DOI: 10.15514/ISPRAS-2022-34(4)-1

Case study: Source code static analysis for performance issues detection

[1] A.Y. Gerasimov, ORCID: 0000-0001-9964-5850 <gerasimov.alexander@huawei.com>
[1] A.A. Kanakhin, ORCID: 0000-0000-0000-000 <kanakhin.alexey@huawei.com>
[1] P.A. Privalov, ORCID: 0000-0002-8939-5824 <petr.privalov@huawei.com>
[2] A.A. Zhukov, ORCID: 0000-0002-2788-4542 <andrey.zhukov@huawei-partners.com>
[2] E.A. Kaminsky, ORCID: 0000-0002-5040-0999 <evgeny.kaminsky1@huawei-partners.com>

[1] Chong-Ming Software and Technology Center, Huawei Technologies Co. Ltd.,
17k2, Krylatskaya st., Moscow, Russia, 121614
[2] Coleman Services,
Bld. 2, Shchipok st. 5/7, Moscow, Russia, 155054

Abstract. Source code static analysis is widely used for program errors detection. Mostly it is used for finding critical issues like security vulnerabilities, critical program defects leading to runtime errors like crash and unexpected behavior of programs. Many SCSA tools are used for checking code conformance to different coding style guides. In this case study we present results of applying SCSA techniques for checking performance coding rules of Huawei and evaluate whether manually fixing found issues in accordance with the guidelines could impact performance, or if the compiler already applies all necessary optimizations during compilation.

Keywords: source code static analysis; program performance; compilers

For citation: Gerasimov A.Y., Kanakhin A.A., Privalov P.A., Zhukov A.A., Kaminsky E.A. Case study: source code static analysis for performance issues detection. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 4, 2022. pp. 7-20. DOI: 10.15514/ISPRAS-2022-34(4)-1

Acknowledgements. This paper presents a work of the team. Authors has done a contribution to the paper, but Cooddy development team responsible for engine and checkers implementation and ideas and code review should be pointed out: Pavel Mezhuev, Aleksey Demidov, Veronika Butkevich, Natalya Chernova, Damir Gimatdinov.

Применение статического анализа исходного кода для поиска проблем с производительностью: примеры из практики

[1] А.Ю. Герасимов, ORCID: 0000-0001-9964-5850 <gerasimov.alexander@huawei.com>
[1] А.А. Канахин, ORCID: 0000-0001-9800-2722 <kanakhin.alexey@huawei.com>
[1] П.А. Привалов, ORCID: 0000-0002-8939-5824 <petr.privalov@huawei.com>
[2] А.А. Жуков, ORCID: 0000-0002-2788-4542 <andrey.zhukov@huawei-partners.com>
[2] Е.А. Каминский, ORCID: 0000-0002-5040-0999 <evgeny.kaminsky1@huawei-partners.com>

[1] ООО "Техкомпания Хуавэй",
Россия, 121614, Москва, ул. Крылатская 17к2
[2] Coleman Services,
Россия, 155054, Москва, ул. Щипок 5/7, стр. 2

Аннотация. Статический анализ исходного кода программ широко используется для обнаружения ошибок. В основном он используется для обнаружения критических недостатков программ, таких как уязвимости безопасности, критических ошибок времени исполнения, таких как разрушение программы и неожиданное поведение. Многие инструменты статического анализа кода программ используются для проверки кода программ на соответствие правилам кодирования. В этой работе мы представляем результаты применения техник анализа кода программ для обнаружения ошибок производительности из руководства по программированию производительных программ компании Huawei и результаты проверки, влияет ли исправление программы в соответствии с этими правилами на результирующую производительность программ, или компилятор в состоянии автоматически оптимизировать программу.

Ключевые слова: статический анализ исходного кода; производительность программ; компиляторы

Для цитирования: Герасимов А.Ю., Канахин А.А., Привалов П.А., Жуков А.А., Каминский Е.А. Применение статического анализа исходного кода для поиска проблем с производительностью: примеры из практики. Труды ИСП РАН, том 34, вып. 4, 2022 г., стр. 7-20. 10.15514/ISPRAS-2022-34(4)-1

Благодарности: Эта статья представляет результат работы команды исследователей. Авторы статьи внесли основной вклад в её написание, но также должен учитываться вклад участников команды исследований и разработки в создание ядра и анализаторов инструмента Cooddy, инспекцию кода и идей, положенных в основу реализации инструмента: Павла Межуева, Алексея Демидова, Вероники Буткевич, Натальи Черновой и Дамира Гиматдинова.

1. Introduction

Programming languages like C and C++ are commonly used in performance-critical applications. Both of them are compiled languages—they pass through a compilation and an optimization step before being assembled into an executable binary. Early on, compilers were not proficient enough to optimize some code constructs like double checks and repeated calculations. To account for this, various coding guidelines placed the burden of this optimization on programmers, which sometimes affected code readability. Today, compiler optimization capabilities are much wider because of the evolution of their optimization algorithms and an increased performance budget for compilation. As an example, GCC, which is the most widely used C++ compiler at the time of writing, provides over 100 distinct optimization flags [1-3].

In spite of the advancements in automatic optimization, there are still cases where manual optimization is required to achieve maximum performance. In this paper, we demonstrate our program for automatic source code analysis capable of diagnosing for many of the rules from various coding guides, especially Huawei coding guidelines. Our research goal is to evaluate whether changing source code in accordance with analysis results could visibly impact performance on a computation-heavy open source project. This evaluation takes into account that optimizations made by a modern compiler do not need to be manually implemented by the developer, and an additional goal is to review each category of issues and find out whether it is already optimizable by the compiler. As our compiler of choice we selected GCC over other compilers because it is the most widely used inside the industry.

2. Experiment

As our target for analysis we wanted to choose a C++-based, open-source, command-line application which performs a lot of calculations. Firstly, an application with a command-line interface and no graphical user interface would be easy to analyze and measure performance of. Secondly, an application performing a lot of calculations (as opposed to spending most of the time waiting for user input, sending/answering web requests, etc.) could be significantly optimized by changing the source code to reduce redundant data copies, redundant loop calculations, cache misses and other time sinks under programmer's control.

After some consideration we settled on a project called “Yosys”, a framework for Verilog RTL synthesis (i.e. synthesis of a logic circuit based on some specification of how such a circuit should operate; such specification is written at the register transfer level (RTL)) [4]. In addition to the requirements outlined above, we chose "Yosys" due to a high number of detected issues by our application.

Although 5210 issues were found, most of these were not worth analyzing in detail, either because they were caused by a common problem or because they were obvious false positives (FPs). After an initial filtering we narrowed the amount of interesting issues down to 1845. For review we have split found issues into categories based on the kind of problem they describe. Each category was assessed separately to determine whether the detected issue affects performance in a meaningful way, and if so, if it is optimized by the compiler. For comparing source code, as compiled into assembler, we used “Compiler Explorer”, a widely used web tool for inspecting results of compilation [5]. Unless specified otherwise, all ASM examples are compiled with GCC 12.1, with `-O2` optimizations.

3. Results

When describing results, each section is titled after a particular class of issue, with our internal detect class name in parenthesis for later reference.

3.1 Replace multiple if-else statements with a switch statement (RedundantMultipleIfElseChecker)

The guideline states that a tree of if-else conditions should be replaced with a switch statement where possible.

A switch statement is syntactically simpler than an equivalent if-else tree. First, using a switch ensures that each branch is taken based on an equality comparison with the same value or expression. Secondly, case values in a switch are required to be constant expressions (known at compile time), which further simplifies the optimization job for the compiler. Indeed, compilers do optimize switches better than if-else trees [6].

Project analysis found 36 defects of this type, with 100% TP rate. Unfortunately, none of the defects were located on performance-critical code paths, so fixing them will not improve application performance.

3.2 Replace access to a few arrays with the same index to one array of structs (RedundantCacheAccessChecker)

Defines the proximity of data accessed at the same time to improve data access. The idea is to put data in the same cache line and basic purpose is to find the cases where 2 or more arrays accessed with same indexes.

Specifically, it may be considered that, in the data structure, a data field that is frequently accessed is defined before, and a data field that is seldom accessed is defined after. In this way, most accesses need to be processed only once by loading the cache; otherwise, multiple times of loading are required. The idea is to encapsulate the field assignment in the extracted hotspot structure to ensure that the non-hotspot fields are assigned values at the same time to avoid omission.

The main FP sources are:

- Detect on very small arrays like 2-32 elements which fully fits in cache and have aggressive random access in cycle. In addition to caching, often such an array is used as a way to address variables by indexes instead of by names (for example a common use-case is to store X and Y coordinates as an array of two elements). Other times an array is simply a source of constant data which is likely to be optimized away completely.
- Values read from different arrays is not synchronized by index, and grouping items by structure in one array will have negative effect in performance.

Only 4 cases appear to be true positive, others cases fall into one of FP categories described above, which make it just ~2.5% TP rate.

3.3 Reorder condition sub-expressions to avoid redundant heavy-weight calculations (WeightingConditionChecker)

This checker leverages the short-circuit evaluation principle, implemented in C and C++. Short-circuit evaluation guarantees that the right-hand operand of built-in `&&` (logical AND) and `||` (logical OR) operands will not be evaluated if evaluation of the left-hand operand already determined expression result. This means that a programmer can reorder operands in if conditions so that more expensive to compute operands appear last, which will yield an increase in performance in cases where these operands are never evaluated.

The heuristic by which the approximate cost of an operator is determined is as follows. Each operand has a "cost" value, determined by the most expensive operation performed (fig. 1).

Cost	Operation
1	Literal expression
2	Variable access expression
3	Conditional expression
4	Call expression

Fig. 1. Weighting condition checker operation cost

Operands are sorted by their cost, with lower cost operands being placed first. Care is taken to avoid reordering operands which share variables, to avoid cases like reordering `nullptr` checks and pointer dereferencing. Sadly, no similar algorithm exists for preventing reordering function calls with side effects which affect each other.

Checker could be especially useful in cases where a condition with many operands is inside a loop.

Analysis of Yosys found 138 defects, with 87 of them being true positives (63% TP rate). The main sources of false positives (51 cases) are short functions inlined by the compiler. Function calls have a cost of 4, yet the real operation inside is often much cheaper. Reordering such conditions will have no effect or sometimes an opposite effect.

82 cases contain external functions which made these cases difficult to analyze and optimize. Due to branch predictor and speculative execution such optimizations generally will have no effect.

Other cases do not lie on critical execution paths, and fixing them requires deep knowledge of the project to know with certainty that reordering conditions will produce no undesired side-effects.

3.4 Avoid redundant heap allocations (RedundantHeapAllocChecker)

The rule states that one should avoid redundant heap allocations, i.e. situations where neither manual lifetime management, nor big memory blocks are required. Frequent memory allocation and deallocation can be a serious performance issue. In some cases, a better approach might be to re-use a block of memory allocated once.

Analysis produced 67 defects, with 6 of them being true positives (9% TP rate). The main FP sources are:

- Pointer leaves the scope of the function where memory is allocated, i.e. manual memory management is, in fact, required (33/67 ≈ 49% of all cases);
- There are common recommendations not to allocate more than 16KB per function on the stack and use heap allocation instead. Our tool uses a limit of 4KB. There are cases where allocated size is constant and greater than 4KB, which can be considered a FP (12/67 ≈ 18% of all cases);
- Cases where we cannot assume the allocated memory size is FP due to insufficient knowledge of the code. (16/67 ≈ 24% of all cases).

True positive cases that we decided not to fix:

- Issues in non-performance critical code, mostly used for debugging and error messaging functions;
- Console output of statistics;
- File output dump functions;
- Dead code (functions not used by the application);
- Lookup of libraries and files by path;
- One case where a 20-byte structure was allocated on heap and was freed at the end of the function. But the defect is not on any performance critical path and has no measurable performance impact.

3.5 Avoid double checking the same value (DoubleCheckChecker)

The rule states that if a pointer validity check is performed at some point in the code and enters a “safe code block”, any subsequent checks within the safe code block are redundant. For safety and security reasons, SEI CERT C Coding Standard [7] recommends that any called function validate its parameters.

Influence of the multiple null checks on performance is questionable for two reasons. First, if both checks are visible by the compiler, it will optimize the latter check during global common subexpression elimination (available in GCC under the `-fgcse` flag). Additionally, if the check is not removed for whatever reason, branch prediction will minimize the second branch to a no-op. There are, however, several older methods of branch prediction that can create situations where an incorrect branch is taken [8]. Static prediction is the simplest branch prediction technique because it does not rely on information about the dynamic history of code executing. Instead, it predicts the

outcome of a branch based solely on the branch instruction. With static prediction all decisions are made at compile time, before the execution of the program. The early implementations of SPARC [9] and MIPS [10] always predict that a conditional jump will not be taken, so they always fetch the next sequential instruction. And this is probably even source of recommendation to handle exceptional cases inside if with return. A more advanced form of static prediction presumes that backward branches will be taken and that forward branches will not. A backward branch is one that has a target address that is lower than its own address.

GCC uses `-fdelete-null-pointer-checks` flag (commonly enabled under `-O2`) to enable global dataflow analysis that eliminates useless checks for null pointers. The optimization algorithm assumes that a `nullptr` can never be dereferenced, since dereferencing it would lead to undefined behavior under C++ Standard, and a trap in most real-world cases. This means that if a pointer has already been dereferenced, any later checks for `nullptr` can be discarded.

These optimizations make found issues irrelevant. However, such a rule can still be useful with very old versions of compiler or some specific architectures where the optimization passes like the ones described above are not available.

3.6 Avoid redundant memory zeroing (RedundantZeroMemoryChecker)

The rule states that redundant calls to `memset`, such as after allocating memory with `calloc`, should be avoided. The memory operation functions `memset/memset_s` involve system calls and have a relatively high overhead. If the memory is used to store a string, you can avoid zeroing the entire block, since the '\0' terminator prevents reading the unused tail.

Analysis found 7 defects. We classified 3 cases as FPs. These cases had no efficient fix—a zero-initialized structure, with some fields being individually initialized afterwards, was not possible to optimize.

Other 4 cases had a similar structure: a call to `calloc` followed by a call to `memset`.

Source	GCC 6.1	GCC 4.1.2
`struct SomeStruct {`	`Create():`	`Create():`
`  float a;`	`  mov  esi, 64`	`  sub     %rsp, 8`
`  double b;`	`  mov  edi, 1`	`  mov     %edi, 1`
`  char* c;`	`  jmp  calloc`	`  mov     %esi, 64`
`  int d[10];`		`  call    calloc`
`};`		`  cld`
		`  mov     %rdx, %rax`
		`  mov     %ecx, 8`
`SomeStruct* Create() {`		`  xor     %eax, %eax`
`  SomeStruct* s;`		`  mov     %rdi, %rdx`
`  s =`		`  rep     stosq`
`(SomeStruct*)calloc(1,`		`  mov     %rax, %rdx`
		`  add     %rsp, 8`
`sizeof(SomeStruct));`		`  ret`
`  memset(s, 0,`		
`sizeof(SomeStruct));`		
`  return s;`		
`}`		

Fig. 2. Optimization of memory zeroing under different versions of GCC

Additionally, this code is well optimized by a modern compiler, and only an old version of GCC 4.1.2 do not (fig. 2).

3.7 Avoid passing function arguments by value (RedundantArgCopyChecker)

This class of issues stems from a syntax feature of the C and C++ languages: when passing an argument to a function, the default operation is to take a copy of the passed value. This can cause a performance issue when big structures are copied by accident.

There are two alternatives to pass-by-copy. In C, one can modify the function signature to take a pointer (or const pointer if the value is not meant to be modified), and take an address of a value when passing it to the function. In C++, a more streamlined option is to pass a reference or const reference, which does not require changing the caller code and also disallows null values.

Of course, both of these methods involve pointer indirection, which can introduce a pessimization into the callee code. Most codestyle rules recommend passing values by copy only when they are small enough to be copied in registers. Our research found that 16 bytes is the maximum structure size which can be safely like this (fig. 3).

12 bytes structure

```
foo():
  sub    rsp, 24
  lea    rdi, [rsp+4]
  call   T::T()
  mov    rdi, QWORD PTR [rsp+4]
  mov    esi, DWORD PTR [rsp+12]
  call   bar(T)
  add    rsp, 24
  ret
```

16 bytes structure

```
foo():
  sub    rsp, 24
  mov    rdi, rsp
  call   T::T()
  mov    rdi, QWORD PTR [rsp]
  mov    rsi, QWORD PTR [rsp+8]
  call   bar(T)
  add    rsp, 24
  ret
```

20 bytes structure

```
foo():
  sub    rsp, 40
  mov    rdi, rsp
  call   T::T()
  sub    rsp, 32
  movdqa xmm0, XMMWORD PTR [rsp+32]
  mov    eax, DWORD PTR [rsp+48]
  movups XMMWORD PTR [rsp], xmm0
  mov    DWORD PTR [rsp+16], eax
  call   bar(T)
  add    rsp, 72
  ret
```

Fig. 3. Assembly generated from function foo, which creates structure of specified size and passes that structure to function bar by value

Our analysis found only a single issue, where a structure of 32 bytes was copied when passed into a print function. Since the cost of printing greatly outweighs the cost of pushing values on the stack, we consider this case not performance critical.

3.8 Avoid using RTTI (RttiChecker)

RTTI (Run-Time Type Information) is a special mechanism in the C++ language which allows the user to retrieve type information at runtime (mainly the type name), as well as traverse inheritance trees [11, sect. 17.8]. This is the mechanism behind features like `typeid` and `dynamic_cast`. This language feature, along with exceptions, has long been a polarizing discussion point [12, sect. C.146], and a notorious breaker of the "zero-overhead abstraction" rule [13], since for polymorphic classes type information now needs to be stored alongside the value, regardless of whether `dynamic_cast` is actually used or not. For this reason, RTTI is disabled in many high-performance projects [14], and the rule to disable RTTI via compiler flags is present in many C++ style guides [15, 16].

To explain the options a programmer has when avoiding RTTI, let us conduct a one-paragraph review of different kinds of polymorphism. Dynamic polymorphism, the very same that is being used by virtual inheritance in C++, is the practice of using dynamic method dispatch when calling methods of polymorphic objects. Dynamic method dispatch is called dynamic since the work of selecting an appropriate derived method happens at runtime (using what in essence is just pointers to functions) [17]. This approach does not restrict the amount of derived classes that can be created and allows derived classes to be declared in separate translation units. This means that the C++ virtual polymorphism model is openly extensible without modifying the base class, and so it is an example of open type set polymorphism (referred to later as simply open polymorphism). A polar opposite to open polymorphism is closed polymorphism, where no extension of the class hierarchy is allowed. An example of closed polymorphism is the variant data type; a variable of such type is a wrapper around one of N types specified during declaration [18, p.24]. Of course, to call variant types polymorphic we also need to implement method dispatch, and here, since we know all types in advance, we can avoid using runtime pointer indirection and just create a branch structure, which checks which type is stored inside and call its appropriate method. This logic is generated statically, and so variant-based polymorphism is an example of static polymorphism.

A general recommendation of the C++ community when replacing RTTI with other mechanisms is to implement static, closed polymorphism, as opposed to dynamic, open polymorphism that virtual methods and inheritance represent [19, p. 18][18, p. 80].

3.9 Other rules

Analysis was also performed on several other rules. Most of these deserve only a passing mention, since all detects generated by them were optimized by the compiler.

These additional rules were:

- Avoid repeated variable initialization (RedundantInitializationChecker);
- Avoid complex calculations inside the loop condition (RedundantLoopCondCalcChecker);
- Avoid repeated nested dereferencing, such as nested member variable access via a chain of pointers (RedundantAddressCalculationChecker);
- Do not mark global variables as volatile (RedundantVolatileGlobalVarChecker). Detects from this rule were not optimized, but it was impossible to determine accurately whether they were TP or FP without deep knowledge of the code.

4. Related Works

4.1 Proprietary tools

PVS-Studio, trial version is used [20].

PVS-Studio has 35 rules on performance optimization in C++. 12 of them intersected with the rules detected by Cooddy.

After running this tool on Yosys we found 128 warnings, detected by 10 rules. 81 of these warnings are true positive (FP rate of 37%). The only warning that was common with Cooddy was the one from RedundantArgCopyChecker.

4.2 Open source projects

CppCheck has 11 performance checkers that apply to CWE398, CWE597, CWE628, CWE704. Most of them are related to std::string. No rules are common with Cooddy checkers [21].

4.3 Non-commercial article-based tools

CARAMEL is a novel static technique that detects and fixes performance bugs that have non-intrusive fixes likely to be adopted by developers. Each performance bug detected by CARAMEL is associated with a loop and a condition. When the condition becomes "true" during the loop execution, all the remaining computation performed by the loop is wasted. CARAMEL analyses C/C++/Java applications [22].

Other tools are Toddler [23], Clarity [24], LDoctor [25].

5. Conclusion

We can split the code guidelines we reviewed into 4 main categories: valid and useful rules, outdated rules, rules that can be implemented in SCSA but were not possible to be properly evaluated using our method and software, and finally rules that cannot be implemented in SCSA at all.

5.1 Useful rules

Out of the rules reviewed RedundantArgCopyChecker is the only one we consider useful when working on a modern architecture and with a modern compiler. Although we did not find many issues on the “Yosys” project, it’s likely because on any project with significant attention such issues are quickly fixed when on performance-critical paths. SCSA can be used during development to more quickly spot such issues.

5.2 Outdated rules

Rules in this category are outdated in the sense that they are no longer a programmer’s burden—today’s compiler technology is advanced enough that all of the possible performance issues are optimized away. The checkers in this category are:

- RedundantMultipleIfElseChecker;
- DoubleCheckChecker;
- RedundantInitializationChecker;
- RedundantLoopCondCalcChecker;
- RedundantAddressCalculationChecker;
- RedundantZeroMemoryChecker.

5.3 Poor method

These issues are useful and it is possible to gain performance by fixing them, but the current implementation in our SCSA tool is not advanced enough to evaluate their impact properly. These issue categories are:

- RedundantCacheAccessChecker, due to a high number of false positives;
- WeightingConditionChecker, because in its current implementation it does not consider inlined functions and functions with side-effects when reordering conditional operands;
- RedundantHeapAllocChecker, due to a high number of false positives, mainly when pointers to allocated memory leave the function scope.

5.4 Not implementable in automatic SCSA

We consider two checkers unimplementable in SCSA engines in general: RedundantVolatileGlobalVarChecker and RttiChecker.

RedundantVolatileGlobalVarChecker is essentially a “code knowledge” rule. The reason for its existence is due to a very broad misuse of the volatile keyword as an erroneous way to create atomic variables [26; 27, sect. 5.1.2.3 para.2; 12, section CP.8]. The volatile keyword should only be used in very specific circumstances, and in these circumstances its necessity is a fact hidden from the compiler and known only by the programmer.

RttiChecker is not possible to implement properly because even in cases when no code uses any functionality dependent on RTTI, a compiler does not know whether the source files are compiled into an object file are linked to an object file with RTTI enabled. Some compilers do not support linking object files in such a way [28].

5.5 Final observations

The analysis performed shows that more often than not, compiler technology covers the common issues with the source code and allows the programmer to write code for readability and simplicity as a first priority. Additionally, automatic SCSA could be of use when configured to exclude classes of issues that are no longer relevant in a modern environment. Cooddy in particular was often not able to properly filter out false positives, but the approach in general is viable and further research is ought to improve Cooddy’s ability to cover issues still affecting today’s code. On the other hand there are some specific architectures which has no branch prediction module or sophisticated memory management units. In this case checking performance coding rules has a sense and SCSA can be helpful.

References

[1] GCC, the GNU Compiler Collection. Available at: https://web.archive.org/web/20220913154847/http://gcc.gnu.org/, accessed 2022-09-13.
[2] The State of Developer Ecosystem 2021: C++. Available at: https://web.archive.org/web/20220609172327/https://www.jetbrains.com/lp/devecosystem-2021/cpp/, accessed 2022-06-09.
[3] GCC: Options That Control Optimization. Available at: https://web.archive.org/web/20220910030424/https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, accessed 2022-09-10.
[4] Yosys Open SYnthesis Suite. Available at: https://github.com/YosysHQ/yosys, accessed 2022-09-10.
[5] Compiler Explorer. Available at: https://godbolt.org/
[6] V. Lazarenko. From Switch Statement Down to Machine Code. Available at: https://web.archive.org/web/20220313040503/http://lazarenko.me/switch/, accessed 2022-03-13.

[7] SEI CERT C Coding standard. API00-C. Functions should validate their parameters. Available at: https://wiki.sei.cmu.edu/confluence/display/c/API00-C.+Functions+should+validate+their+parameters, accessed 2022-03-13.
[8] J.E. Smith. A study of branch prediction strategies. In Proc. of the 8th Annual Symposium on Computer Architecture, 1981, pp. 135-148.
[9] SPARC International. Available at: https://sparc.org/, accessed 2022-03-13.
[10] D. Patterson. Computer Organization and Design, Fifth Edition. Morgan Kaufmann, 2013, 800 p.
[11] ISO/IEC N4860 – Programming Language C++. [Working draft]. Available at: https://web.archive.org/web/20220901051849/https://isocpp.org/files/papers/N4860.pdf, accessed 2022-09-01.
[12] C++ Core Guidelines. Available at: https://web.archive.org/web/20220913102723/https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines, accessed 2022-09-13.
[13] B. Stroustrup. Foundations of C++. Lecture Notes in Computer Science, vol. 7211, 2013, pp. 1-25.
[14] Electronic Arts Standard Template Library. Available at: https://github.com/electronicarts/EASTL/blob/master/doc/FAQ.md#info11-what-c-language-features-does-eastl-use-eg-virtual-functions, accessed 2022-09-01.
[15] Google C++ Style Guide. Available at: https://google.github.io/styleguide/cppguide.html#Run-Time_Type_Information__RTTI_, accessed 2022-09-01.
[16] LLVM Coding Standards. Available at: https://llvm.org/docs/CodingStandards.html#do-not-use-rtti-or-exceptions, accessed 2022-09-01.
[17] S. Milton, H.W. Schmidt. Dynamic Dispatch in Object-Oriented Languages. Australian National University, TR-CS-94-02, 1994.
[18] J.R. Bandela. Polymorphism != Virtual. Flexible Runtime Polymorphysm wihtout Inheritance. In the CppCon 2019 Presentation Materials, 2019, available at: https://github.com/CppCon/CppCon2019/blob/master/Presentations/polymorphism__virtual/polymorphism__virtual__john_bandela__cppcon_2019.pdf, accessed 2022-09-01.
[19] L. Dionne. Runtime polymorphism: back to the basics. In the CppCon 2017 Presentation Materials, 2017, available at: https://github.com/CppCon/CppCon2017/blob/master/Presentations/Runtime%20Polymorphism%20-%20Back%20to%20the%20Basics/Runtime%20Polymorphism%20-%20Back%20to%20the%20Basics%20-%20Louis%20Dionne%20-%20CppCon%202017.pdf, accessed 2022-09-01.
[20] PVS-Studio. Available at: https://pvs-studio.com/en/docs/warnings/#MicroOptimizationsCPP, accessed 2022-09-01.
[21] CPPCheck. Available at: https://github.com/danmar/cppcheck, accessed 2022-09-01.
[22] A. Nistor, P-C. Chang et al. Caramel: detecting and fixing performance problems that have non-intrusive fixes. In Proc. of the 37th International Conference on Software Engineering, vol. 1, 2015, pp. 902-912
[23] A. Nistor, L. Song et al. Toddler: Detecting Performance Problems via Similar Memory-Access Patters. In Proc. of the 35th International Conference on Software Engineering (ICSE), 2013, pp. 562-571.
[24] O. Olivio, I. Dilling, C. Lin. Static detection of asymptotic performance bugs in collection traversals. In Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2016, pp. 369-378.
[25] L. Song, S. Lu. Performance Diagnostics for Inefficient Loops. In Proc. of the IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET), 2017, pp. 370-380.
[26] SEI CERT C Coding Standard. 3 Recommendations. Rec. 14. Concurrency (CON). CON02-C. Do not use volatile as a synchronization primitive. Available at: https://web.archive.org/web/20220916130555/https://wiki.sei.cmu.edu/confluence/display/c/CON02-C.+Do+not+use+volatile+as+a+synchronization+primitive, accessed 2022-09-16.
[27] ISO International Standard ISO/IEC 9899:201x – Programming Language C. [Working draft]. Geneva, Switzerland: International Organization for Standardization (ISO). Available at: https://web.archive.org/web/20220831233111/https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf, accessed 2022-08-31.

[28] XL C/C++ for Linux: -qrtti, -fno-rtti. Available at: https://web.archive.org/web/20220916134348/https://www.ibm.com/docs/en/xl-c-and-cpp-linux/13.1.1?topic=descriptions-qrtti-fno-rtti-qnortti-only, accessed 2022-09-16.

Информация об авторах / Information about authors

Александр Юрьевич ГЕРАСИМОВ – кандидат физико-математических наук, руководитель группы анализа программ в Московском исследовательском центре Российского исследовательского института компании Huawei с 2020. Сфера научных интересов: автоматический анализ программ, обеспечение качества программ, компиляторные технологии, цикл разработки безопасного ПО.

Alexander Yurievich GERASIMOV – Doctor of Philosophy in Computer Sciences, Head of the Program Analysis team of Moscow Research Center of Russian Research Institute of Huawei Company from 2020. Research interests: automatic program analysis, quality assurance, compiler construction technologies, secure software development cycle.

Алексей Алексеевич КАНАХИН – кандидат физико-математических наук, ведущий инженер-исследователь в Московском исследовательском центре Российского исследовательского института компании Huawei с 2019 г. Сфера научных интересов: высокопроизводительные вычисления, архитектура компьютера, отказоустойчивость компьютерных систем, автоматический анализ программ.

Alexey Alexeyevich KANAKHIN – Doctor of Philosophy in Physics of Semiconductors, Senior Research Engineer in Moscow Research Center of Russian Research Institute of Huawei Company from 2019. Research interests: high-performance computing, computer architecture, fault-tolerant computer systems, automatic program analysis.

Петр Алексеевич ПРИВАЛОВ – магистр прикладной математики и физики, ведущий инженер группы анализа программ в Московском исследовательском центре Российского исследовательского института компании Huawei с 2020. Сфера научных интересов: автоматический анализ программ, обеспечение качества программ, компиляторные технологии, цикл разработки безопасного ПО, архитектура программ, многопоточные алгоритмы.

Petr Alekseevich PRIVALOV – Master of Science in applied mathematics and physics, Senior Software Engineer at the Program Analysis team of Moscow Research Center of Russian Research Institute of Huawei Company from 2020. Research interests: automatic program analysis, quality assurance, compiler construction technologies, secure software development cycle, program architecture and design, multithread algorithms.

Андрей Александрович ЖУКОВ – магистр по специальности "информатика и вычислительная техника", сотрудник группы анализа программ в Московском исследовательском центре Российского исследовательского института компании Huawei с 2022. Сфера научных интересов: автоматический анализ программ, компиляторные технологии, метапрограммирование, архитектура программного обеспечения.

Andrey Alexandrovich ZHUKOV – Master of Science in information and computation systems, member of the Program Analysis team of Moscow Research Center of Russian Research Institute of Huawei Company from 2022. Research interests: automatic program analysis, compiler technology, metaprogramming, software architecture.

Евгений Аркадьевич КАМИНСКИЙ – старший разработчик в команде анализа программ в Московском исследовательском центре Российского исследовательского института компании Huawei с 2022. Сфера научных интересов: автоматический анализ программ, обеспечение качества программ, компиляторные технологии, цикл разработки безопасного ПО.

Evgenii Arkadievich KAMINSKII – senior developer of the Program Analysis team of Moscow Research Center of Russian Research Institute of Huawei Company from 2022. Research interests: automatic program analysis, quality assurance, compiler construction technologies, secure software development cycle.