“Perfrewrite”

Memory and Time Complexity Analysis via Source Code Instrumentation

Michael Kruse
INRIA Saclay Grand Large/LRI Alchemy
Overview

- Static: Analysis of the source code
- Profiling: Observation of executed code
- Empirical: Regression of many executions
Automatic Performance Analysis Methods

Overview

- Static: Analysis of the source code
- Profiling: Observation of executed code
- Empirical: Regression of many executions
- “Perfrewrite”: Mix of existing techniques
  - Source code *instrumentation* as used for profiling
  - *Source analysis* to know what to rewrite
  - Instrumented code does *abstract interpretation*
Static Analysis of the Source Code

- **PIPS**, polyhedral model tools, call graph

- **Exact**, no uncertainty or noise

- **Machine-independent**, output is characteristic to the program only
  - Source code must meet some strict rules
    - Loop bounds must be known (while loops, break, goto, ...)
    - Pointers are evil (function pointers, pointer chasing, ...)
    - Analysis time can grow quickly with if conditions within loops
  - Therefore this analysis is usually only done on procedure-level

- Dynamic conditions not evaluable statically

- **Machine-independent**, how long will I have to wait when I run it?
Profiling

- **gprof, Scalasca, Tau, Polyspace, DynaProf, Intel VTune, Valgrind, IPM, . . .**
- **Lots of tools available**
- **Source code-independent**, at least some tools
- **Results are very concrete**
  - *Noise* (OS scheduler, hardware interrupts, . . .)
  - *Machine-specific* results, does not always represent program characteristics
  - *Fixed input*, no complexity
Empirical Regression Analysis

- **time, GNU R, Matlab**

  - **Source code-independent**
  
  - Results based on real machine executions
    
    - **Noise** (OS scheduler, hardware interrupts,...)
    
    - **Uncertainty**, obtained complexity/regression model is a guess
      
      - Linear growth? Quadratic? Cubic? Exponential?
      
      - And why?
    
    - Waste of compute time
      
      - Energy consumption
      
      - Angry coworkers whose’ quota is used up
Perfrewrite

Goals

*Wanted information*: Runtime, memory usage, MPI communication depending on input size

- Ability to process pre-existing programs
- Application domain is scientific computing
  - C code usually written by physicist
  - Huge, multidimensional arrays (fields)
  - Field content does not directly influence runtime, only its size
- Rewrite as few a possible
  - Source code must stay readable in case manual intervention is required
- Run on desktop computers, not supercomputers
- Be ready for C++ programs
Source Code Instrumentation

- Source program

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

```c
void execute(int n) {
    ENTER_FUNCTION
    double* field =
        (double*)malloc(n * sizeof(*field));
    double localSum = 0;
    for (int i = 0; i < n; ++i)
        localSum += field[i];
    double globalSum;
    MPI_Allreduce(&localSum, &globalSum, 1,
        MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    free(field);
    EXIT_FUNCTION
}
```
Source Code Instrumentation

Overload floating-point operations

```c
void execute(int n) {ENTER_FUNCTION
    Double* field =
        (Double*)malloc(n * sizeof(*field));
    Double localSum = 0;
    for (int i = 0; i < n; ++i)
        localSum += field[i];
    Double globalSum;
    MPI_Allreduce(&localSum, &globalSum, 1,
        MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    free(field);
EXIT_FUNCTION}
```
Source Code Instrumentation

- Substitute size-dependent integers with symbolic representation

```c
void execute(Num n) {
    ENTER_FUNCTION
    Double* field = (Double*)malloc(n * sizeof(*field));
    Double localSum = 0;
    for (int i = 0; i < n; ++i)
        localSum += field[i];
    Double globalSum;
    MPI_Allreduce(&localSum, &globalSum, 1,
                  MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    free(field);
    EXIT_FUNCTION}
```
Number of loop iterations is input-dependent

```c
void execute(Num n) {ENTER_FUNCTION
    Double* field =
        (Double*)malloc(n * sizeof(*field));
    Double localSum = 0;
    for (int i = 0; i < n; ++i) PERF_LOOP((n) - (0))
        localSum += field[i];
    Double globalSum;
    MPI_Allreduce(&localSum, &globalSum, 1,
        MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    free(field);
EXIT_FUNCTION}
```
Memory allocation are also input-size-dependent

```c
void execute(Num n) {ENTER_FUNCTION
    Double* field =
        perf_malloc<Double>(n * sizeof(*field));
    Double localSum = 0;
    for (int i = 0; i < n; ++i) PERF_LOOP((n) - (0))
        localSum += field[i];
    Double globalSum;
    MPI_Allreduce(&localSum, &globalSum, 1,
        MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    free(field);
EXIT_FUNCTION}
```
**Source Code Instrumentation**

- Use a fake pointer to simulate allocations of symbolic size

```c
void execute(Num n) {
    DynamicMem<Double> field =
        perf_malloc<Double>(n * sizeof(*field));
    Double localSum = 0;
    for (int i = 0; i < n; ++i) PERF_LOOP((n) - (0))
        localSum += field[i];
    Double globalSum;
    MPI_Allreduce(&localSum, &globalSum, 1,
        MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    free(field);
}
```
Source Code Instrumentation

- Overload library functions to accept symbolic representations

```c
void execute(Num n) {ENTER_FUNCTION
    DynamicMem<Double> field =
        perf_malloc<Double>(n * sizeof(*field));
    Double localSum = 0;
    for (int i = 0; i < n; ++i) PERF_LOOP((n) - (0))
        localSum += field[i];
    Double globalSum;
    MPI_Allreduce(&localSum, &globalSum, 1,
        MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    free(field);
EXIT_FUNCTION}
```
Source Code Instrumentation

Program Output

C:\Users\Meinersbur\src\perftest\Debug\perftest_perf.exe

### Start perf of C:\Users\Meinersbur\src\perftest\Debug\perftest_perf.exe

### Perf done in 7.877 seconds
retval: 0
Total flops: <891652.0>*N Flops
Heap leak: + 0 Byte
Peak heap: <384.0>*N Byte
Receive operations: 1
Received bytes: 192*N
Send operations: 0
Sent bytes: 0
Reductions: 1
Reduced values: 1
Source Code Instrumentation

Result Call Graph
3rd Party Software

- **Clang**  C/C++/Objective C *frontend*
  - Parser
  - Semantic analysis
  - Token search
  - Code fragment generation
  - Text rewriter

- **GiNaC**  Computer-Algebra-System library
  - Formula representation of input-size dependent value

- **Any C++ compiler (gcc, clang+llvm, msvc, . . .)**
  - For compiling the rewritten program

- **Graphviz**
  - Draw the call graph
Workflow

C source code

Parse  \rightarrow  Initial input-size vars  \rightarrow  Dependent vars  \rightarrow  Import from other .o

Rewrite source  \leftarrow  Expand macros

Clang

C++ code

Compile & Link  \rightarrow  Runtime library

Executable

stdout  \leftarrow  Run .dot  \rightarrow  Graphviz  \rightarrow  Call graph

GiNaC
Encountered Difficulties

- Declarations hidden in preprocessor macros
Encountered Difficulties

- Declarations hidden in preprocessor macros
- Multiple translation units
Encountered Difficulties

- Declarations hidden in preprocessor macros
- Multiple translation units
- Ambiguities
Encountered Difficulties

- Declarations hidden in preprocessor macros
- Multiple translation units
- Ambiguities
- Clang
  - Batteries included
  - But only half-charged
Encountered Difficulties

- Declarations hidden in preprocessor macros
- Multiple translation units
- Ambiguities
- Clang
  - Modifying an AST not really supported
  - No distinction between Concrete Syntax Tree and Abstract Syntax Tree
  - Lot of bit juggling to save memory
  - Importing AST nodes only for C language and buggy
  - Does not remember all tokens of a declaration (static, struct, ...)
  - Semantic analyser not designed as a library
    - All public methods
    - Methods require objects only available during parsing (Scope)
Future Outlook

- Track memory accesses
  - Log order of access for memory layout transformation
  - Assume L1 cache for innermost loop, L2 . . .
Future Outlook

- Track memory accesses
  - Log order of access for memory layout transformation
  - Assume L1 cache for innermost loop, L2 . . .

- Real abstract interpretation
  - Execute every loop iteration just once
  - Assign probability to if-condition
Future Outlook

- Track memory accesses
  - Log order of access for memory layout transformation
  - Assume L1 cache for innermost loop, L2 . . .
- Real abstract interpretation
  - Execute every loop iteration just once
  - Assign probability to if-condition
- Predict execution time assuming a constant delay of an operation

Use better prediction engines for kernels
On-the-fly benchmarking
GPU predictor
Future Outlook

- Track memory accesses
  - Log order of access for memory layout transformation
  - Assume L1 cache for innermost loop, L2 . . .

- Real abstract interpretation
  - Execute every loop iteration just once
  - Assign probability to if-condition

- Predict execution time assuming a constant delay of an operation

- Use better prediction engines for kernels
  - On-the-fly benchmarking
  - GPU predictor
That's all Folks!
History

Memory Usage Regression
History

Computer-Algebra-System

QdaggerQ_povy := 12*(VOLUMEPLUSRAND+1)*spinorsize
13824 \cdot L^2 \cdot T + 2304 \cdot L^3 \cdot T + 27648 \cdot L \cdot T + 27648 \cdot L^2 + 4608 \cdot L^3 + 2304

memory := init_gauge_field + init_geometry_indices + init_monomials + init_spinor_field + init_r
16 \cdot T + 4 \cdot (L + 4)^2 \cdot (T + 4) + 4 \cdot (L + 4)^3 \cdot (T + 4) + 112296 \cdot L^2 \cdot T + 16580 \cdot L^3 \cdot T + 4 \cdot (L + 4) \cdot (T + 4) + 298224 \cdot L \cdot T + 29822

simplify(memory)
912 \cdot L + 352 \cdot T + 112348 \cdot L^2 \cdot T + 16584 \cdot L^3 \cdot T + 298452 \cdot L \cdot T + 298432 \cdot L^2 + 37448 \cdot L^3 + 8756
```javascript
var VOLUME = new LatticePoly(0, 0, 0, 0, 0, 0, 0, 1);
var RAND = new LatticePoly(0, 0, 0, 0, 0, 0, 0, 6, 0);
var EDGES = new LatticePoly(0, 0, 0, 12, 12, 0, 0, 0);
var VOLUMEPLUSRAND = VOLUME.add(RAND).add(EDGES);
var _cb2rand = RAND.add(EDGES.mult(2));

var tqlqcd_mpi_init = new LatticePoly();
if (mpidim >= 3) {
    tqlqcd_mpi_init = tqlqcd_mpi_init.add(T.mult(LX).mult(LY).mult(0.5).mult(spinor_size));
    tqlqcd_mpi_init = tqlqcd_mpi_init.add(T.mult(LX).mult(LY).mult(0.5).mult(spinor_size));
    tqlqcd_mpi_init = tqlqcd_mpi_init.add(T.mult(LX).mult(LY).mult(0.5).mult(spinor_size));
    tqlqcd_mpi_init = tqlqcd_mpi_init.add(T.mult(LX).mult(LY).mult(0.5).mult(spinor_size));
    tqlqcd_mpi_init = tqlqcd_mpi_init.add(T.mult(LX).mult(LY).mult(0.5).mult(spinor_size));
    tqlqcd_mpi_init = tqlqcd_mpi_init.add(T.mult(LX).mult(LY).mult(0.5).mult(halfspinor_size));
}

V = VOLUMEPLUSRAND.add(_cb2rand);
var init_gauge_field = new LatticePoly();
init_gauge_field = init_gauge_field.add(V.times(size));
init_gauge_field = init_gauge_field.add(V.times(4).add(1).times(sus_size));
if (halfspinor) {
    init_gauge_field = init_gauge_field.add(2 * size);
    init_gauge_field = init_gauge_field.add(VOLUME.mul(size));
    init_gauge_field = init_gauge_field.add(VOLUME.mult(4).add(1).mul(sus_size));
} else {
    if (gauge_copy) {
        init_gauge_field = init_gauge_field.add(VOLUME.add(RAND).mul(size));
        init_gauge_field = init_gauge_field.add(VOLUME.add(RAND).mul(8).add(1).mul(sus_size));
    }
}

var init_geometry_indices = new LatticePoly();
init_geometry_indices = init_geometry_indices.add(V.mult(size));
init_geometry_indices = init_geometry_indices.add(V.mult(size));
init_geometry_indices = init_geometry_indices.add(V.mult(size));
init_geometry_indices = init_geometry_indices.add(V.mult(size));
init_geometry_indices = init_geometry_indices.add(T.add(4).mult(LX.add(4)).mult(size));
init_geometry_indices = init_geometry_indices.add(T.add(4).mult(LX.add(4)).mult(LY.add(4)).add(size));
init_geometry_indices = init_geometry_indices.add(T.add(4).mult(LX.add(4)).mult(LY.add(4)).mult(size));
init_geometry_indices = init_geometry_indices.add(V.mult(size));
init_geometry_indices = init_geometry_indices.add(V.mult(size));
init_geometry_indices = init_geometry_indices.add(V.mult(size));

if (mpidim >= 3) {
    init_geometry_indices = init_geometry_indices.add(T.mult(LX).mult(LY).mult(size));
}
```

---

**History**

**JavaScript Application**

History

JavaScript Application with Call Graph Generation

```javascript
    1743 }
    1744 }
    1745 return usage;
    1746 }
    1747
    1748 function read_input() {
    1749     var usage = new Usage(arguments.callee.toString());
    1750     usage.addMemory(16384); // Initial buffer size for parser
    1751     return usage;
    1752 }
    1753
    1754 function mul_r(N) {
    1755     assert(N != undefined);
    1756     var usage = new Usage(arguments.callee.toString());
    1757     usage.addFlops(N, 24);
    1758
    1759     return usage;
    1760 }
    1761
    1762 function main() {
    1763     var usage = new Usage(arguments.callee.toString());
    1764     usage.call(read_input());
    1765     usage.call(mlqcd_mpi_init());
    1766     g_dbw2rand = 0;
    1767     usage.call(init_gauge_field());
    1768     usage.call(init_geometry_indices(VOLUMEPLUSRAND));
    1769     usage.call(init_monomials());
    1770     usage.call(init_pion_field());
    1771     usage.call(geometry());
    1772     if (halfspinor) {
    1773         usage.call(init_dirac_halfspinor());
    1774     }
    1775     usage.call(read_gauge_field());
    1776     usage.call(init_blocks()); // Possibly called multiple times but without additional effects
    1777     if (computerSubspace) usage.call(generate_ofl_subspace(g_Ns, VOLUME));
    1778     if (computerInversion) usage.call(op_invert());
    1779     usage.exit();
    1780     return usage;
    1781 }
```
History

JavaScript Call Graph Output
“Translating” some application to JavaScript didn’t work out

- Call by reference, pointer passing
  - Do it in C itself!

Manual rewriting too cumbersome

- Change ⇒ Compile ⇒ Error ⇒ Change ⇒ Compile ⇒ Next error
- Also idea with colleague to combine it with GPU prediction
  - Do it automatically!
That's all Folks!