamic analyzer. This information is used to identify the source code object corresponding to the instrumented lower level object. Instrumentation is supported for both Fortran and C/C ++ programs. LLVM IR is common for diverse source languages, but the debug information may vary slightly. For example, structures of a special kind are used to describe Fortran COMMON blocks. Therefore, for other languages, a high-level description of instrumented data may be lost.

Dynamic analysis comprises the following sequence of steps:

1. Construction of LLVM representation for each file that should be analyzed.
2. Linking of LLVM bitcode files together into a single LLVM bitcode file. At this step LLVM tool *llvm-link* is used.
3. Instrumentation of the single file obtained in the previous step.
4. Further compilation of the instrumented file. It should be combined with the rest of the analyzed project, as well as with the runtime library of dynamic analyzer.

The output of the instrumented program is either a textual list of data dependencies in analyzed loops, or a description of the dependencies in JSON format. The resulting JSON file can be passed to the static analyzer of SAPFOR to refine the analysis results.

Instrumentation pass makes the following IR-level transformations:

- inserts declarations of functions of the dynamic analyzer,
- inserts calls to these functions into the function bodies of the instrumented module,
- declares a global pool to store meta-information and to identify each instrumented object,
- creates internal functions to initialize meta-information and to register types and global objects,
- inserts calls to initialization functions at the beginning of the program entry point.

Calls to runtime library functions allow dynamic analyzer to handle memory accesses, calls to functions, the beginning of the whole loop, as well as the beginning of each loop iteration and exits from loop. Correspondence between actual and formal parameters is also established.

## 5 Evaluation

The implemented tool was tested on the Fortran [8] and C [9] versions of the NAS Parallel Benchmarks (NPB). Estimation of the time overhead is shown in Fig. 1 and Fig. 2. The growth of the consumed memory (in the number of times) is shown in Fig. 3 and Fig. 4.

Benchmarks have been executed on Intel Xeon CPU E5-1660 v2, 3.70 GHz, with Turbo Boost disabled. To obtain LLVM IR and compile programs, the Clang and Flang compilers version 7.1.0 were used, based on the GCC 7.4 system libraries. Compilation was performed using the –O3 option. In contrast to the use of binary instrumentation, the use of this option is allowed, since the optimizations are applied after the instrumentation of the original code. Memory consumption and time overhead have been also measured for Intel Advisor. In this case, the dependency analysis was performed with the –O0 option to presume reliable results. However, slowdown is evaluated relative to programs compiled with the –O3 option.

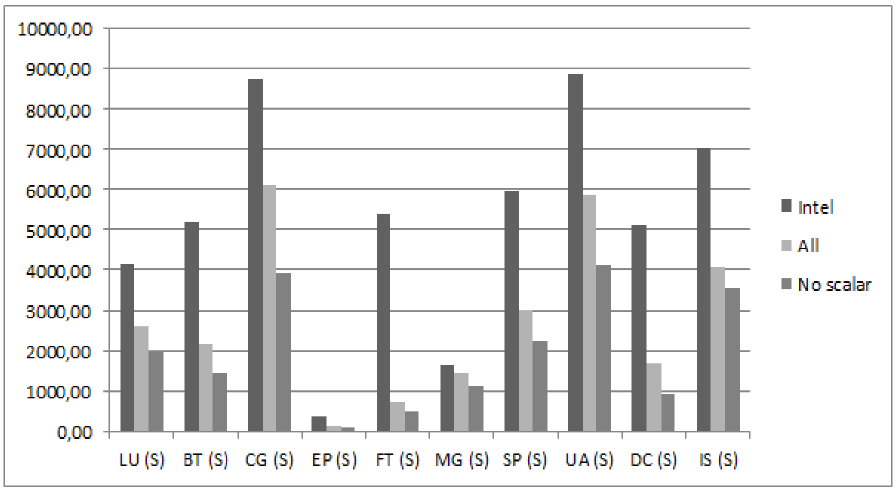

**Fig. 1.** Slowdown (number of times in relation to the original sources) of the C version of the NAS Parallel Benchmarks with full (All) and partial (No scalar) instrumentation and its comparison with time overhead of Intel Advisor.

In order to reduce memory consumption and time overhead, static analysis tools implemented in SAPFOR were used [7]. Scalar variables, which only have loads and stores as uses, could be analyzed with static analysis techniques in most cases. In case of data dependencies SAPFOR determines whether privatization techniques are applicable to these scalar variables. Our static analysis tool also reveals reduction variables in a source code. Dynamic analyzer can safely ignore accesses to these variables. Moreover, accesses to these variables can be optimized by using LLVM tools, for example, the corresponding low-level memory references can be promoted to be register references. The application of this optimization allows us to reduce the slowdown of the analyzed programs up to 40% (in the case of EP benchmark). Thus, taking into account this optimization, the execution time increases on average up to 2000 times. But together with the application of static analysis, a complete analysis of the entire program is provided. At the same time, analysis of the one third of all loops performed by Intel Advisor slows down the program by an average of 5,000 times.

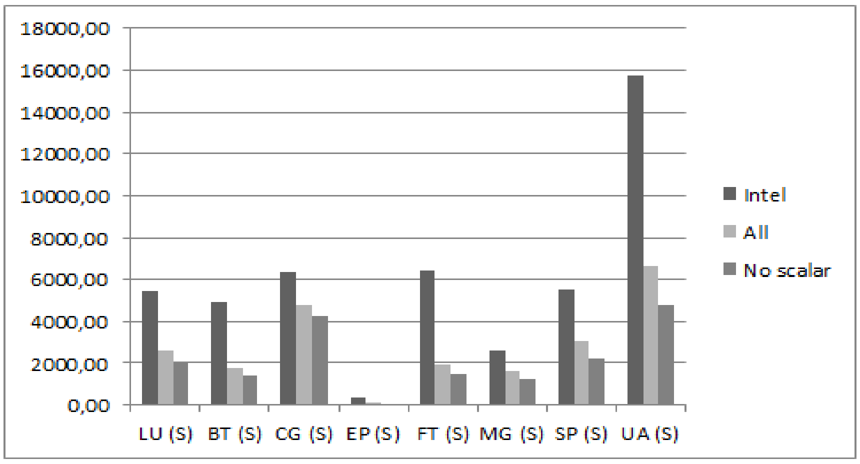

**Fig. 2.** Slowdown (number of times in relation to the original sources) of the Fortran version of the NAS Parallel Benchmarks with full (All) and partial (No scalar) instrumentation and its comparison with time overhead of Intel Advisor.

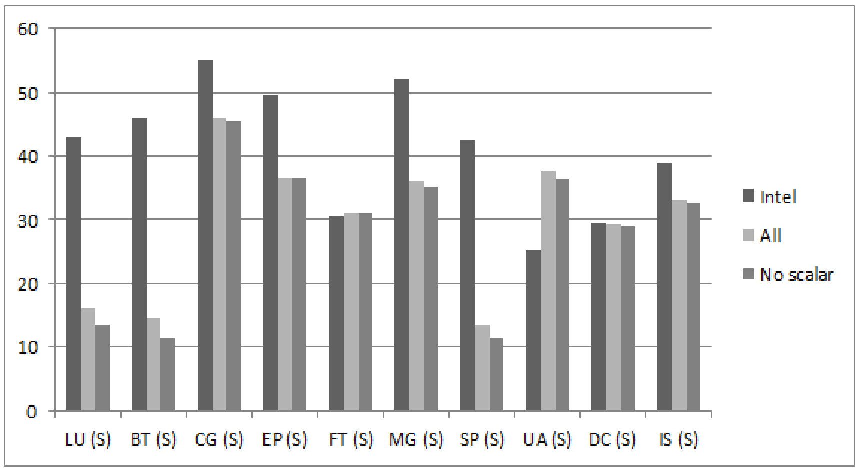

**Fig. 3.** Memory consumption (number of times in relation to the original sources) of the C version of the NAS Parallel Benchmarks with full (All) and partial (No scalar) instrumentation and its comparison with memory overhead of Intel Advisor.

We also implement special command line options which allow the user to select functions which should be analyzed. In this case, the instrumentation is performed for specified functions, as well as for functions called from these ones. Such selective analysis is used to explore properties which are necessary to parallelize a specific region of code and significantly speeds up program analysis. For example, for the LU benchmark (class S), in case of exploration of one of the main loops the slowdown is of 707 times only. Dynamic analysis of this loop highlights a regular data dependence, which can be eliminated by pipelined execution of the loop.

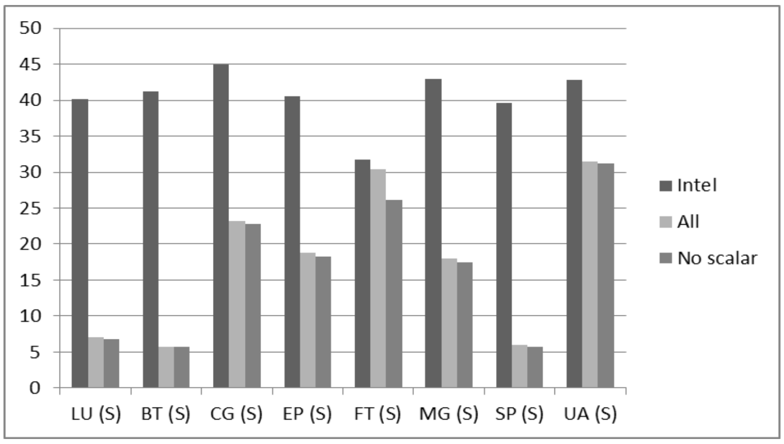

**Fig. 4.** Memory consumption (number of times in relation to the original sources) of the Fortran version of the NAS Parallel Benchmarks with full (All) and partial (No scalar) instrumentation and its comparison with memory overhead of Intel Advisor.

## 6 Conclusions

The paper considers a dynamic analysis tool designed to determine data dependencies. It is implemented in SAPFOR system. This tool can be used both to obtain analysis results for the purpose of semi-automatic parallelization of programs in SAPFOR, and for the purpose of the manual program parallelization. It determines the main types of data dependencies (flow, anti, output) and collects all variables accessed in program loops. It also recognizes variables which can be privatized. It is especially important to note that privatizable arrays are determined. Static approaches have limited capabilities to analyze such arrays. Our approach also allows us to reveal a possibility of pipelined execution of loops. Moreover, the computed data dependence distances are enough to insert the *across* directive which is a part of DVMH languages and which enables the pipelined execution of marked loops. Preliminary static analysis of scalar variables reduces the overhead of dynamic analysis without losing the completeness of the results. Instrumentation of LLVM representation instead of compiled executable (binary instrumentation) makes it possible to optimize the program after transformation without losing the accuracy of the analysis results. Future works involve further reduction of runtime overhead since parallelization for HPC systems with distributed memory requires analysis of the entire program.

The source code for the SAPFOR system is available on GitHub [13].

This work was partially supported by Presidium RAS, program I.26 "Fundamentals of creating algorithms and software for advanced ultra-high performance computing".

## References

1. Klinov, M.S., Kriukov, V.A.: Avtomaticheskoe rasparallelivanie Fortran-programm. Otobrazhenie na klaster. Vestnik Nizhegorodskogo universiteta im. N.I. Lobachevskogo, No. 2, pp. 128–134 (2009), last access 05.12.2019.
2. Bakhtin, V.A., Zhukova, O.F., Kataev, N.A., Kolganov, A.S., Kriukov, V.A., Podderiugina N.V., Pritula, M.N., Savitskaia, O.A., Smirnov, A.A.: Avtomatizatsiia rasparallelivaniia programmnykh kompleksov. Nauchnyi servis v seti Internet: trudy XVIII Vserossiiskoi nauchnoi konferentsii (19-24 sentiabria 2016 g., g. Novorossiisk). M.: IPM im. M.V. Keldysha, pp. 76–85 (2016). https://doi.org/10.20948/abrau-2016-31, last access 05.12.2019.
3. Konovalov, N.A., Krukov, V.A, Mikhajlov, S.N., Pogrebtsov, A.A.: Fortan DVM: a Language for Portable Parallel Program Development. Programming and Computer Software. vol. 21, No. 1, pp. 35–38 (1995).
4. Bakhtin, V.A., Klinov, M.S., Kriukov, V.A., Podderiugina, N.V., Pritula, M.N., Sazanov, Iu.L.: Rasshirenie DVM-modeli parallelnogo programmirovaniia dlia klasterov s geterogennymi uzlami. Vestnik Iuzhno-Uralskogo gosudarstvennogo universiteta, seriia "Matematicheskoe modelirovanie i programmirovanie", №18 (277), vypusk 12. Cheliabinsk: Izdatelskii tsentr IuUrGU, S. 87–92 (2012).
5. Kim, M., Kim, H., Luk, C.K.: Prospector: A dynamic data-dependence profiler to help parallel programming. HotPar’10: Proceedings of the USENIX workshop on Hot Topics in parallelism (2010).
6. Dorta, A.J., Rodríguez, C., de Sande, F., and Gonzalez-Escribano, A.: The OpenMP Source Code Repository. Parallel, Distributed, and Network-Based Processing, Euromicro Conference (2005).
7. Kataev, N.A.: Application of the LLVM Compiler Infrastructure to the Program Analysis in SAPFOR. Voevodin V., Sobolev S. (eds) Supercomputing. RuSCDays 2018. Communications in Computer and Information Science, vol 965. Springer, Cham. Pp. 487–499 (2018), https://doi.org/10.1007/978-3-030-05807-4_41, last access 05.12.2019.
8. NAS Parallel Benchmarks. UFL. https://www.nas.nasa.gov/publications/npb.html, last access 05.12.2019.
9. Seo, S., Jo, G., Lee, J.: Performance Characterization of the NAS Parallel Benchmarks in OpenCL. 2011 IEEE International Symposium on. Workload Characterization (IISWC), pp. 137–148 (2011).
10. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO’04). Palo Alto, California (2004).
11. Intel Parallel Studio. https://software.intel.com/en-us/parallel-studio-xe, last access 05.12.2019.
12. Kim, M., Kim, H., Luk, CK.: SD3: A Scalable Approach to Dynamic Data-Dependence Profiling. 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture. IEEE (2011), https://doi.org/doi:10.1109/MICRO.2010.49, last access 05.12.2019.
13. SAPFOR. https://github.com/dvm-system, last access 05.12.2019.