SoftwareX 7 (2018) 95–100

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

libVersioningCompiler: An easy-to-use library for dynamic generation and invocation of multiple code versions

S. Cherubin [*], G. Agosta

Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Via G. Ponzio 34/5, I-20133 Milano, Italy

ARTICLE INFO

Article history:
Received 23 May 2017
Received in revised form 13 March 2018
Accepted 23 March 2018

Keywords:
Dynamic compilation
Versioning compiler
Continuous optimization

ABSTRACT

We present libVersioningCompiler, a C++ library designed to support the dynamic generation of multiple versions of the same compute kernel in a HPC scenario. It can be used to provide continuous optimization, code specialization based on the input data or on workload changes, or otherwise to dynamically adjust the application, without the burden of a full dynamic compiler. The library supports multiple underlying compilers but specifically targets the llvm framework.

We also provide examples of use, showing the overhead of the library, and providing guidelines for its efficient use.

Code metadata

Current code version	v1.2
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-17-00040
Legal Code License	LGPL v3
Code versioning system used	git
Software code languages, tools, and services used	C++, cmake, LLVM, Clang.
Compilation requirements, operating environments & dependencies	Suggested system: Ubuntu 16.04 x86_64 or version greater. Required dependencies: dl, uuid-dev. Optional dependencies: llvm-4.0-dev, libclang-4.0-dev.
If available Link to developer documentation/manual	See README.md in the repository
Support email for questions	stefano.cherubin@polimi.it

1. Motivation and significance

Designing and implementing High Performance Computing (HPC) applications is a difficult and complex task that requires to master several specialized languages and performance-tuning tools; however, this prerequisite is incompatible with the current trend that opens HPC infrastructures to a wider range of users [1,2]. The current model that sees the HPC center staff directly supporting the development of applications will become unsustainable in the long term. Thus, the availability of effective APIs and programming languages is crucial to provide migration paths towards novel heterogeneous HPC platforms as well as to guarantee the developers' ability to work effectively on these platforms.

While in general purpose scenarios profile-guided compile-time code transformation can provide sufficient optimization, in HPC scenarios where large data sets are employed, profiling may be infeasible. In these cases, which are becoming more and more common [3], dynamic approaches can prove more effective. The practice of improving the application code at runtime through dynamic recompilation is known as *continuous program optimization* [4–6]. Although it has been studied for more than a decade, very few people adopt it in practice since it is difficult to perform manually, and, when performed automatically, it can compromise software maintainability. At the same time, autotuning is used both to tune software parameters and to search the space of compiler optimizations for optimal solutions [7]. Autotuning frameworks can select one of a set of different versions of the same computational kernel to best fit the HPC system runtime conditions, such as system resource partitioning, as long as such versions are generated at compile time. Few of these frameworks are actually able to perform continuous optimization, and those that support it

[*] Corresponding author.
E-mail address: stefano.cherubin@polimi.it (S. Cherubin).

https://doi.org/10.1016/j.softx.2018.03.006
2352-7110/© 2018 Published by Elsevier B.V.

do so only through specific versions of a dynamic compiler [8,9] or through cloud-based platforms [10].

libVersioningCompiler (abbreviated libVC) can be used to perform continuous program optimization using simple C++ APIs. libVC allows different versions of the executable code of a computational kernel to be transparently generated on the fly. Continuous program optimization with libVC can be performed by dynamically enabling or disabling code transformations, and changing compile-time parameters according to the decisions of other software tools such as a generic application autotuner.

The rest of the paper is organized as follows. In Section 2 we describe the software architecture, the internal APIs and their functionalities. In Section 3 we introduce an example of intended use and discuss benefits and overhead deriving from the implementation of continuous program optimization through libVC in a generic scenario. In Section 4 we highlight the impact of libVC in both industry and research field. Finally, we draw some conclusions in Section 5.

2. Software description

The goal of libVC is to allow a C/C++ compute kernel to be dynamically compiled multiple times while the program is running, so that different specialized versions of the code can be generated and invoked. This capability is especially useful when the optimal parametrization of the compiler depends on the program workload. In these cases, the ability to switch at runtime between different versions of the same code can provide significant benefits, as shown in [11,12].

Indeed, in general purpose code it is preferable to profile the application to statically generate ahead of time the most efficient versions. However, in HPC code the execution times are usually so long that a profiling run may not be an attractive choice. On the contrary, libVC enables the exploration and tuning of the parameter space of the compiler at runtime, while the program is performing useful work.

libVC considers as valid compute kernels any C-like procedure or function that can be compiled to object code. There is just one constraint that should be enforced on the compute kernel: it must respect C linkage rules. This means that no name mangling should be applied to the compute kernel itself. Within our model, the `Compiler` is the tool used to compile the compute kernel, and the `Version` is the configuration passed to the compilation task. We assume to work with deterministic `Compilers`. In this scenario, a `Version` produces at most one executable code. No executable code is generated when the specified configuration is invalid.

2.1. Software architecture

The libVC source code is available under the LGPLv3 licence. It is compliant with the C++11 standard and it comes with configuration files to ease the setup by using the `CMake` build system. The minimum required `CMake` version is 3.0.2. The build system automatically checks the presence of the optional dependencies `LLVM` and `libClang`, whose version must be greater than `4.0.0`. Whenever these dependencies are not satisfied, some features are automatically disabled during the library installation. Please see Code metadata Table for a detailed and exhaustive list of dependencies.

Description of the software model

Fig. 1 shows a simplified UML class diagram of this software. It is possible to identify three main classes in the source code. The simplest class, which is called `Option`, represents each of the flag and parameters that are passed to LibVC in order to compile a version of a computing kernel. The `Compiler` abstract class defines the interface that allows the host application to interact with `Compiler` implementations. libVC provides up to three possible implementations for the `Compiler` abstract class: `SystemCompiler`, which relies on system calls to external compilers that are already installed in the host system; `SystemCompilerOptimizer`, which is an extension of a `SystemCompiler` that also supports external optimization tools (such as the LLVM optimizer `opt`); and `ClangLibCompiler`, which exploits the compiler-as-a-library paradigm through the `Clang` APIs.[1] Please note that `ClangLibCompiler` is installed only if `LLVM` and `libClang` dependencies are satisfied. The last important class is the `Version` class, which represents a compute kernel defined in a specific source file, with a given compiler configuration. A `Version` object is compiled with the chosen `Compiler` using an ordered list of `Options`. It contains a unique identifier, references to `Compiler` and `Options` used to compile it, and references to the files that are generated by the `Compiler` while compiling the `Version`. The configuration of a `Version` object is immutable throughout the lifetime of that object. The `Version` class also provides APIs to control the stages of the compilation process: it is possible to create a `Version` object and postpone the execution of the selected `Compiler` to a later stage.

2.2. Software functionalities

libVC provides an easy-to-use interface that can be employed to perform the dynamic compilation of a kernel, and to load compiled `Versions` as C-like function pointers. libVC itself does not provide any automatic selection of which `Version` should be executed. The decision of which `Version` is the most suitable for a given task is left to policies defined by the programmer or other autotuning frameworks such as mARGOt [13] or cTuning [14].

libVC comes in two different flavours: with detailed low-level APIs and with simple high-level APIs. The latter is optimized for the most common use cases, they exploit the default system compiler and do not support any external optimization tool, whereas low-level APIs allow a more fine grained setup and support split-compilation techniques [15]; hence, the resulting source code is slightly more verbose.

The typical usage of libVC involves different stages. The first task must be the declaration and initialization of the `Version`-independent tools, such as `Compilers` and `Version` builders, which are helper objects designed to properly setup a `Version` configuration. Low-level APIs allow the programmer to customize one or more `Compiler` implementations. High-level APIs expose a special function to transparently perform this task; it is required to be invoked just once in the whole process lifetime. After that, it is possible to proceed to the `Version` configuration. The programmer can, by using low-level APIs, dynamically forge and arrange `Options`, set the chosen `Compilers`, manipulate file and kernel names to identify the code to be compiled. The `Version` builder is the component which allows this low-level setup. Once the `Version` builder has its fields filled up, it can be finalized to generate a `Version` object. High-level APIs receive all these parameters and produce a `Version` object in a single function call. High-level APIs limit the configuration to one `Version` at a time while low-level APIs allows parallel configuration of multiple `Versions`. Once a `Version` object is finalized, it has to be compiled. The compilation task is activated by the programmer through a dedicated API. It may trigger more than one sub-task when it involves split-compilation techniques. In the absence of compilation errors, and regardless of which APIs are being used, at the end of this stage libVC generates a binary shared object. From this same shared

1 http://clang.llvm.org/docs/Tooling.html.

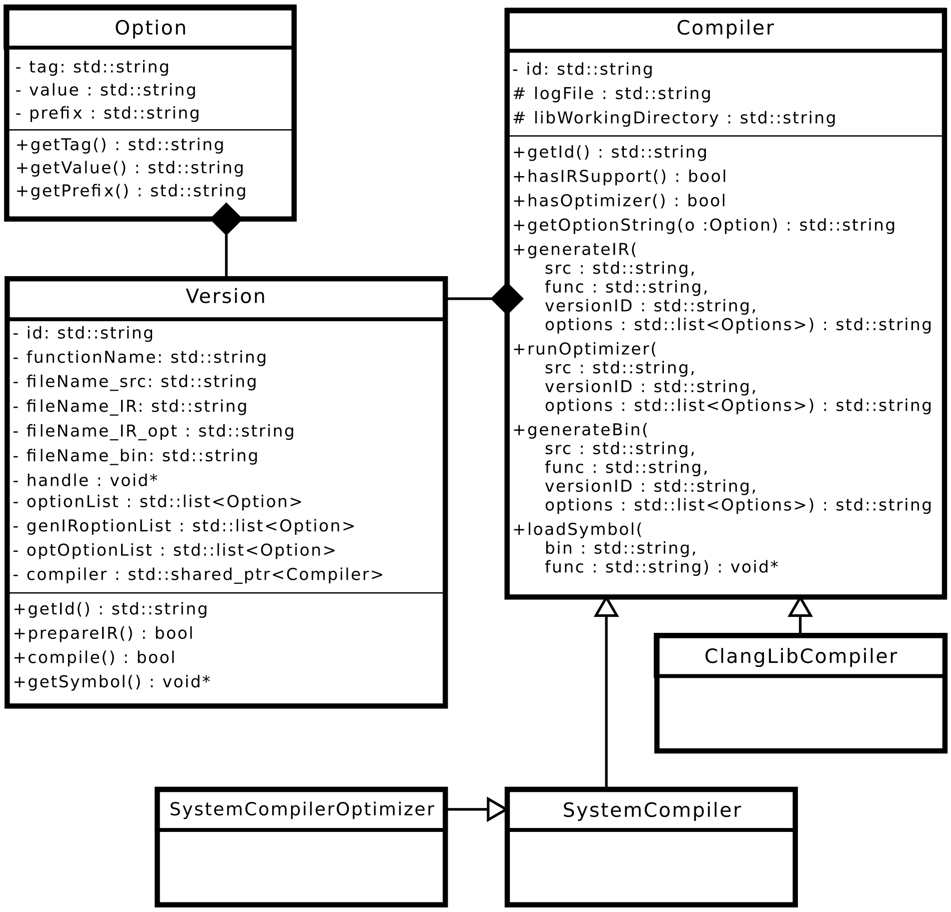
Option
- tag: std::string
- value : std::string
- prefix : std::string
+getTag() : std::string
+getValue() : std::string
+getPrefix() : std::string
Compiler
- id: std::string
# logFile : std::string
# libWorkingDirectory : std::string
+getId() : std::string
+hasIRSupport() : bool
+hasOptimizer() : bool
+getOptionString(o :Option) : std::string
+generateIR(src : std::string, func : std::string, versionID : std::string, options : std::list<Options>) : std::string
+runOptimizer(src : std::string, versionID : std::string, options : std::list<Options>) : std::string
+generateBin(src : std::string, func : std::string, versionID : std::string, options : std::list<Options>) : std::string
+loadSymbol(bin : std::string, func : std::string) : void*
Version
- id: std::string
- functionName: std::string
- fileName_src: std::string
- fileName_IR: std::string
- fileName_IR_opt : std::string
- fileName_bin: std::string
- handle : void*
- optionList : std::list<Option>
- genIRoptionList : std::list<Option>
- optOptionList : std::list<Option>
- compiler : std::shared_ptr<Compiler>
+getId() : std::string
+prepareIR() : bool
+compile() : bool
+getSymbol() : void*
ClangLibCompiler
SystemCompilerOptimizer
SystemCompiler

Fig. 1. Simplified UML class diagram of libVC.

object libVC loads a function pointer symbol, which points to the kernel.

The target kernel may include other files or refer to external symbols. libVC will act just as a compiler invocation and will try to resolve external symbols according to the given compiler and linker options.

libVC defers the resolution of the compilation parameters to run-time. The only piece of information that is needed at design-time is the prototype of the kernel, which has to be used for a proper function pointer cast.

libVC also provides hooks to enable tracking and versioning of the compiled versions.

3. Illustrative examples

libVC can be exploited to apply a wide range of optimization through the dynamic compilation. The official repository[2] provides some examples of usage in the test files. In this section we show and discuss a generic use case of continuous program optimization performed through libVC. Listing 1 illustrates the dynamic adaptation of a counting sort algorithm to the data workload. In particular, the counting sort implementation is specialized through recompilation using libVC every time the `min` and `max` value of range of the data to be sorted change. When the `min` and `max` values of the range of the data are known at compile-time it is possible to perform array allocation and loop optimizations more efficiently.

Listing 1 reveals the several stages of the compilation flow of libVC. In the `main` function, an initialization is needed before using libVC. This is done in line 40 using a simple API invocation. From line 8 to line 20 we see how to configure a new `Version` for dynamic compilation. The following lines (22–27) perform the actual dynamic compilation. It is possible to notice in line 69 the call to the dynamically compiled kernel, which is very similar to the call to a statically linked kernel (line 53).

As proof of concept, we tested the benefits of continuous program optimization implemented with libVC by comparing the time-to-solution of the statically linked kernel against a dynamically compiled version of the same kernel, as shown in listing 1. We compiled both the statically linked and the dynamically compiled kernels using the same compiler and the same optimization level. A full project using code from listing 1 is available on github.[3] We run this example to sort an array of 1 billion 32-bits integers. The platform used to execute the experiment is a supercomputer NUMA node that features two Intel Xeon E5-2630 V3 CPUs (@2.4 GHz) with 128 GB of DDR4 memory (@1866 MHz) on a dual channel memory configuration.

Table 1 shows that dynamically compiled kernels always performs better with respect to the reference statically linked implementation. We define as range size the difference between `max` and `min` values of the range of the data to be sorted. We observe an important speedup when the range size is smaller than 8192 possible values. In those cases the main part of the speedup comes from a more efficient memory allocation of the array in the dynamically compiled kernels. We also notice that the overhead

2 https://github.com/skeru/libVersioningCompiler.

3 https://github.com/skeru/countingsort_libVC.

Listing 1: Benchmark of a statically linked kernel performing counting sort against a dynamically compiled version of the same kernel using LIBVC high-level APIs

```
// libVersioningCompiler High-Level API header file
#include "versioningCompiler/Utils.hpp"

// define kernel signature
typedef void (*kernel_t)(std::vector<int32_t> &array);

vc::version_ptr_t getDynamicVersion(int32_t min, int32_t max) {
  // version configuration using libVC - start
  const std::string kernel_dir = PATH_TO_KERNEL;
  const std::string kernel_file = kernel_dir + "kernel.cpp";
  const std::string functionName = "vc_sort";
  const vc::opt_list_t opt_list = {
    vc::make_option("-O3"),
    vc::make_option("-std=c++11"),
    vc::make_option("-I"+kernel_dir),
    vc::make_option("-D_MIN_VALUE_RANGE="+std::to_string(min)),
    vc::make_option("-D_MAX_VALUE_RANGE="+std::to_string(max)),
  };
  vc::version_ptr_t version = vc::createVersion(kernel_file, functionName, opt_list);
  // version configuration using libVC - end

  // version compilation - start
  kernel_t f = (kernel_t) vc::compileAndGetSymbol(version);
  if (f) {
    return version;
  }
  // version compilation - end
  return nullptr;
}

int main(int argc, char const *argv[]) {
  const std::vector<std::pair<int, int> > data_range = {
    std::make_pair<int,int>(0,256),
    std::make_pair<int,int>(0,512),
    std::make_pair<int,int>(0,1024),
  };
  const size_t data_size = 1000000000;

  // initialize libVersioningCompiler
  vc::vc_utils_init();

  for (const auto range : data_range) {
    TimeMonitor time_monitor_ref;
    TimeMonitor time_monitor_dyn;
    TimeMonitor time_monitor_ovh;

    // running reference version - statically linked
    for (size_t i = 0; i < iterations; i++) {
      // produce workload to process
      auto wl = WorkloadProducer<int32_t>::get_WL_with_bounds(range.first, range.second);
      const auto meta = wl.getMetadata();
      time_monitor_ref.start();
      sort(wl.data, meta.minVal, meta.maxVal); // call reference
      time_monitor_ref.stop();
    }

    // measuring overhead of preparing a new version - start
    time_monitor_ovh.start();
    auto v = getDynamicVersion(range.first, range.second);
    kernel_t my_sort = (kernel_t) v->getSymbol();
    time_monitor_ovh.stop();
    // measuring overhead of preparing a new version - end

    // running dynamic version - dynamically compiled
    for (size_t i = 0; i < iterations; i++) {
      // produce workload to process
      auto wl = WorkloadProducer<int32_t>::get_WL_with_bounds(range.first, range.second);
      time_monitor_dyn.start();
      my_sort(wl.data); // just a call to a function pointer
      time_monitor_dyn.stop();
    }

    // consider average time-to-solution
    std::cout << range.second << "␣" << time_monitor_ref.getAvg() << "␣" << time_monitor_dyn.getAvg() << "␣" <<
        time_monitor_ovh.getAvg() << std::endl;
  }
  return 0;
}
```

Table 1
Experimental results of Time-To-Solution (TTS) averaged over 100 executions on a Ubuntu x86_64 system. Kernels were compiled using `gcc` 5.4.0 with optimization level `-O3`.

Range size [elements]	TTS reference [ms]	TTS LIBVC [ms]	Speedup [%]	Overhead [ms]	Payback [iterations]
256	2831.33	2368.12	19.56	1355.99	3
512	2822.84	2352.27	20.00	1345.25	3
1024	2820.67	2347.28	20.17	1356.86	3
2048	2831.92	2351.99	20.41	1361.37	3
4096	2914.13	2440.47	19.41	1353.05	3
8192	3967.59	3966.21	0.03	1354.12	982
16384	5168.64	5163.51	0.10	1370.82	268
32768	6459.75	6430.77	0.45	1358.26	47

of dynamically compiling a new `Version` is not related with the range size. This overhead can be absorbed within 3 iterations when the range size is small, and within less than one thousand iterations in the worst case.

It is also possible to use LIBVC to dynamically compile and run several functions or the same function with different options. A more complex example of usage of LIBVC which exploits these features can be found on github[4] where we dynamically compile and run the full PolyBench/C [16] benchmark suite within the same C++ program.

4. Impact

LIBVC is a software tool that supports the generation and execution of multiple versions of C++ kernels. This means that LIBVC allows a wider range of users to adopt continuous optimization practices by generating workload-dependent specializations of one or more kernels. Accordingly, LIBVC enables the development of autotuning techniques, as well as the comparison of different autotuning algorithms within a neutral platform with any desired compiler. By providing the option to select multiple compilers, LIBVC can be easily adopted by industrial users, such as supercomputing centers, as they are often constrained to vendor-specific compilers.

LIBVC is used within the European project ANTAREX [17,18], which aims at expressing the capability of applications to self-adapt to runtime conditions (we call this practice *autotuning*) through a Domain Specific Language (DSL) and at providing runtime management and autotuning support for applications that target green and heterogeneous HPC systems up to Exascale. The application functionality is expressed through C/C++ code (possibly including legacy code), whereas the non-functional aspects of the application, including parallelization, mapping, and adaptivity strategies are expressed through the DSL developed in the project. The application autotuning is delayed to the runtime phase, where the *software knobs* (application parameters, code transformations and code variants) are configured according to the runtime information that is retrieved from the execution environment. LIBVC serves to dynamically provide code transformations and code variants in the ANTAREX tool flow. The ANTAREX consortium includes two major European supercomputing centers, as well as industrial users in the automotive and bioinformatics application domains.

Case study: Geometrical docking miniapp

To assess the impact of the proposed tools on a real-world application we employ a miniapp developed within the ANTAREX project [17] to emulate the workload of the geometric approach to molecular docking. This class of application is useful in the in-silico drug-discovery process, which is an emerging application of HPC, and consists in finding the best fitting ligand molecule with a pocket in the target molecule [19]. This process is performed by approximating the chemical interactions with the proximity between atoms.

We processed a database of 113161 ligand molecule–pocket pairs on the same test platform we describe in Section 3. The evaluation of every ligand molecule–pocket pair is independent with respect to the other pairs. Therefore, we implemented an MPI-based version of the same miniapp. The input dataset is partitioned among the slave processes.

The initial code base was not developed by the authors, it was developed by another team at Politecnico di Milano. We integrated the code which is executed by each slave process with LIBVC, as for the serial version. It took one hour of work to integrate the miniapp source code with the LIBVC. The integration required to add or modify a total of 60 lines of code over an original code size of 1300 lines of code, which is less than 5% of the code size.

The baseline miniapp took 4354.95 s before the integration. After the integration the miniapp took 1783.93 s – including the overhead for dynamic compilation – for a speedup of 2.44× with respect to the baseline. The speedup is achieved by exploiting code specialization on geometrical functions.

Although the overhead of performing dynamic compilation on every parallel process slows down the running time, the speedup we obtained in the serial version of the miniapp is confirmed also in the parallel case. We run the MPI-based miniapp using 4, 8, 16, and 32 parallel processes. We obtained a speedup of 2.39×, 2.24×, 1.99×, and 1.63× respectively.

Case study: OpenModelica compiler

To assess the impact of the proposed tools on a legacy code we employ the C code which is automatically generated by a state-of-the-art compiler for Modelica. Modelica is a widely-used object-oriented language for modeling and simulation of complex systems. OpenModelica [20] is an open source compiler for the Modelica language. It translates Modelica code into C code, which is later compiled with `clang` and linked against an external equation solver library.

As test case, we simulated a transmission line model [21] of 1000 elements. We modified the C and Makefile code automatically generated by the OpenModelica compiler to integrate the simulation C source code with LIBVC and properly compile it. It took two hours of work to integrate the automatically generated code with the LIBVC. The integration required to add or modify a total of 65 lines of C code and 5 lines of Makefile code over an original code size of 633 390 lines of code, which is less than 0.015% of the code size.

The baseline code took 374.25 s before the integration. After the integration the simulation took 295.00 s – including the overhead for dynamic compilation – for a speedup of 1.27× with respect to the baseline. The speedup is achieved by recompiling the C code which implements the model description by using a deeper optimization level (`-O3`) with respect to the default one (`-O0`). In this case, the compilation time that it is spent on optimizations is widely paid back by a faster execution time.

5. Conclusions

We have presented LIBVC, a lightweight library to support continuous optimization in HPC environments. The tool is employed within the context of the ANTAREX project to optimize the execution of computationally intensive kernels that are repeatedly called within large scale applications with long execution times. While the library is designed to be integrated with other tools in the ANTAREX workflow, it can also be used as a standalone tool with minimal effort by application developers.

[4] https://github.com/skeru/polybench_libVC.

Acknowledgments

This work is partially supported by the European Union's Horizon 2020 research and innovation programme, under grant agreement No 671623, FET-HPC ANTAREX.

The authors wish to thank Danilo Labanca, Valentina Ionata, and Diego Gaboardi from Politecnico di Milano for providing the source code of the miniapp employed in Section 4.

The authors wish to thank Emanuele Baldino, and Francesco Casella from Politecnico di Milano for providing the Modelica source code of the Transmission Line example employed in Section 4.

References

[1] Ziegler W, D'Ippolito R, D'Auria M, Berends J, Nelissen M, Diaz R. Implementing a one-stop-shop providing SMEs with integrated HPC simulation resources using fortissimo resources. In: eChallenges e-2014 conference proceedings; 2014. p. 1–11.

[2] Koller B, Struckmann N, Buchholz J, Gienger M. Towards an environment to deliver high performance computing to small and medium enterprises. In: Sustained simulation performance 2015. Springer; 2015. p. 41–50.

[3] Reed DA, Dongarra J. Exascale computing and big data. Commun ACM 2015;58(7):56–68. http://dx.doi.org/10.1145/2699414.

[4] Kistler T, Franz M. Continuous program optimization: A case study. ACM Trans Program Lang Syst 2003;25(4):500–48. http://dx.doi.org/10.1145/778559.778562.

[5] Nuzman D, Eres R, Dyshel S, Zalmanovici M, Castanos J. JIT technology with C/C++: Feedback-directed dynamic recompilation for statically compiled languages. ACM Trans Archit Code Optim 2013;10(4):59:1–25. http://dx.doi.org/10.1145/2541228.2555315.

[6] Fahs B, Rafacz T, Patel SJ, Lumetta SS. Continuous optimization. In: Proceedings of the 32nd annual international symposium on computer architecture, ISCA '05. Washington, DC, USA: IEEE Computer Society; 2005. p. 86–97. http://dx.doi.org/10.1109/ISCA.2005.19.

[7] Benkner S, Franchetti F, Gerndt HM, Hollingsworth JK. Automatic application tuning for HPC architectures (Dagstuhl Seminar 13401). Dagstuhl Rep 2014;3(9):214–44. http://dx.doi.org/10.4230/DagRep.3.9.214.

[8] Chen H, Lu J, Hsu W-C, Yew P-C. Continuous adaptive object-code re-optimization framework. In: Yew P-C, Xue J, editors. Advances in computer systems architecture. Berlin, Heidelberg: Springer Berlin Heidelberg; 2004. p. 241–55. http://dx.doi.org/10.1007/978-3-540-30102-8_20.

[9] Basu P, Williams S, Straalen BV, Oliker L, Colella P, Hall M. Compiler-based code generation and autotuning for geometric multigrid on GPU-accelerated supercomputers. Parallel Comput 2017;64(Suppl. C):50–64. http://dx.doi.org/10.1016/j.parco.2017.04.002. High-end computing for next-generation scientific discovery.

[10] Cohen J, Rayna T, Darlington J. Understanding resource selection requirements for computationally intensive tasks on heterogeneous computing infrastructure. In: Bañares JÁ, Tserpes K, Altmann J, editors. Economics of grids, clouds, systems, and services. Cham: Springer International Publishing; 2017. p. 250–62. http://dx.doi.org/10.1007/978-3-319-61920-0_18.

[11] Chen Y, Huang Y, Eeckhout L, Fursin G, Peng L, Temam O, et al. Evaluating iterative optimization across 1000 datasets. In: Proceedings of the 31st ACM SIGPLAN conference on programming language design and implementation, PLDI '10. New York, NY, USA: ACM; 2010. p. 448–59. http://dx.doi.org/10.1145/1806596.1806647.

[12] Tartara M, Crespi Reghizzi S. Continuous learning of compiler heuristics. ACM Trans Archit Code Optim 2013;9(4):46:1–25. http://dx.doi.org/10.1145/2400682.2400705.

[13] Gadioli D, Palermo G, Silvano C. Application autotuning to support runtime adaptivity in multicore architectures. In: Embedded computer systems: Architectures, modeling, and simulation (SAMOS), 2015 international conference on. IEEE; 2015. p. 173–80.

[14] Fursin G, Lokhmotov A, Plowman E. Collective knowledge: Towards R&D sustainability. In: Proceedings of the conference on design, automation and test in Europe (DATE'16); 2016. p. 864–69.

[15] Cohen A, Rohou E. Processor virtualization and split compilation for heterogeneous multicore embedded systems. In: Design automation conference; 2010. p. 102–107. http://dx.doi.org/10.1145/1837274.1837303.

[16] Yuki T. Understanding PolyBench/C 3.2 kernels. In: International workshop on polyhedral compilation techniques (IMPACT); 2014.

[17] Silvano C, Agosta G, Cherubin S, Gadioli D, Palermo G, Bartolini A, et al. The ANTAREX approach to autotuning and adaptivity for energy efficient HPC systems. In: Proceedings of the ACM international conference on computing frontiers, CF '16. New York, NY, USA: ACM; 2016. p. 288–93. http://dx.doi.org/10.1145/2903150.2903470.

[18] Silvano C, Agosta G, Bartolini A, Beccari AR, Benini L, Bispo J. et al., Autotuning and adaptivity approach for energy efficient exascale HPC systems: The ANTAREX approach. In: Proceedings of the 2016 conference on design, automation & test in Europe. DATE '16; 2016. p. 708–13.

[19] Beccari AR, Cavazzoni C, Beato C, Costantino G. Ligen: A high performance workflow for chemistry driven de novo design. ACS Publications; 2013.

[20] Fritzson P, Aronsson P, Pop A, Lundvall H, Nystrom K, Saldamli L. Openmodelica - a free open-source environment for system modeling, simulation, and teaching. In: 2006 IEEE conference on computer aided control system design, 2006 IEEE international conference on control applications, 2006 IEEE international symposium on intelligent control; 2006. p. 1588–95. http://dx.doi.org/10.1109/CACSD-CCA-ISIC.2006.4776878.

[21] Casella F, Simulation of large-scale models in modelica: State of the art and future perspectives. In: Linköping electronic conference proceedings; 2015. p. 459–468.