Chapter 4. Split Compilation: the libVersioningCompiler Approach

Stefano Cherubin and Giovanni Agosta

Politecnico di Milano, Italy
{stefano.cherubin, giovanni.agosta}polimi.it

Abstract

libVersioningCompiler is a library that allows partial dynamic recompilation within an application. It is designed to ease the burden of performing continuous program optimization within the context of High Performance Computing applications. In the context of the ANTAREX tool flow, libVersioningCompiler can be employed through the ANTAREX DSL, so that its operation can be combined with that of other components of the toolchain, to achieve fine tuning of compilation options and code version management.

1 Motivation

Designing and implementing High Performance Computing (HPC) applications is a difficult and complex task that requires mastery of 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.

Profile-guided code transformations at compile-time usually provide a good optimization level in a general-purpose scenario. On the contrary, in HPC scenarios where large data sets are employed, a proper profiling may be unfeasible. 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, 5, 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 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 chapter is organized as follows. In section 2 we discuss the background and related works on partial dynamic recompilation and split compilation. In section 3.1 we describe the software architecture, the internal APIs and their functionalities. In section 3.2 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. Finally, we draw some conclusions in section 5.

2 Background and Related Works

Partial dynamic (re)compilation is a technique used as part of continuous program optimisation [4]. It allows the compiler to further optimize the code, during the execution of long runs of an application, which are typical of HPC scenarios. While most high level languages include mechanisms for selective compilation which can be exploited for fine tuning the dynamic compilation options, e.g. for hiding compilation latencies [11], the C/C++ applications commonly used in HPC scenarios generally lack this option. Some support is provided by domain-specific tools, such as RuntimeCompiledC++ [12], which focuses on interactive modification and recompilation of program fragments by the programmer. To enable partial dynamic compilation, ANTAREX DSL aspects can introduce calls to a support library, libVC [13], allowing to weaken the boundary between compile-time and runtime, and enabling continuous optimization.

3 libVersioningCompiler

3.1 Software Architecture

The goal of libVC is to allow C/C++ compute kernels 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 [14, 15].

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.

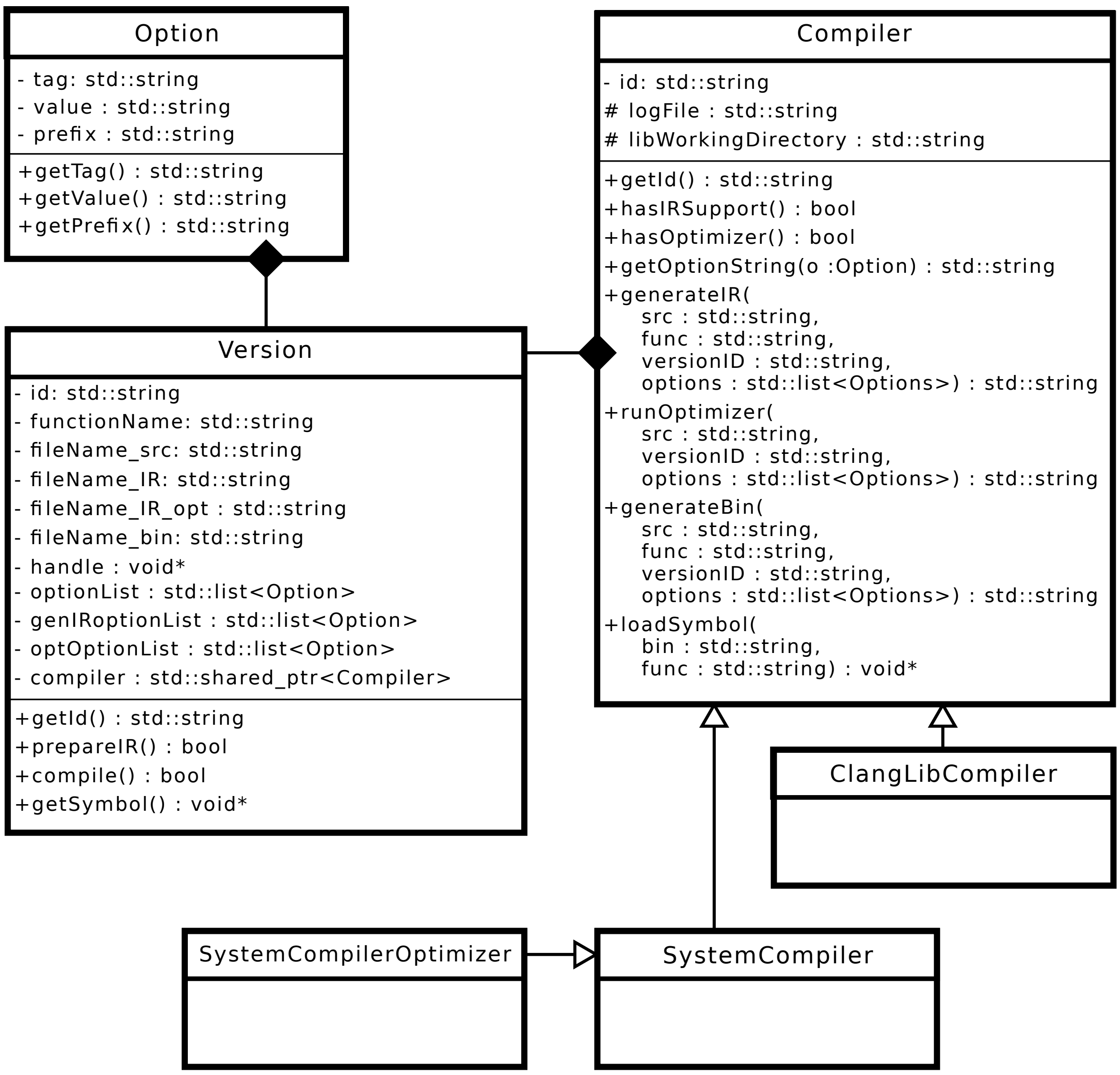
Option
- tag: std::string
- value : std::string
- prefix : std::string
+getTag() : std::string
+getValue() : std::string
+getPrefix() : std::string
Compiler
- id: std::string
# logFile : std::string
# libWorkingDirectory : std::string
+getId() : std::string
+hasIRSupport() : bool
+hasOptimizer() : bool
+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

Figure 1: Simplified UML class diagram of libVC

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.

Description of the software model Figure 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: `System-`

Compiler, 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.

3.2 Usage Examples

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 [16] or cTuning [17].

libVC comes with 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 [18]; 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 the 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 object libVC loads one or more function pointer symbols, which point to the kernel functions in the shared object.

The target kernels 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.

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

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 each kernel, which have to be used for a proper function pointer cast.

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

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: 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;

```

[2] `https://github.com/skeru/libVersioningCompiler`

```
39    // initialize libVersioningCompiler
40    vc::vc_utils_init();
41
42    for (const auto range : data_range) {
43      TimeMonitor time_monitor_ref;
44      TimeMonitor time_monitor_dyn;
45      TimeMonitor time_monitor_ovh;
46
47      // running reference version - statically linked
48      for (size_t i = 0; i < iterations; i++) {
49        // produce workload to process
50        auto wl = WorkloadProducer<int32_t>::get_WL_with_bounds(range.first, range
            .second);
51        const auto meta = wl.getMetadata();
52        time_monitor_ref.start();
53        sort(wl.data, meta.minVal, meta.maxVal); // call reference
54        time_monitor_ref.stop();
55      }
56
57      // measuring overhead of preparing a new version - start
58      time_monitor_ovh.start();
59      auto v = getDynamicVersion(range.first, range.second);
60      kernel_t my_sort = (kernel_t) v->getSymbol(0);
61      time_monitor_ovh.stop();
62      // measuring overhead of preparing a new version - end
63
64      // running dynamic version - dynamically compiled
65      for (size_t i = 0; i < iterations; i++) {
66        // produce workload to process
67        auto wl = WorkloadProducer<int32_t>::get_WL_with_bounds(range.first, range
            .second);
68        time_monitor_dyn.start();
69        my_sort(wl.data); // just a call to a function pointer
70        time_monitor_dyn.stop();
71      }
72
73      // consider average time-to-solution
74      std::cout << range.second << "␣" << time_monitor_ref.getAvg() << "␣" <<
            time_monitor_dyn.getAvg() << "␣" << time_monitor_ovh.getAvg() << std::
            endl;
75    }
76    return 0;
77  }
```

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).

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[3] where we dynamically compile and run the full PolyBench/C [19] benchmark suite within the same C++ program.

[3]`https://github.com/skeru/polybench_libVC`

3.3 ANTAREX DSL Integration

One of the strategies supported in the ANTAREX toolflow is the capability to generate versions of a function and to select the one that satisfies certain requirements at runtime. Figure 2 shows an aspect that clones a set of functions and changes the types of the newly generated clones. Each clone has the same name as the original with the addition of a provided suffix. We start with a single user-defined function which is cloned by the aspect `CloneFunction` (called in line 13). Then, it recursively traverses calls to other functions inside the clone and generates a clone for each of them. Inside the clones, calls to the original functions are changed to calls to the clones instead, building a new call tree with the generated clones. At the end of the aspect `CreateFloatVersion` (lines 16–17,) we use the previously defined `ChangePrecision` aspect to change the types of all generate clones.

```
 1 import ChangePrecision;
 2 import clava.ClavaJoinPoints;
 3 
 4 aspectdef CreateFloatVersion
 5   input $func, suffix end
 6   output $clonedFunc end
 7 
 8   $double = ClavaJoinPoints.builtinType('double');
 9   $float = ClavaJoinPoints.builtinType('float');
10 
11   /* clone the target functions and the child calls */
12   var clonedFuncs = {};
13   call cloned : CloneFunction($func, suffix, clonedFuncs);
14 
15   /* change the precision of the cloned function */
16   for($clonedFunc of clonedFuncs)
17     call ChangePrecision($clonedFunc, $double, $float);
18 
19   $clonedFunc = cloned.$clonedFunc;
20 end
```

Figure 2: Example of LARA aspect to clone an existing function and change the type of the clone.

The aspect `Multiversion` – in Figure 3 – adapts the source code of the application in order to call the original version of a function or a generated cloned version with a different type, according to the value of a parameter given by the autotuner at runtime. The main aspect calls the previously shown aspect, `CreateFloatVersion`, which clones the target function and every other function it uses, while also changing their variable types from `double` to `float` (using the aspects presented in Figure 2).This is performed in lines 8–9 of the example. From lines 13 to 34, the `Multiversion` aspect generates and inserts code in the application that is used as switching mechanism between the two versions. It starts by declaring a variable to be used as a knob by the autotuner, then it generates the code for a switch statement and replaces the statement containing the original call with the generated switch code. Finally, in lines 36–38, the aspect surrounds both calls (original and float version) with timing code. An excerpt of the resulting C code can bee seen in Figure 4.

In the ANTAREX toolflow, the capability of providing several versions of the same function is not limited to static features. libVC enables the exploration and tuning of the parameter space of the compiler at runtime.

Figure 5 shows an example of usage of libVC through LARA, which demonstrates how to specialize a function. The user provides this aspect with a target function call and a set of compilation options. These include compiler flags and possible compiler definitions, e.g., data

```
import CreateFloatVersion;
import lara.code.Timer;
import clava.ClavaJoinPoints;

aspectdef Multiversion
  input $func, knobName end

  call fVersion : CreateFloatVersion($func, "_f");
  var $floatFunc = fVersion.$clonedFunc;
  var timer = new Timer();

   /* Identify call by name... */
   select function.body.stmt.call{$func.name} end
   apply
    /* ... and by type signature */
    if(!$func.functionType.equals($call.functionType))
      continue;

    /* Add knob for choosing the version */
    $int = ClavaJoinPoints.builtinType('int');
    $body.exec addLocal(knobName, $int, 0);

    /* create float declaration for first argument */
    var $arg = createFloatArg($call.args[0]);
    /* Create call based on float version of function */
    $floatFunc.exec $fCall : newCall([$arg, $call.args[1]]);
    /* Copy current call */
    $call.exec $callCopy : copy();

    /* Create switch */
    var $condition = ClavaJoinPoints.exprLiteral(knobName);
    var switchCases = {0: $callCopy, 1: $fCall};
    call switchJp : CreateSwitch($condition, switchCases);
    $stmt.exec replaceWith(switchJp.$switch);

    /* Time calls to both original and float functions*/
    timer.time($callCopy, "Original time:");
    timer.time($fCall, "Float time:");
   end
end
```

Figure 3: Example of LARA aspect that generates an alternative version of a function and inserts a mechanism in the code to switch between versions.

discovered at runtime, which is used as a compile-time constant in the new version. Based on the target function call, the aspect finds the function definition which is passed to the library. After the options are set, the original function call is replaced with a call of the newly compiled and loaded specialized version of the kernel.

It is worth noting that the combination of LARA and LIBVC can also be used to support compiler flag selection and phase-ordering both statically and dynamically.

4 Experimental Evaluation

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[4]. We run this example

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

```
switch (version) {
  case 0: {
    clock_gettime(CLOCK_MONOTONIC, &time_start_0);
    SumOfInternalDistances(atoms, 1000);
    clock_gettime(CLOCK_MONOTONIC, &time_end_0);
    double time_0 = calc_time(time_start_0, time_end_0);
    printf("Original time:%fms\n", time_0);
  }
  break;
  case 1: {
    clock_gettime(CLOCK_MONOTONIC, &time_start_1);
    SumOfInternalDistances_f(atoms_f, 1000);
    clock_gettime(CLOCK_MONOTONIC, &time_end_1);
    double time_1 = calc_time(time_start_1, time_end_1);
    printf("Float time::%fms\n", time_1);
  }
  break;
}
```

Figure 4: Excerpt of the C code resulting from the generation of alternative code versions.

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.

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

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`.

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 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.

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 [20] 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

```
import antarex.libvc.LibVC;

aspectdef SimpleLibVC

  input
    name, $target, options
  end

  var $function = $target.definition;
  var lvc = new LibVC($function, {logFile:"log.txt"}, name);

  var lvcOptions = new LibVCOptions();
  for (var o of options) {
    lvcOptions.addOptionLiteral(o.name, o.value, o.value);
  }
  lvc.setOptions(lvcOptions);

  lvc.setErrorStrategyExit();

  lvc.replaceCall($target);
end
```

Figure 5: Example of LARA aspect to replace a function call to a kernel with a call to a dynamically generated version of that kernel.

of HPC, and consists in finding the best fitting ligand molecule with a pocket in the target molecule [21]. 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 3.2. 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 seconds before the integration. After the integration the miniapp took 1783.93 seconds – 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 [22] 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 [23] 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 633390 lines of code, which is less than 0.015% of the code size.

The baseline code took 374.25 seconds before the integration. After the integration the simulation took 295.00 seconds – 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.

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 Conference 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. Springer, 2015.

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

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

[5] 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, December 2013.

[6] 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, Washington, DC, USA, 2005. IEEE Computer Society.

[7] 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.

[8] 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, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[9] Protonu Basu, Samuel Williams, Brian Van Straalen, Leonid Oliker, Phillip Colella, and Mary Hall. Compiler-based code generation and autotuning for geometric multigrid on gpu-accelerated supercomputers. *Parallel Computing*, 64(Supplement C):50 – 64, 2017. High-End Computing for Next-Generation Scientific Discovery.

[10] 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, Cham, 2017. Springer International Publishing.

[11] Simone Campanoni, Martino Sykora, Giovanni Agosta, and Stefano Crespi Reghizzi. Dynamic look ahead compilation: a technique to hide jit compilation latencies in multicore environment. In *International conference on compiler construction*, pages 220–235. Springer, 2009.

[12] Doug Binks, Matthew Jack, and Will Wilson. Runtime compiled c++ for rapid ai development. *Game AI Pro: Collected Wisdom of Game AI Professionals*, page 201, 2013.

[13] 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.

[14] Yang Chen, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Liang Peng, Olivier Temam, and Chengyong Wu. Evaluating iterative optimization across 1000 datasets. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 448–459, New York, NY, USA, 2010. ACM.

[15] Michele Tartara and Stefano Crespi Reghizzi. Continuous learning of compiler heuristics. *ACM Trans. Archit. Code Optim.*, 9(4):46:1–46:25, January 2013.

[16] Davide Gadioli, Gianluca Palermo, and Cristina Silvano. Application autotuning to support runtime adaptivity in multicore architectures. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*, pages 173–180. IEEE, 2015.

[17] Grigori Fursin, Anton Lokhmotov, and Ed Plowman. Collective Knowledge: towards R&D sustainability. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'16)*, pages 864–869, March 2016.

[18] A. Cohen and E. Rohou. Processor virtualization and split compilation for heterogeneous multicore embedded systems. In *Design Automation Conference*, pages 102–107, June 2010.

[19] Tomofumi Yuki. Understanding PolyBench/C 3.2 kernels. In *International workshop on Polyhedral Compilation Techniques (IMPACT)*, pages 1–5, 2014.

[20] Cristina Silvano, Giovanni Agosta, Stefano Cherubin, Davide Gadioli, Gianluca Palermo, Andrea Bartolini, Luca Benini, Jan Martinovič, Martin Palkovič, Kateřina Slaninová, 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, pages 288–293, New York, NY, USA, 2016. ACM.

[21] Andrea R Beccari, Carlo Cavazzoni, Claudia Beato, and Gabriele Costantino. Ligen: a high performance workflow for chemistry driven de novo design, 2013.

[22] P. Fritzson, P. Aronsson, A. Pop, H. Lundvall, K. Nystrom, L. Saldamli, D. Broman, and A. Sandholm. 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*, pages 1588–1595, Oct 2006.

[23] Francesco Casella. Simulation of large-scale models in modelica: State of the art and future perspectives. In *LINKÖPING ELECTRONIC CONFERENCE PROCEEDINGS*, pages 459–468, 2015.