UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

“Perfrewrite”

Memory and Time Complexity Analysis via Source Code Instrumentation

Michael Kruse

INRIA Saclay Grand Large/LRI Alchemy

Automatic Performance Analysis Methods

Overview

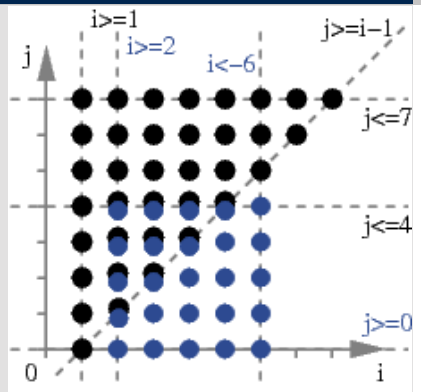
- Static: Analysis of the source code
- Profiling: Observation of executed code
- Empirical: Regression of many executions

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

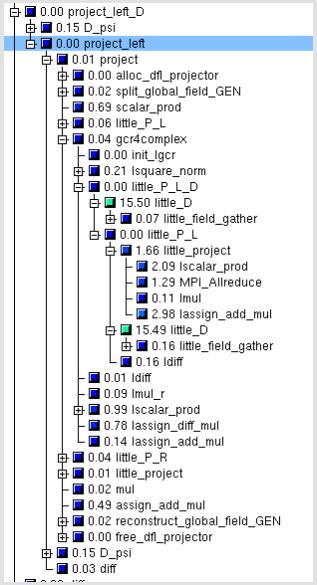


- *PIPS*, polyhedral model tools, call graph

+ *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 in usually only done on procedure-level
− Dynamic conditions not evaluatable statically
− *Machine-independent*, how long will I have to wait when I run it?

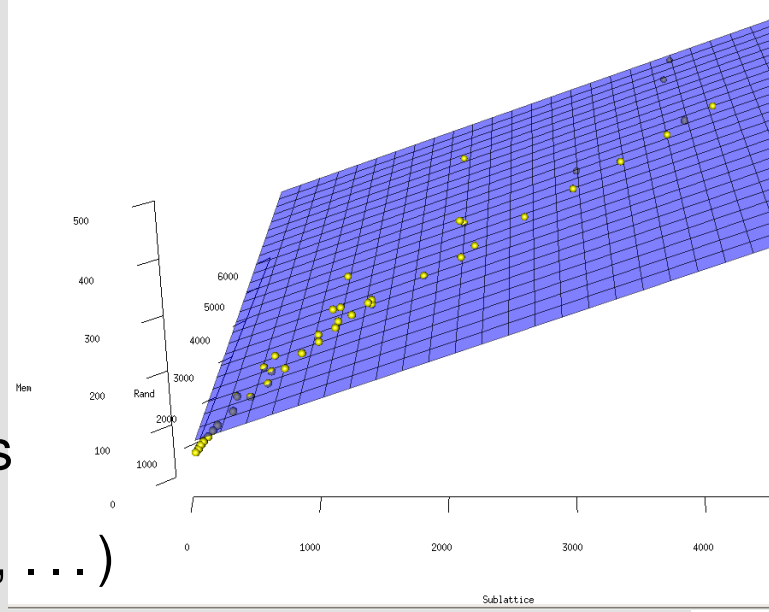
Profiling

- *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*

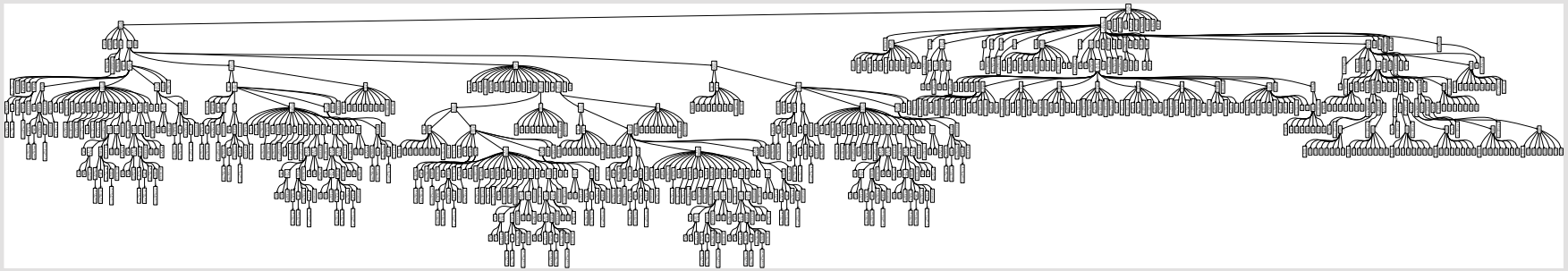
+ *Source code-independent*
+ Results based on real machine executions
− *Noise* (OS scheduler, hardware interrupts, . . .)
− *Uncertainty*, obtained complexity/regression model is a guess
  - Linear growth? Quadratic? Cubic? Exponential?
  - And why?
− Waste of compute time
  - Energy consumption
 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

Goal



Source Code Instrumentation

- Source program

```
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);
}
```

Source Code Instrumentation

- Call graph instrumentation

```
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

```
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

```
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}
```

Source Code Instrumentation

- Number of loop iterations is input-dependent

```
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}
```

Source Code Instrumentation

- Memory allocation are also input-size-dependent

```
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

```
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

- Overload library functions to accept symbolic representations

```
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

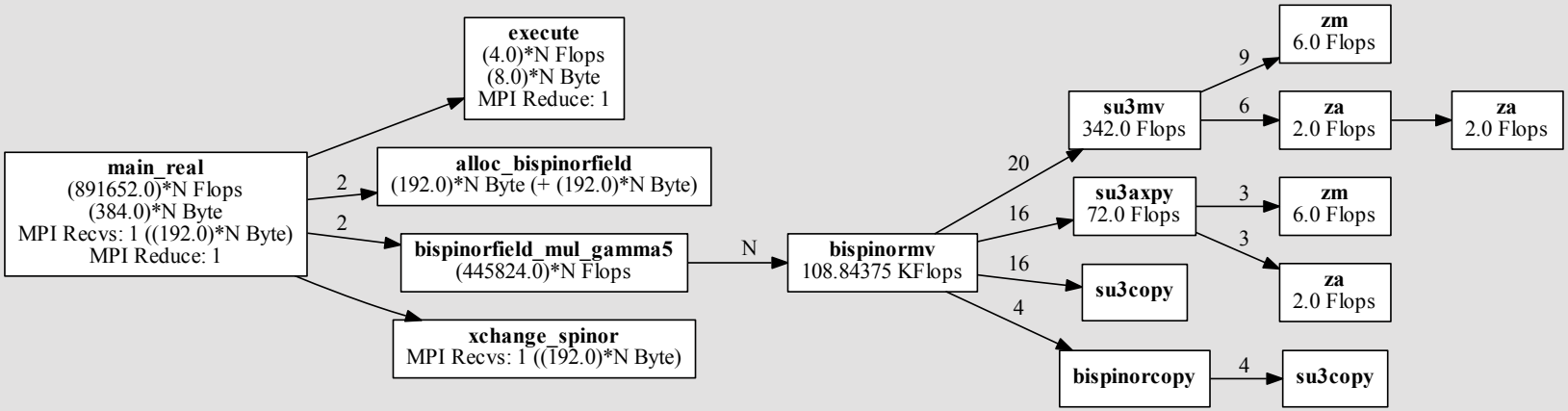
```
C:\Users\Meinersbur\src\perftest\Debug\perftest_perf.exe
### Start perf of C:\Users\Meinersbur\src\perftest\Debug\perftest_perf.exe



### 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

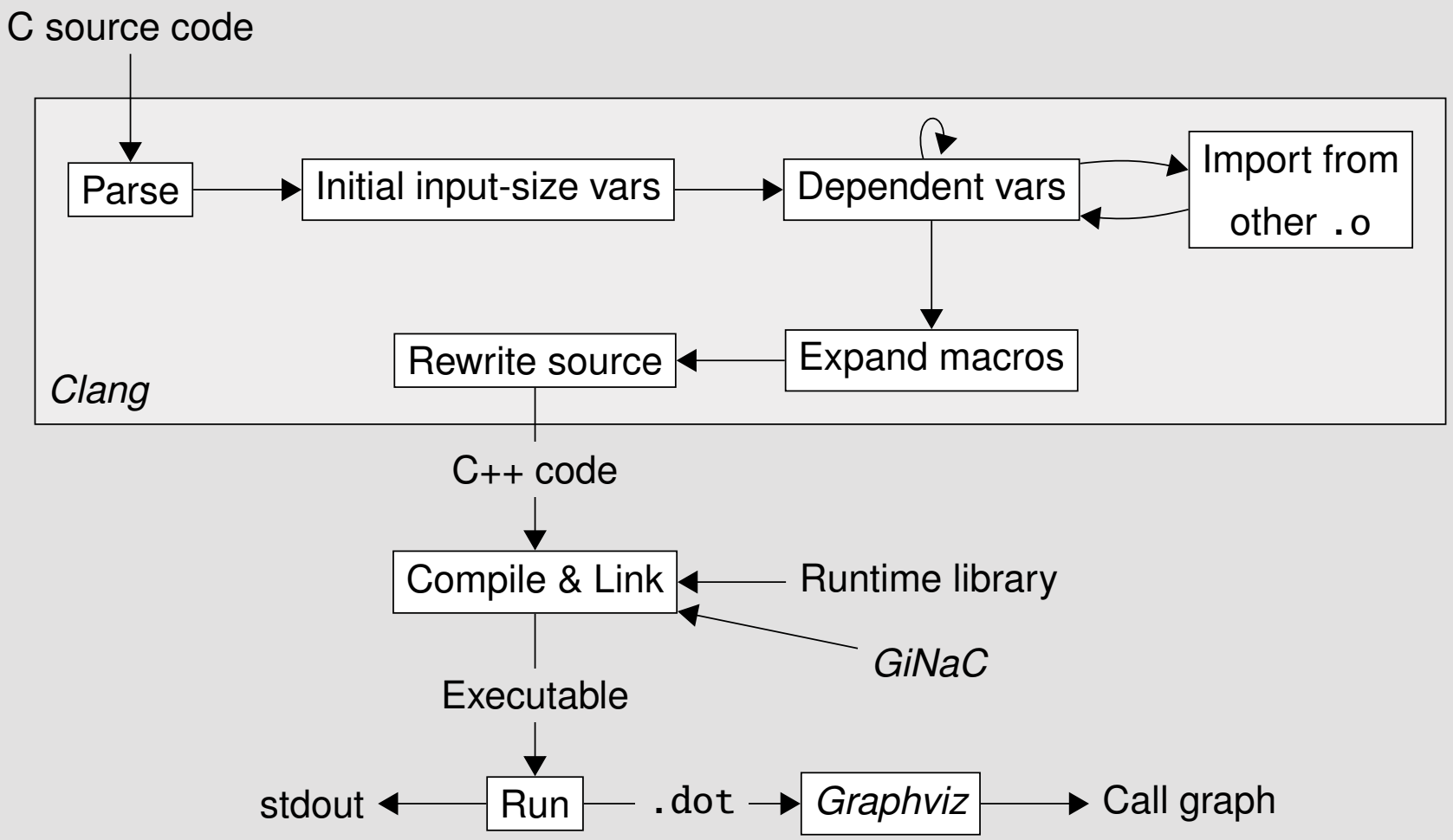
Result Call Graph



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
Initial input-size vars
Dependent vars
Import from other .o
Expand macros
Rewrite source
Clang
C++ code
Compile & Link
Runtime library
GiNaC
Executable
Run
stdout
.dot
Graphviz
Call graph

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

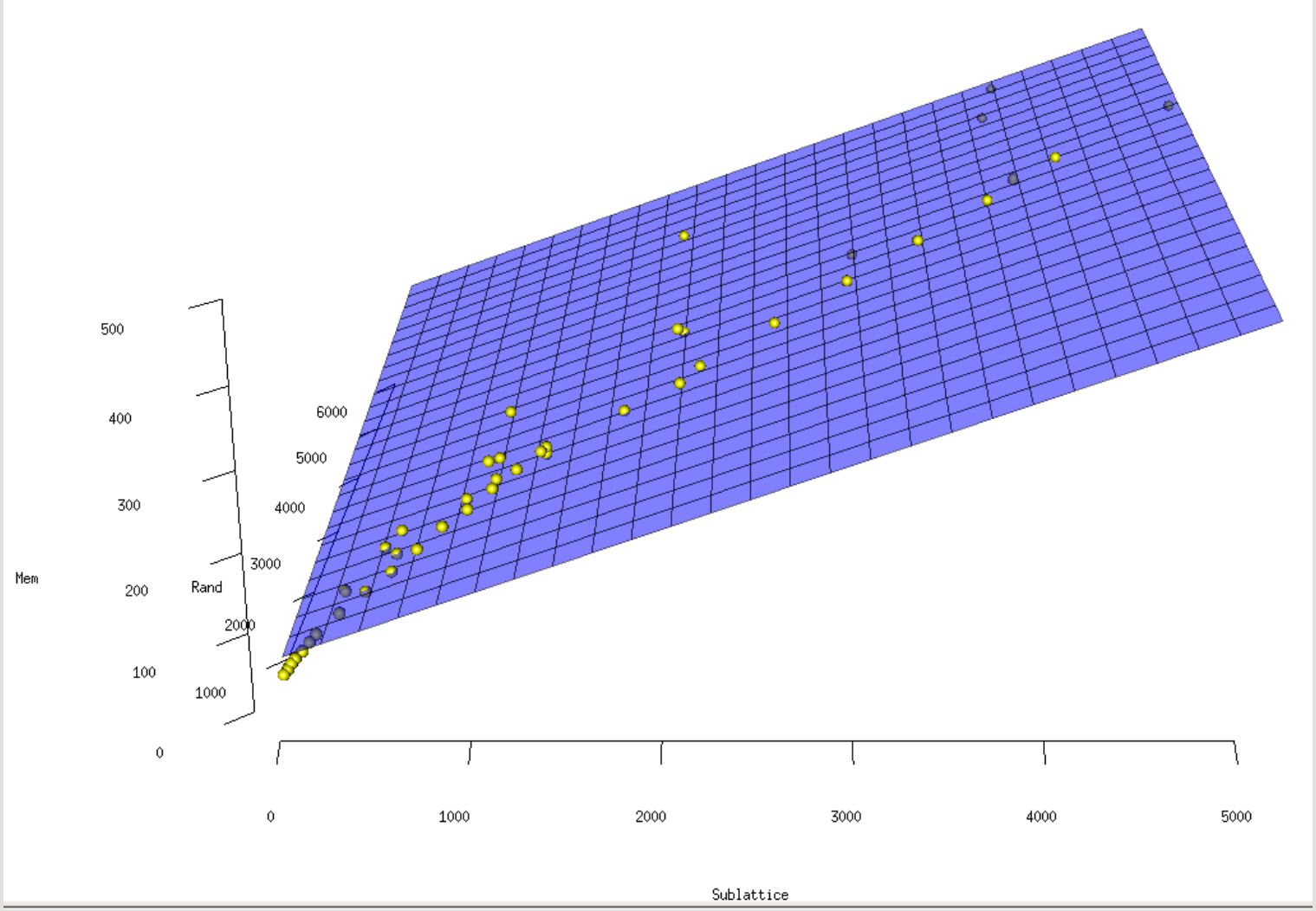
Future Outlook

- 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

MuPAD Pro 3.1 - [phmc.mnb]

Datei Bearbeiten Ansicht Einfügen Format Notebook Debug Extras Fenster Hilfe

Courier New 11 Westlich

```
• QdaggerQ_poly := 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)
```

$$912 \cdot L + 352 \cdot T + 112348 \cdot L^2 \cdot T + 16584 \cdot L^3 \cdot T + 298452 \cdot L \cdot T + 298432 \cdot L^2 + 37448 \cdot L^3 + 8756$$

Bereit | Keine Verbindung | Eing, 1te | Ze 117 : 17

History

JavaScript Application

```
402                 var VOLUME = new LatticePoly(0, 0, 0, 0, 0, 0, 0, 1);
403                 var RAND = new LatticePoly(0, 0, 0, 0, 0, 2, 6, 0);
404                 var EDGES = new LatticePoly(0, 0, 0, 12, 12, 0, 0, 0);
405                 var VOLUMEPLUSRAND = VOLUME.add(RAND).add(EDGES);
406                 var g_dbw2rand = RAND.add(EDGES.mult(2));
407
408                 var tmlqcd_mpi_init = new LatticePoly();
409                 if (mpidim >= 3) {
410                     tmlqcd_mpi_init = tmlqcd_mpi_init.add(T.mult(LX).mult(LY).mult(0.5).mult(spinorsize));
411                     tmlqcd_mpi_init = tmlqcd_mpi_init.add(T.mult(LX).mult(LY).mult(0.5).mult(spinorsize));
412                     tmlqcd_mpi_init = tmlqcd_mpi_init.add(T.mult(LX).mult(LY).mult(0.5).mult(spinorsize));
413                     tmlqcd_mpi_init = tmlqcd_mpi_init.add(T.mult(LX).mult(LY).mult(0.5).mult(spinorsize));
414                     tmlqcd_mpi_init = tmlqcd_mpi_init.add(T.mult(LX).mult(LY).mult(0.5).mult(halfspinorsize));
415                     tmlqcd_mpi_init = tmlqcd_mpi_init.add(T.mult(LX).mult(LY).mult(0.5).mult(halfspinorsize));
416                 }
417
418                 V = VOLUMEPLUSRAND.add(g_dbw2rand);
419                 var init_gauge_field = new LatticePoly();
420                 init_gauge_field = init_gauge_field.add(V.times(psize));
421                 init_gauge_field = init_gauge_field.add(V.times(4).add(1).times(su3size));
422                 if (halfspinor) {
423                     init_gauge_field = init_gauge_field.add(2 * psize);
424                     init_gauge_field = init_gauge_field.add(VOLUME.mult(psize));
425                     init_gauge_field = init_gauge_field.add(VOLUME.mult(4).add(1).mult(su3size));
426                 } else {
427                     if (gaugecopy) {
428                         init_gauge_field = init_gauge_field.add(VOLUME.add(RAND).mult(psize));
429                         init_gauge_field = init_gauge_field.add(VOLUME.add(RAND).mult(8).add(1).mult(su3size));
430                     }
431                 }
432
433                 var init_geometry_indices = new LatticePoly();
434                 init_geometry_indices = init_geometry_indices.add(V.mult(psize));
435                 init_geometry_indices = init_geometry_indices.add(V.mult(psize));
436                 init_geometry_indices = init_geometry_indices.add(V.mult(4).mult(intsize));
437                 init_geometry_indices = init_geometry_indices.add(V.mult(4).mult(intsize));
438                 init_geometry_indices = init_geometry_indices.add(T.add(4).mult(psize));
439                 init_geometry_indices = init_geometry_indices.add(T.add(4).mult(LX.add(4)).mult(psize));
440                 init_geometry_indices = init_geometry_indices.add(T.add(4).mult(LX.add(4)).mult(LY.add(4)).mult(psize));
441                 init_geometry_indices = init_geometry_indices.add(T.add(4).mult(LX.add(4)).mult(LY.add(4)).mult(LZ.add(4)).mult(intsize));
442                 init_geometry_indices = init_geometry_indices.add(V.mult(intsize));
443                 init_geometry_indices = init_geometry_indices.add(V.add(2).mult(intsize));
444                 init_geometry_indices = init_geometry_indices.add(V.mult(intsize));
445                 if (mpidim >= 3) {
446                     init_geometry_indices = init_geometry_indices.add(T.mult(LX).mult(LY).mult(intsize));
```

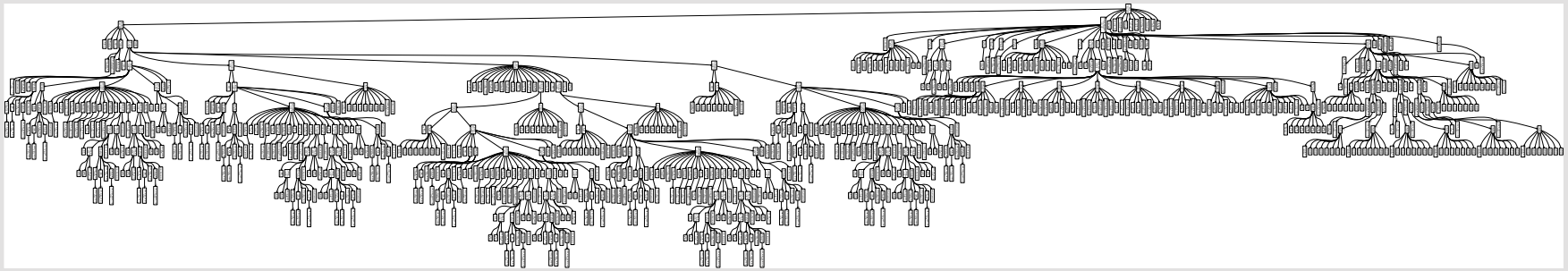
History

JavaScript Application with Call Graph Generation

```
1743        }
1744
1745        return usage;
1746    }
1747
1748    function read_input() {
1749        var usage = new Usage(arguments.callee.toString());
1750
1751        usage.addMemory(16386); // initial buffer size for parser
1752
1753        return usage;
1754    }
1755
1756    function mul_r(N) {
1757        assert(N != undefined);
1758        var usage = new Usage(arguments.callee.toString());
1759
1760        usage.addFlops(N, 24);
1761
1762        return usage;
1763    }
1764
1765    function main() {
1766        var usage = new Usage(arguments.callee.toString());
1767
1768        usage.call(read_input());
1769        usage.call(tmlqcd_mpi_init());
1770        g_dbw2rand = 0;
1771        usage.call(init_gauge_field());
1772        usage.call(init_geometry_indices(VOLUMEPLUSRAND));
1773        usage.call(init_monomials());
1774        usage.call(init_spinor_field());
1775        usage.call(geometry());
1776        if (halfspinor) {
1777            usage.call(init_dirac_halfspinor());
1778        }
1779        usage.call(read_gauge_field());
1780        usage.call(init_blocks()); // Possibly called multiple times but without additional effects
1781        if (computerSubspace)
1782            usage.call(generate_dfl_subspace(g_N_s, VOLUME));
1783        if (computerInversion)
1784            usage.call(op_invert());
1785
1786        usage.exit();
1787        return usage;
1788    }
1789
```

History

- JavaScript Call Graph Output



History

Manual Rewriting

- "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!