Perfrewrite – Program Complexity Analysis via Source Code Instrumentation

Michael Kruse

Perfrewrite
Program Complexity Analysis via Source Code Instrumentation

Michael Kruse*,[1],

* *INRIA Saclay Île-de-France, Parc Orsay Université, 4 rue Jacques Monod, 91893 Orsay cedex, France*

ABSTRACT

Most program profiling methods output the execution time of one specific program execution, but not its computational complexity class in terms of the big-O notation. Perfrewrite is a tool based on LLVM's Clang compiler to rewrite a program such that it tracks semantic information while the program executes and uses it to guess memory usage, communication and computational complexity. While source code instrumentation is a standard technique for profiling, using it for deriving formulas is an uncommon approach.

KEYWORDS: profiling; computational complexity; source code instrumentation; LLVM Clang

1 Motivation

Several methods are common in order to deduce the performance characteristics of a program, usually used to identify a program's bottleneck that is most worthy to optimize. We are specially interested in HPC applications used by physicists. Our case study are the programs tmLQCD [JU09] and DD-HMC [Lüs05], both implementations of the lattice quantum chromodynamics (LQCD) simulation. The goal is to automatically derive the execution time in terms of the size of the input field.

The usual approach for program complexity analysis is done statically on the source code with tools such as *PIPS* [KAC+96]. Unfortunately, challenges like pointer arithmetic make it near impossible for such tools to process the given programs. Another idea is to use modelling program such as *PAMELA* [vG93], but it requires to rewrite a program in its domain specific language and still does not support language constructs such as pointers.

Profiling with tools like *gprof*, *oprofile*, *Tau* and many more is the most mature technique, but do not return the complexity in big-O notation. This drawback can be coped with by running the program multiple times with different inputs and let a statistics tool (like *GNU R*) analyse it. To get a meaningful result also large input sizes have to be sampled. In case of shared cluster systems probably someone will complain about eating computation time just to find out how long a program takes to execute.

[1]E-mail: michael.kruse@inria.fr

2 Approach

The main idea is to replace all relevant types in a C program by custom C++ classes. For instance, the `double` type gets replaced with a custom `Double`[2] class. Then, we rely on C++ operator overloading to call our method whenever an arithmetic operation is to be executed. In addition to just simply execute the operation, it increases a global *FLOP* counter. Similarly, `malloc` and `free` can be substituted to track memory usage as well as calls to MPI to track communication between cluster nodes.

By itself this does nothing more than standard profiling does. In addition, also loops are annotated using preprocessor macros. If the number of loop iterations is dependent on the program's input size, say n, it will multiply the counters in the loop body by n and divide by the number of loop iterations it actually executes in its configuration. In order to make this work, integral types are also replaced by custom C++ classes such that they carry annotations of their values' origin. The associated semantic information are the actual integer value, the value expressed as a term (a symbolic formula of the input size), and a typical integer value of a typical large input size.

For example, a program has two inputs N and M of size n and m. We execute the program using a configuration $n = 8$ and $m = 4$, a very small problem size. The symbolic representations are $n_{\text{term}} =$ "n" and $m_{\text{term}} =$ "m" respectively. When the values are used in an arithmetic operation, for instance they are multiplied, the annotated value becomes $nm_{\text{term}} =$ "$n \cdot m$". The annotated large input size might be $n_{\text{large}} = 256$ and $m_{\text{large}} = 128$. The annotation after the multiplication is $nm_{\text{large}} = 32768$. It is used whenever two values are compared since the formula representation has no total order, assuming that the large problem case is more relevant. One use is the update of the peak memory usage during the program's execution, avoiding a gigantic step function. In short, we execute the program for a small problem size while interpolating its characteristics for a large problem size.

As most profiling tools do, we log the begin and return of every function. This will allow to create a call tree with stats for every function.

```
void execute(int n) {
  double *field =
    malloc(n * sizeof(*field));
  double localSum = 0;
  for (int i = 0; i < n; ++i)
    localSum += field[i];
  double globalSum;
  if (n > 128)
   MPI_Allreduce(&localSum, &globalSum,
     1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  free(field);
}
```

```
void execute(Num n) {ENTERFUNCTION
  DynamicMem<Double> field =
    perf_malloc<Double>(n * sizeof(*field));
  Double localSum = 0;
  LOOP(n) for (int i = 0; i < n; ++i) ITERATION
    localSum += field[i];
  Double globalSum;
  if (n > 128)
    MPI_Allreduce(&localSum, &globalSum,
      1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  free(field);
EXITFUNCTION}
```

Figure 1: An example program before (left) and after instrumentation (right)

Of course this approach is not perfect. It works only with polynomial complexities. Loops must have the structure of for-loops with iteration count directly depending on the input. Data-dependent while-loops for instance are not possible, but can be worked-around by

[2]Notice the change of capitalization

making the number of while-iterations an input size. But the advantage is that the techniques does not need to cope with logic unrelated to the complexity. No need to handle pointer arithmetic, polymorphism, the halting problem, etc. because the program is executed and these intractabilities are evaluated naturally.

The implemented approach executes every loop body at most twice. The first iteration may initialize global fields[3], the second iteration is assumed to represent all following iterations. The is sufficient for the two case study programs mentioned in the previous section. The program's result will be wrong, but we are not interested in it anyway.

3 Implementation

A simple textual insertion and replacement is not sufficient. We want to track memory blocks, therefore we also need to replace pointers to them. Pointers are typed, meaning the pointer replacement class needs to be a template whose variable type declaration syntax is different than a pointer's. To instrument manually is a tedious task to be avoided if possible. In addition, instrumenting only necessary parts of the code will safe us from incompatibilities in unaffected and working program sections. E. g. C++ classes cannot perfectly emulate the behaviour of C pointers. If necessary the user might need to adapt the code manually, like when a loop syntax does not exactly match any implemented patters. This is why the instrumented code must stay readable.

The Clang compiler [Cla] has the necessary facilities for a semantic analysis of C and C++ code. It also retains the source code locations in the internal representation so our tool can locate and replace the parts that have to be replaced. Clang already includes an Objective-C to C++ translator that works similarly. Also, a full C++ compiler already includes facilities to get the return type of an expression involving custom types.

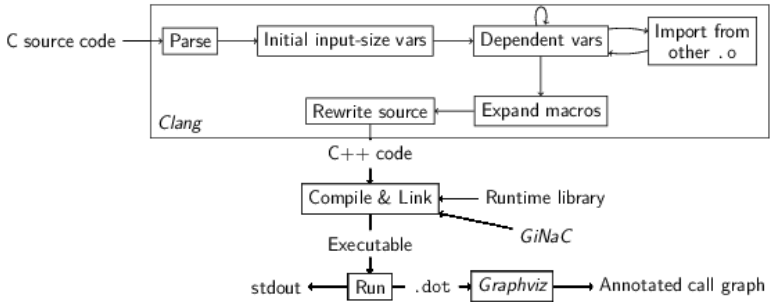

Figure 2: Processing pipeline.

The principal tool working is shown in Figure 2. Intrinsic types are transformed into custom C++ types whenever it is assigned an expression that evaluates to a custom type or the types are incompatible in some context (e.g. when passing by reference). Types may influence themselves, so this is done until a fixpoint is reached. When types types declared `extern` are changed – this includes non-static functions – this change must be propagated to the other source files of the program. If the change is part of a macro, the macro has to be expanded before the instrumentation. Clang does not retain the complete information of how a macro was expanded so the complete parse must be repeated after macro expansion.

[3]of the kind `if (!g_field) g_field = malloc(...)`

The instrumented code can now be compiled using any C++ compiler. The code changes require a custom runtime to be linked against it containing the implementation of the replacement classes. The implementation of the symbolic representation is from GiNaC [BFK+], a computer algebra system using C++ as control language.

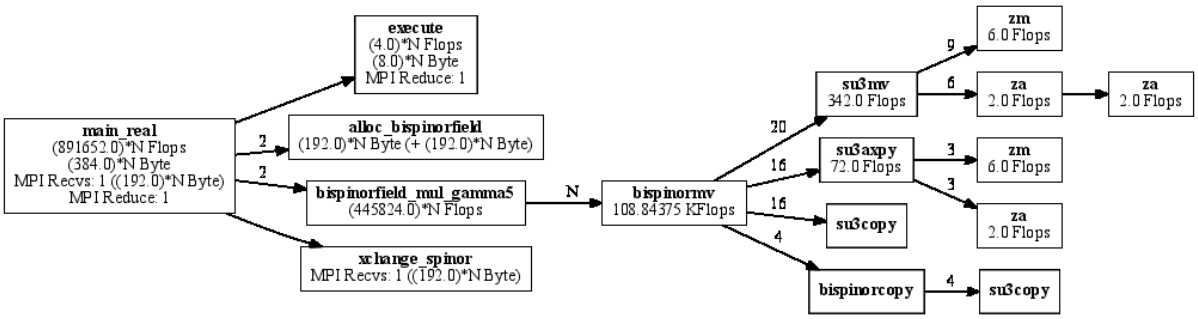

Figure 3: Graphviz rendering of an example program.

When the compilee runs, it writes a call tree into a `.dot` file that can be processed by Graphviz [AT]. Figure 3 shows such a call tree. For every function call it shows how often it has been called, the number of floating point operations, peak memory usage and MPI calls, all as complexities depending on the input parameters.

References

[AT] AT & T Labs Research. Graphviz. `http://graphviz.org`.

[BFK+] Christian Bauer, Alexander Frink, Alexander V. Kisil, Christian Kreckel, Alexei Sheplyakov, and Jens Vollinga. GiNaC is Not a CAS. `http://www.ginac.de`.

[Cla] Clang: A C Language Family Frontend for LLVM. `http://clang.llvm.org`.

[JU09] Karl Jansen and Carsten Urbach. tmLQCD: A Program Suite to Simulate Wilson Twisted Mass Lattice QCD. *Comput. Phys. Commun.*, 180:2717–2738, 2009.

[KAC+96] Ronan Keryell, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, Francois Irigoin, and Pierre Jouvelot. PIPS: a Workbench for Building Interprocedural Parallelizers, Comilers and Optimizers. Technical Paper A/289/CRI, ENS des Mines Paris, May 1996. `http://pips4u.org`.

[Lüs05] Martin Lüscher. Schwarz-Preconditioned HMC Algorithm for Two-Flavour Lattice QCD. *Comput. Phys. Commun.*, 165:199–220, 2005. `http://luscher.web.cern.ch/luscher/DD-HMC/index.html`.

[vG93] Arjan J. C. van Gemund. Performance Prediction of Parallel Processing Systems: The PAMELA Methodology. In *Proc. 7th ACM Int. Conf. on Supercomputing*, ICS '93, pages 318–327, New York, USA, 1993. ACM.