Continuous Program Optimization via Advanced Dynamic Compilation Techniques

Marco Festa, Nicole Gervasoni, Stefano Cherubin, Giovanni Agosta
Politecnico di Milano, DEIB
{marco2.festa,nicoleannamaria.gervasoni}@mail.polimi.it,stefano.cherubin@polimi.it,agosta@acm.org

ABSTRACT

In High Performance Computing, it is often useful to fine tune an application code via recompilation of specific computational intensive code fragments to leverage runtime knowledge. Traditional compilers rarely provide such capabilities, but solutions such as libVC exist to allow C/C++ code to employ dynamic compilation. We evaluate the impact of the introduction of Just-in-Time compilation in a framework supporting partial dynamic (re-)compilation of functions to provide continuous optimization in high performance environments. We show that Just-In-Time solutions can have comparable performance in terms of code quality with respect to the libVC alternatives, and it can provide smaller compilation overhead. We further demonstrate the strength of our approach against another interpreter-based dynamic evaluation solution from the state-of-the-art.

KEYWORDS

Dynamic Compilation, JIT, Continuous Program Optimization

ACM Reference Format:
Marco Festa, Nicole Gervasoni, Stefano Cherubin, Giovanni Agosta. 2019. Continuous Program Optimization via Advanced Dynamic Compilation Techniques. In *Proceedings of PARMA-DITAM Workshop (PARMA-DITAM 2019).* ACM, New York, NY, USA, 6 pages. https://doi.org/00.001/000_1

1 INTRODUCTION

The trend towards the democratisation of access to High Performance Computing (HPC) infrastructures [1; 2] is leading a wider spectrum of developers to work on applications that will run on complex, potentially heterogeneous architectures. Yet, the design and the implementation of HPC applications are difficult tasks for which several tools and languages are used [3; 4]. Typically, each HPC center has its own set of tools, which collect the expertise and work of many experts across the years. Such tools can go from extensive frameworks, such as the OmpSs/Nanos++/Mercurium tool set from Barcelona Supercomputing Center[1], to simpler collections of scripts, which are nonetheless critical to achieve the expected performance [5]. Thus, specialised help is often needed in the form of HPC center staff, developers who are accustomed to the practices of HPC systems and can provide expert knowledge to support the application domain expert. Still, the same democratisation trend, together with the growth of HPC infrastructures toward the Exascale, will strain the ability of HPC centers to provide sufficient personnel for these activities, due to the increased number of users.

To ease this strain, practices such as autotuning and continuous optimisation [6–8] can be successfully applied to make the application itself more aware of its performance and able to cope with platform heterogeneity and workload changes which are not predictable at compile-time, and for which traditional techniques such as profile-guided optimisation may fail due to the difficulty of finding small profile data sets that are representative of the large ones actually used in the HPC runs. In these cases, which are becoming more and more common [9], dynamic approaches can prove more effective.

The operation of such a dynamic approach chiefly consists in generating more than one version of the code of compute-intensive kernel, and then selecting the best version at each invocation of the kernel. The selection can be performed as part of an autotuning algorithm, which can be used both to tune software parameters and to search the space of compiler optimizations for optimal solutions [10]. 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. Some frameworks are actually able to perform continuous optimization, generally through specific versions of a dynamic compiler [11; 12], or through cloud-based platforms [13], or by leveraging an external compiler through a dedicated API [14].

In this paper, we introduce an extension of a state-of-the-art dynamic compilation library to support the *Just In Time* (JIT) compilation paradigm. We discuss the implementation of easy-to-use APIs to realize the continuous optimization approach. Finally, we compare the dynamic compilation overhead due to this kind of compilation technique with other dynamic compilation techniques in the state-of-the art.

The rest of this paper is organized as follows. Section 2 described the Just In Time paradigm, and the implementation of the related APIs within a C++ library. Section 3 discusses the experimental results. Section 4 provides an overview of the related works and a comparative analysis with them. Finally, we draw some conclusions in section 5.

2 A JUST-IN-TIME SOLUTION

Continuous optimization requires the program to re-shape portions of its own executable code to adapt them to runtime conditions. This capability is very common among interpreted programming languages, as the instructions are usually parsed and issued at runtime. In the context of compiled programming languages, instead, it is more complex to achieve this behaviour, as it requires to defer the full compilation process at runtime. Although interpreted languages provide greater flexibility, they typically have throughput

[1] https://pm.bsc.es

PARMA-DITAM 2019, January 2019, Valencia, ES

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/00.001/000_1

which is much lower with respect to compiled ones. Thus, in HPC systems it is preferable to have a reconfiguration time span to optimize the software rather than having a constant overhead on each instruction.

2.1 Continuous Optimization via Dynamic Compilation

Dynamic compilation techniques allow to compile source code into executable code after the software itself has been deployed. Whenever important runtime conditions or checkpoints are met, a dynamic re-configuration of the software system may entail a dynamic (re-)compilation of its source code to apply a different set of optimization which better fit the incoming workload. Dynamic compilation can be performed via ad-hoc software hypervisors, via dynamic generation and loading of software libraries, or via the integration of a compiler stack within the adaptive software system itself.

The former approach suffers from the difficulty of maintaining both the hypervisor and the code to use it. This approach requires a deep knowledge of the hypervisor system, and an accurate configuration over the software system.

The dynamic generation and loading of software libraries is a platform-dependent solution which requires fine tuning of the compiler configuration at deploy time. Moreover, this approach requires to access and to manage additional persistent memory space to handle the dynamically-generated shared objects. This problem has a non-trivial impact on HPC infrastructures, as such systems usually aims at minimizing the access to persistent memory due to its intrinsic high-latency. Although recent proposals have been made to simplify the configuration of compiler settings [14], the limitation given by the memory access still persists. The JIT paradigm removes the problems of the abovementioned approaches, as it does not create any persistent object and it integrates the full compiler functionalities within the adaptive application.

2.2 Principles of Just-in-Time Compilation

In JIT compilation, a fragment of code (usually a function or method) is only compiled when it is first executed. The main issue with JIT compilation is that at each new function encountered, a compilation latency is incurred, leading to potentially large start-up times.

JIT compilers are mainly interesting to combine some properties of static compilation, typically the performance of the generated code, with other properties which are typical of interpreters, such as the ability to leverage runtime knowledge about the program (runtime constant, control flow) or portability. In the context of continuous optimization, JIT compilation is a key enabler, since recompiling the entire program while it is running does not help the running instance, whereas a JIT compiler can replace compiled code fragments with new versions tuned to different parameters. It is worth noting that, while a JIT generally does not preserve the compiler code, this is not a limitation in the context of HPC systems, where a given application may run over a long time, but may be invoked only a few times, thus making the persistance of the optimized code less relevant.

2.3 Providing JIT APIs via LLVM

In this work, we leverage the libVC framework, which provides APIs to enable dynamic compilation and re-use of code versions. It

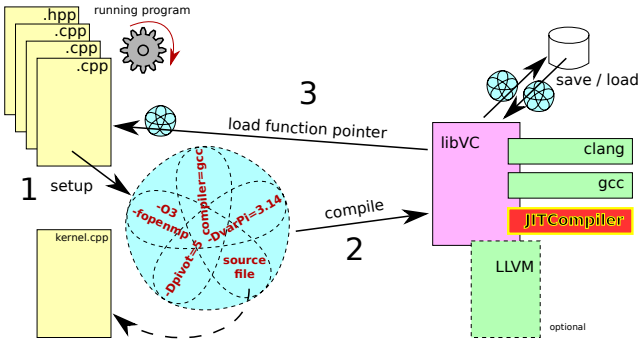

Figure 1: Continuous Program Optimization flow with libVC infrastructure, including the proposed extension

features three different implementation of compiler APIs: `SystemCompiler`, `SystemCompilerOptimizer`, and `ClangLibCompiler`.

The first two solutions require to be configured to properly interact with external compilers already deployed on the host machine. The latter implements the *Compiler as a library* paradigm, and therefore needs llvm to be installed in the host machine. We extend libVC with `JITCompiler`, an additional implementation for the `Compiler` interface, with the aim of providing true JIT compilation capabilities. Figure 1 shows the infrastructure of libVC when it is used to perform continuous program optimization. We highlight the new component with a red box under the other alternatives provided by the framework.

To pursue continuity with the original implementation of libVC, we base our implementation of `JITCompiler` on the llvm compiler framework. In particular, we support the generation of llvm-ir bitcode files, the optimization of such intermediate representation, and the compilation of it into executable code. Whilst the bitcode generation and optimization are implemented similarly to the *Clang as a library* paradigm, the generation of executable code is extremely different and it represents the core of the `JITCompiler` approach. To achieve such improvement we leverage llvm's On-Request-Compilation (ORC) APIs. Our new libVC compiler implementation allows us to parse the bitcode files and to elevate them to their in-memory representation. The freshly imported llvm Module objects are later passed as arguments to the core llvm ORC API handle `addModule`, which actually starts the JIT compilation process. Similar APIs allow us to fetch one or more symbols from the jitted code and use them in the same way libVC provides function pointers to the host code, so they will be able to access the dynamically compiled version of the code via indirect function calls. Like the other compiler implementations defined in libVC, we provide a mechanism to control the memory footprint of the dynamic compilation framework by exposing knobs to offload and to reload from the memory the dynamically compiled modules. This approach is fully compliant with the state-of-the-art libVC paradigma to perform continuous optimization.

3 EXPERIMENTAL EVALUATION

We evaluate our implementation of the llvm-based JIT compiler by comparing its capabilities with the alternatives which are already available within libVC [14]. Our solution exposes the same APIs as the other compiler implementations within libVC. Thus, the

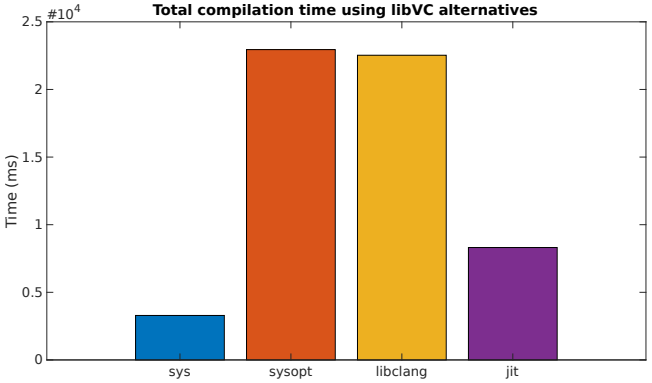

Figure 2: Compilation time of the whole PolyBench/C benchmark suite with different implementation alternatives from LIBVC.

usability of this compiler, and its fitness to continuous optimization use cases are the same as of its alternatives. In this section, we focus on two main performance indicators, namely **code quality** and **compilation time**. To this end, we compare our solution against the other LIBVC compiler implementations over the well-known PolyBench/C benchmark suite [15]. We use this benchmark suite as a proxy to validate the effectiveness of our approach as it is particularly representative of the typical HPC workload. In fact, it contains kernels from several application domain, such as linear algebra, data mining, and image processing, which are the core emerging areas in the HPC world.

The platform used to run the experiments is representative of modern supercomputer nodes. The selected hardware is a NUMA node with two Intel Xeon E5-2630 V3 CPUs (@3.2 GHz) for a total of 16 cores, with hyper threading enabled and 128 GB of DDR4 memory (@1866 MHz) on a dual channel memory configuration. The operating system is Ubuntu 16.04 with version 4.4.0 of the Linux kernel. The experiments run with the machine completely unloaded from other user processes.

We based our tests on the official LIBVC testing setup for PolyBench/C[2]. We configured LIBVC to use the following compilers:

SystemCompiler default host compiler (`gcc` version 5.4.0)
SystemCompilerOptimizer `clang` and `opt`, version 6.0.1
ClangLib `libclang` version 6.0.1
JITCompiler our solution, based on LLVM version 6.0.1

We configured each compiler to run with the same set of compiler options, i.e. we specify only the optimization level `-O3`. We rely on the MEDIUM dataset size, and on the default initialization routines which are defined by the PolyBench/C benchmark suite for each kernel. All kernels use the IEEE-754 floating point double precision data type for the computation.

3.1 Code Quality

We expect to have the same code quality given by the same compiler with the same compiler options. To verify this assertion we compiled and run the PolyBench/C benchmarks and we report in Figure 3 runtime and compilation time for each benchmark. The run time of the different code versions generated using LIBVC compilers is not significantly influenced by the chosen compiler alternative. On the contrary, this choice becomes relevant for the compilation time. Figure 2 bar chart highlights how the `JITCompiler` compiler performance outstands the other clang-based alternatives. This is confirmed by every benchmark in the polybench suite on IEEE-754 double precision type data. Thus, the code quality is not significantly influenced by the compiler choice.

3.2 Compilation Time

The main overhead that applies to continous program optimization via dynamic compilation is given by the compilation time. As the code quality is not influenced by the chosen compiler, the conditions that trigger a re-compilation task do not change. The compilation time of the single kernel is not enough significative per se. Thus, we compare the compilation time of each compiler implementation when it is asked to compile the whole set of PolyBench/C benchmarks. Figure 2 underlines the improved performance of the `JITCompiler` when compared with other clang-based compilers. The `SystemCompiler` is based instead on the `gcc` compiler technology, which performs better in terms of compilation time. Figure 3 confirms that this trend holds for each kernel.

4 RELATED WORKS

The history of JIT compilation is almost as old as computer science. Aycook summarizes early works on JIT compilation techniques starting from the 1960s in his survey [16]. In modern times, the Sun Microsystems Java HotSpot Virtual Machine [17] started to extensively use JIT technologies by running both an interpreter and a compiler, the latter invoked on hot-spots [18].

More recently, the *Graal Project*[3] aims at leveraging the JIT technologies to improve performance of several interpreted, bitcode-interpreted, and compiled languages. In particular, the *GraalVM*[4] is the most relevant outcome of this project. It embraces JIT compilation as the key-stone of the framework in order to enable several speculative optimizations [19]. *GraalVM* has been used to provide JIT support to a wide range of programming languages, from domain specific languages [20] to popular languages, such as JavaScript and C/C++ [21]. However, it relies on LLVM-based solutions to dynamically compile C/C++ source code to LLVM-IR (`clang` front end), and to interpret LLVM-IR (`lli` interpreter). Thus, *GraalVM* can be considered technologically equivalent to the LLVM compiler toolchain we evaluated via LIBVC.

Another effort to enable dynamic compilation over C/C++ code is represented by `Cling` [22] — the `clang`-based C++ interpreter. Since this project implements the Read-Eval-Print-Loop (REPL) paradigm over C++ source code, it is the closest effort to a pure C++ JIT compiler in the state-of-the-art. In the rest of this section we further discuss `Cling` and its usage. Finally, we provide a comparative analysis between our proposed solution and `Cling`.

4.1 Cling

`Cling` originates from the need to process vast amount of data with the capabilities of a compiled language with a strong focus on code efficiency, such as C++. It was developed as a part of CERN's data

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

[3] Oracle Graal project https://openjdk.java.net/projects/graal/
[4] GraalVM https://www.graalvm.org

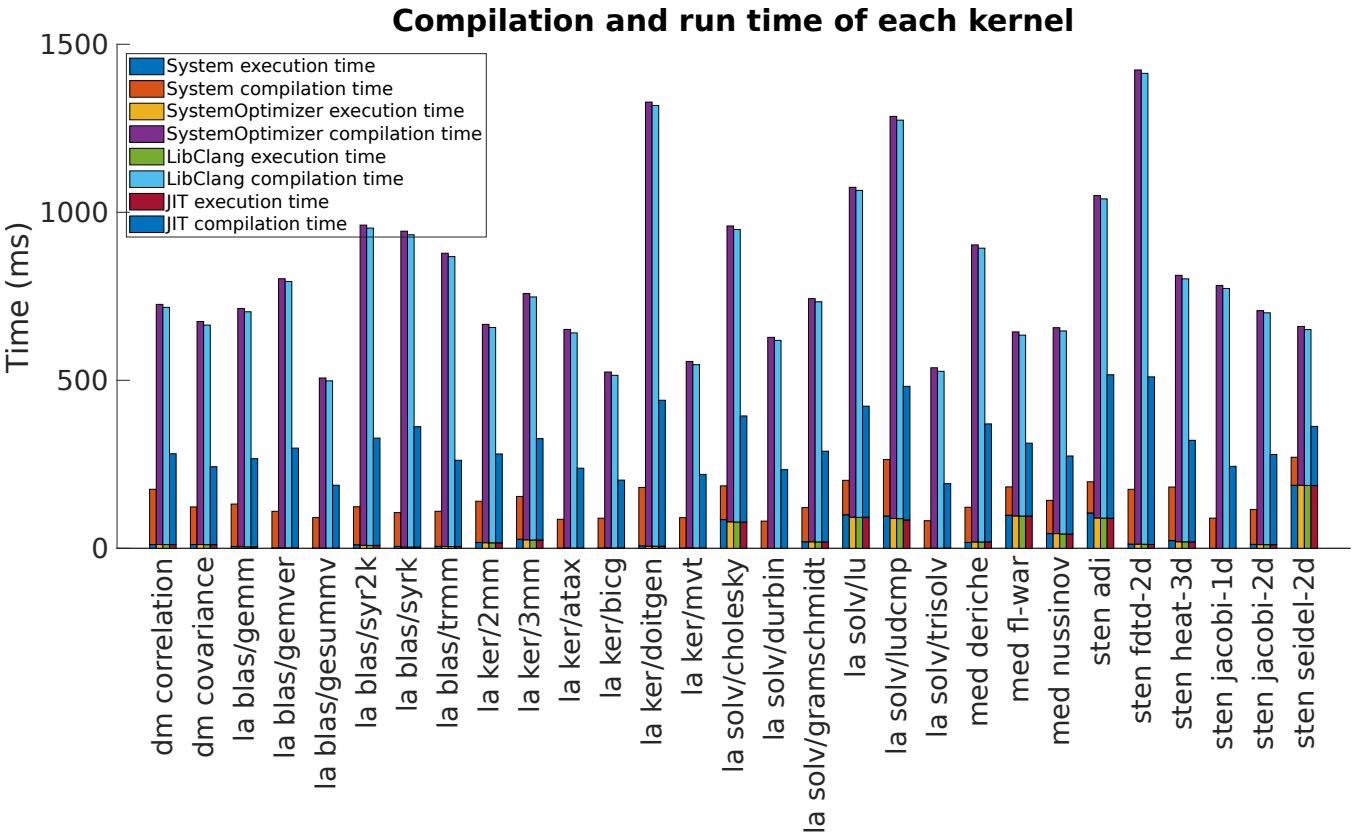

Figure 3: Compile time and run time of each kernel of the PolyBench/C benchmark suite with different implementation alternatives from libVC.

processing framework ROOT. Cling's core objective is to provide a fast and interactive way to run applications able to access experimental results generated by the high-energy-physics research community. A structural analysis of Cling highlights the 3 fundamental parts it is composed of:

Interpreter which implements the parsing, JIT compilation, and evaluation of native C++ capabilities

Meta Processor which provides a command line interface to send commands to the interpreter

User Interface providing the interactive prompt, and an exception handling mechanism

Cling's interpreter is based on the version 5.0.0. of LLVM, and on its C++ frontend Clang. While our `JITCompiler` leverages the `IRCompileLayer` class to provide jitting, Cling uses the `LazyEmitLayer` one. In the former case the behaviour of the class is to immediately JIT the LLVM Module as soon as this is passed to the `addModule` API, the latter instead explicitly waits the symbol to be called to begin compiling.

Given Cling's intrinsic REPL-based implementation, to measure its performance we exploited the bash's 'time' utility to extract the precise execution time lapse of the evaluation of a single kernel. To exclude the overhead due to other Cling's components initialization (such as the User Interface) we run the Interpreter several times with no instructions but the `.q` (exit) directive. In this way, we collected the average time lapse of the setup and the tear down routines of Cling.

Moreover, we separate compilation and execution times, which are indistinguishable due to the lazyness of Cling's jitting strategy, by running the interpreter multiple times. This procedure allows us to infer the single compilation-only time lapse for each kernel.

4.2 Comparative Analysis

We compare Cling with our JIT implementation within libVC by scheduling the run of each kernel from the PolyBench/C benchmark suite. We rely on the same hardware and software setup described in Section 3. We measured execution time and the (re-)compilation overhead. Since Cling does not support any code optimization level, we slightly modified the previously-described experimental setup to allow a fair comparison between Cling and `JITCompiler`. In particular, we disabled any optimization in our `JITCompiler` via the `-O0` compiler option. We collected data using the `MEDIUM` dataset size preset and the IEEE-754 floating point single precision data type for the computation.

In Figure 4 we compare `JITCompiler` from libVC with the external tool Cling. As expected, the compiling phase is generally a more time requesting task with respect to the execution one for the PolyBench/C benchmark suite with the `MEDIUM` data set. Figure 4 clearly shows that the compiling plus execution time of

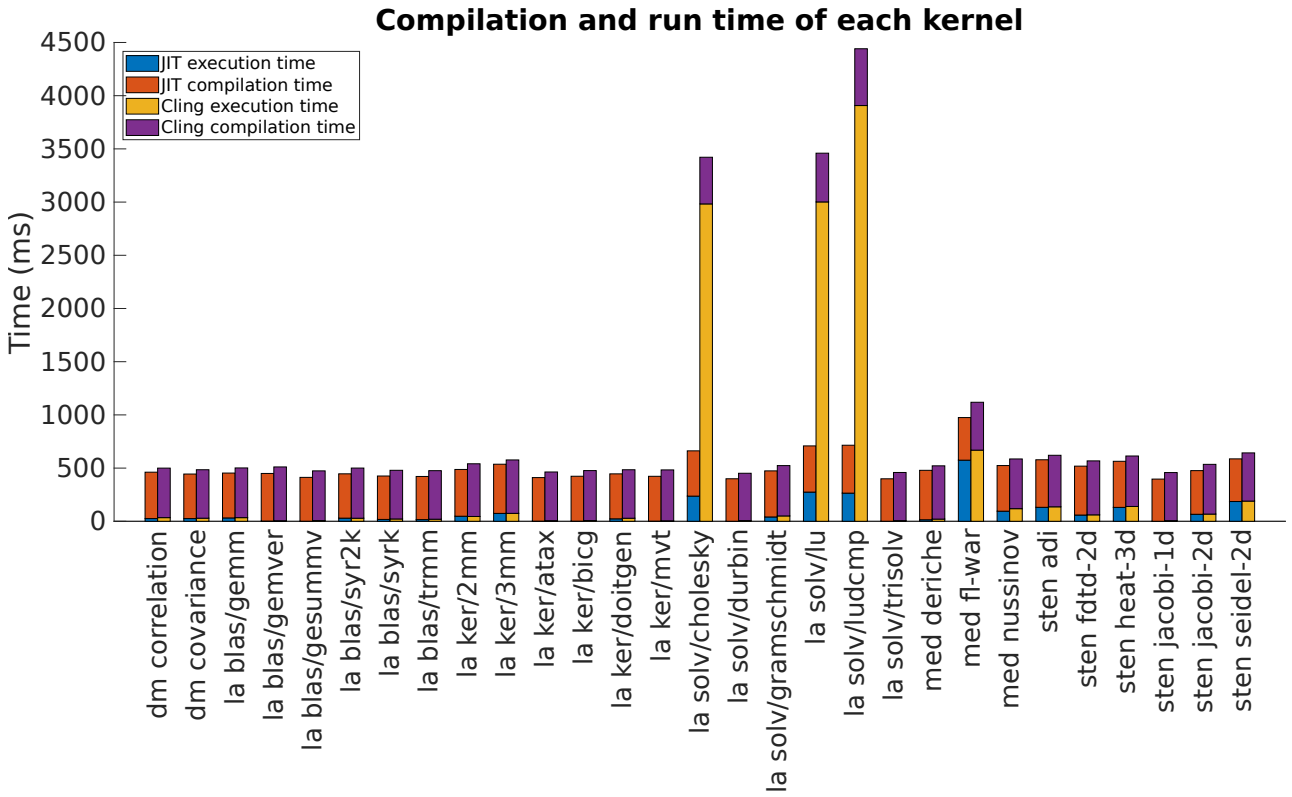

Figure 4: Compilation and execution time of the overall PolyBench/C benchmark suite with Cling and with JITCompiler.

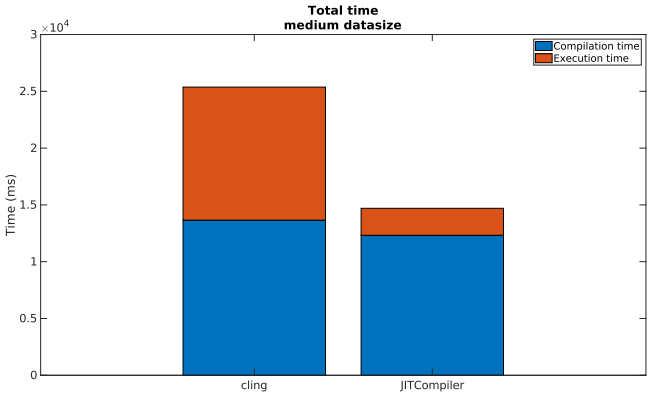

Figure 5: Compilation and execution time of the overall PolyBench/C benchmark suite with Cling and with JITCompiler.

the JITCompiler outperforms the Cling interpreter even when the former does not schedule any optimization, i.e. it runs with -O0 optimization level.

Figure 5 better highlights the saving coming from the use of JITCompiler instead of the lazy interpretation of C++ code given by Cling, as it shows the overall amount of time spent compiling and running once the full PolyBench/C benchmark suite.

Figure 6 shows the compilation time plus the execution time of Cling compared with the equivant of the JITCompiler when the latter is using the -O3 optimization level. Running the compiled code several times allows us to analyze the actual speedup given by the JICompiler. Although the total compilation time, as seen in Figure 5, does not show astonishing differences, with an increased number of invocation of the same kernel the JITCompiler optimization capabilities — which are paid by longer compilation times — allows the system to reach better performance.

5 CONCLUSIONS

We extended the libVC framework with JIT compilation capabilities, leveraging the llvm compiler framework. We compare the resulting dynamic compiler with the pre-existing compilation options provided by libVC, which are based on standard compilers (Clang as a library, gcc, clang + opt). We show how the JIT integration improves the performance of code compiled with libVC on the PolyBench/C suite.

Regardless the different approaches to dynamically run C++ code (REPL VS APIs), libVC's JITCompiler outperforms Cling when it is invoked from an host application, as in the case of continuous program optimization via dynamic compilation.

ACKNOWLEDGMENTS

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

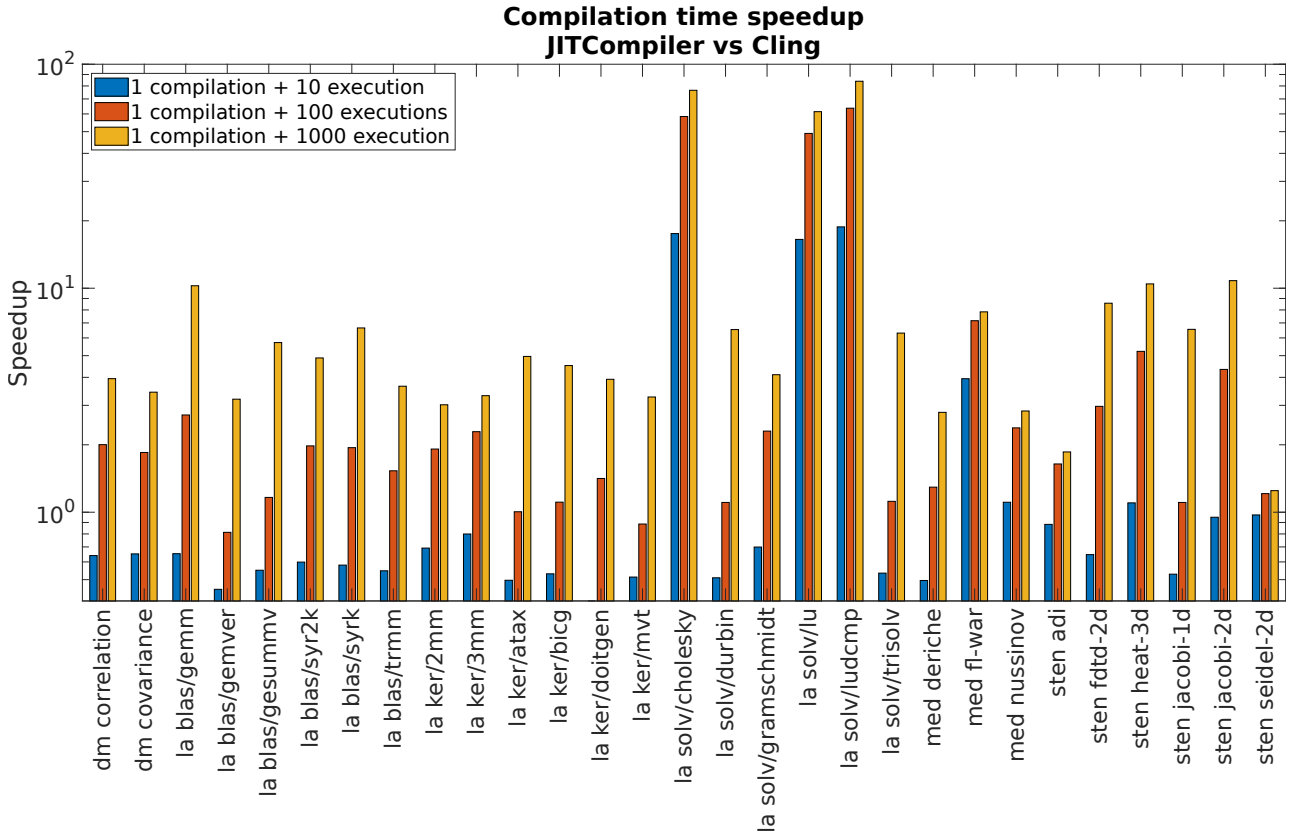

Figure 6: Speedup achieved by using JITCompiler instead of Cling with 10, 100, and 1000 invocations of the kernel.

REFERENCES

[1] W. Ziegler, R. D'Ippolito, M. D'Auria, J. Berends, M. Nelissen, and R. Diaz. Implementing a "one-stop-shop" providing smes with integrated hpc simulation resources using fortissimo resources. In *eChallenges e-2014 Conf. Proceedings*, pages 1–11, Oct 2014.

[2] Bastian Koller, Nico Struckmann, Jochen Buchholz, and Michael Gienger. Towards an environment to deliver high performance computing to small and medium enterprises. In *Sustained Simulation Performance 2015*, pages 41–50, 2015.

[3] Cristina Silvano et al. The ANTAREX approach to autotuning and adaptivity for energy efficient hpc systems. In *Proc. of the ACM Internat. Conf. on Computing Frontiers*, pages 288–293, 2016.

[4] Cristina Silvano et al. The ANTAREX tool flow for monitoring and autotuning energy efficient HPC systems. In *Internat. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 308–316, July 2017.

[5] Cristina Silvano et al. ANTAREX: A DSL-based approach to adaptively optimizing and enforcing extra-functional properties in high performance computing. In *21st Euromicro Conf. on Digital System Design (DSD)*, pages 600–607, Aug 2018.

[6] Thomas Kistler and Michael Franz. Continuous program optimization: A case study. *ACM Trans. Program. Lang. Syst.*, 25(4):500–548, jul 2003.

[7] Dorit Nuzman, Revital Eres, Sergei Dyshel, Marcel Zalmanovici, and Jose Castanos. Jit technology with c/c++: Feedback-directed dynamic recompilation for statically compiled languages. *ACM Trans. Archit. Code Optim.*, 10(4):59:1–59:25, dec 2013.

[8] Brian Fahs, Todd Rafacz, Sanjay J. Patel, and Steven S. Lumetta. Continuous optimization. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 86–97, 2005.

[9] Daniel A. Reed and Jack Dongarra. Exascale computing and big data. *Communications of the ACM*, 58(7):56–68, jun 2015.

[10] Siegfried Benkner, Franz Franchetti, Hans Michael Gerndt, and Jeffrey K Hollingsworth. Automatic Application Tuning for HPC Architectures (Dagstuhl Seminar 13401). *Dagstuhl Reports*, 3(9):214–244, 2014.

[11] Howard Chen, Jiwei Lu, Wei-Chung Hsu, and Pen-Chung Yew. Continuous adaptive object-code re-optimization framework. In Pen-Chung Yew and Jingling Xue, editors, *Advances in Computer Systems Architecture*, pages 241–255, 2004.

[12] Protonu Basu et al. Compiler-based code generation and autotuning for geometric multigrid on gpu-accelerated supercomputers. *Parallel Computing*, 64(Supplement C):50 – 64, 2017.

[13] Jeremy Cohen, Thierry Rayna, and John Darlington. Understanding resource selection requirements for computationally intensive tasks on heterogeneous computing infrastructure. In José Ángel Bañares, Konstantinos Tserpes, and Jörn Altmann, editors, *Economics of Grids, Clouds, Systems, and Services*, pages 250–262, 2017.

[14] Stefano Cherubin and Giovanni Agosta. libVersioningCompiler: An easy-to-use library for dynamic generation and invocation of multiple code versions. *SoftwareX*, 7:95 – 100, 2018.

[15] Tomofumi Yuki. Understanding PolyBench/C 3.2 kernels. In *International workshop on Polyhedral Compilation Techniques (IMPACT)*, 2014.

[16] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, jun 2003.

[17] Sun Microsystems Java team. The Java HotSpot Virtual Machine, v1.4.1.

[18] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspottm server compiler. In *Proc. of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, 2001.

[19] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. Speculation without regret: Reducing deoptimization meta-data in the graal compiler. In *Proc. of the 2014 International Conf. on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 187–193, 2014.

[20] Vlad Vergu and Eelco Visser. Specializing a Meta-Interpreter: JIT compilation of DynSem specifications on the graal vm. In *Proceedings of the 15th International Conf. on Managed Languages & Runtimes*, ManLang '18, pages 16:1–16:14, 2018.

[21] Matthias Grimmer, Manuel Rigger, Roland Schatz, Lukas Stadler, and Hanspeter Mössenböck. Trufflec: Dynamic execution of c on a java virtual machine. In *Proc. of the 2014 Internat. Conf. on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 17–26, 2014.

[22] V Vasilev, Ph Canal, A Naumann, and P Russo. Cling – the new interactive interpreter for root 6. *Journal of Physics: Conf. Series*, 396(5):052071, 2012.