Toolkit for Profiling and Call Graph Analysis for RISC Architectures Based on Execution Traces

Shurygin Anton, Dolgov Alexander, Petushkov Igor
Reporters: Shurygin Anton, Dolgov Alexander
Department of Microprocessor Technologies/ Moscow Institute of Physics and Technology
Engineering & Telecommunication Conference,
November 20 – 21, 2024

CONTENT

- PROFILING PROBLEM
- PROFILE GENERATION
- CONTROL-FLOW VISUALIZATION
- RESULTS

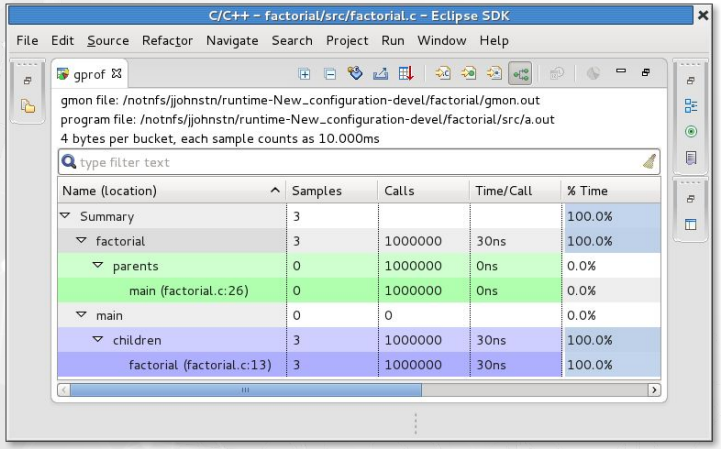
PROFILING PROBLEM

INTRODUCTION

Motivation for the relevance of profile generation based on execution traces

INTRODUCTION

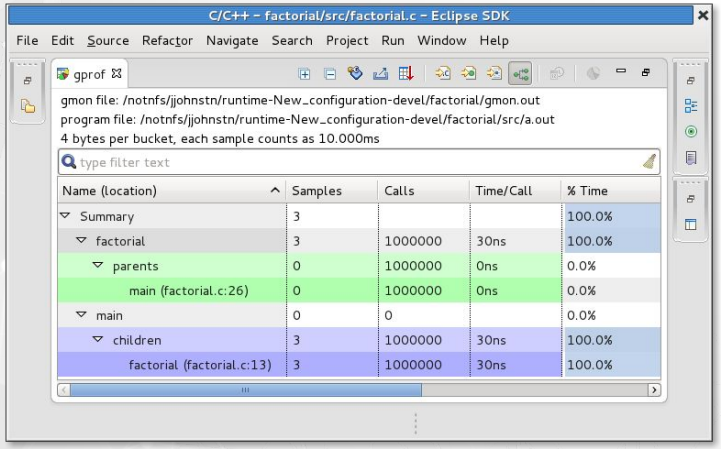
- It is important to have profiling tools to analyze the application's performance.
- There are many ready-made solutions to get an execution profile.
- Why develop something new?



Eclipse Wiki., Eclipse C/C++ SDK, GProf profile view

INTRODUCTION

- It is important to have profiling tools to analyze the application's performance.
- There are many ready-made solutions to get an execution profile.
- Why develop something new?
- The problem of generating a profile based on a binary execution trace.



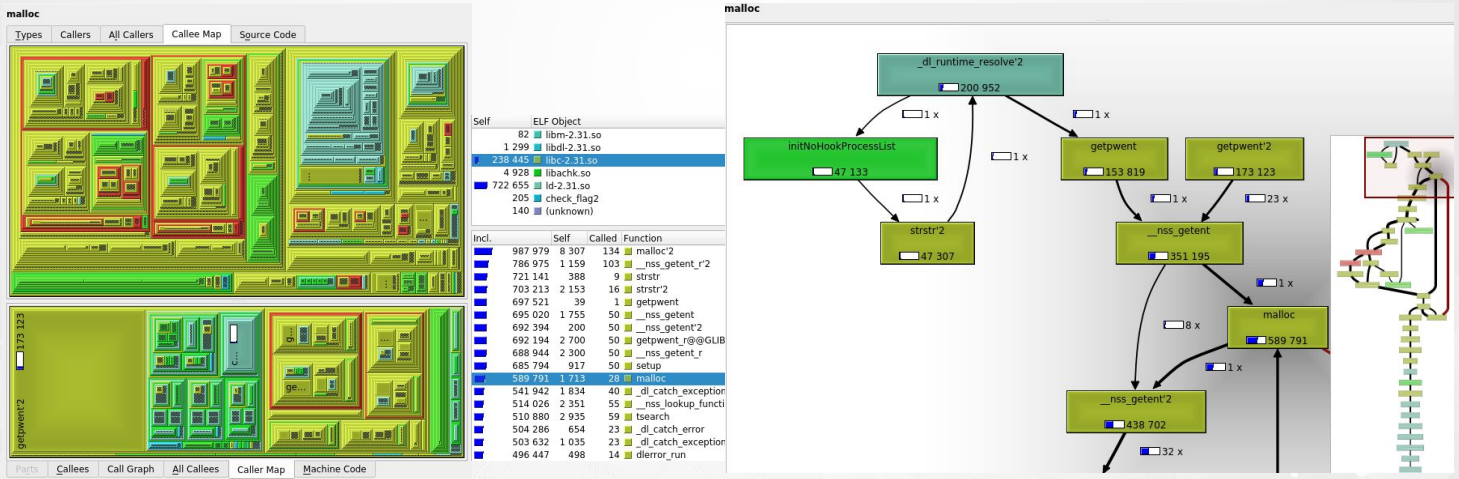
Eclipse Wiki., Eclipse C/C++ SDK, GProf profile view

INTRODUCTION

- Existing multi-platform profilers on RISC architectures provide an insufficiently accurate profile.

INTRODUCTION

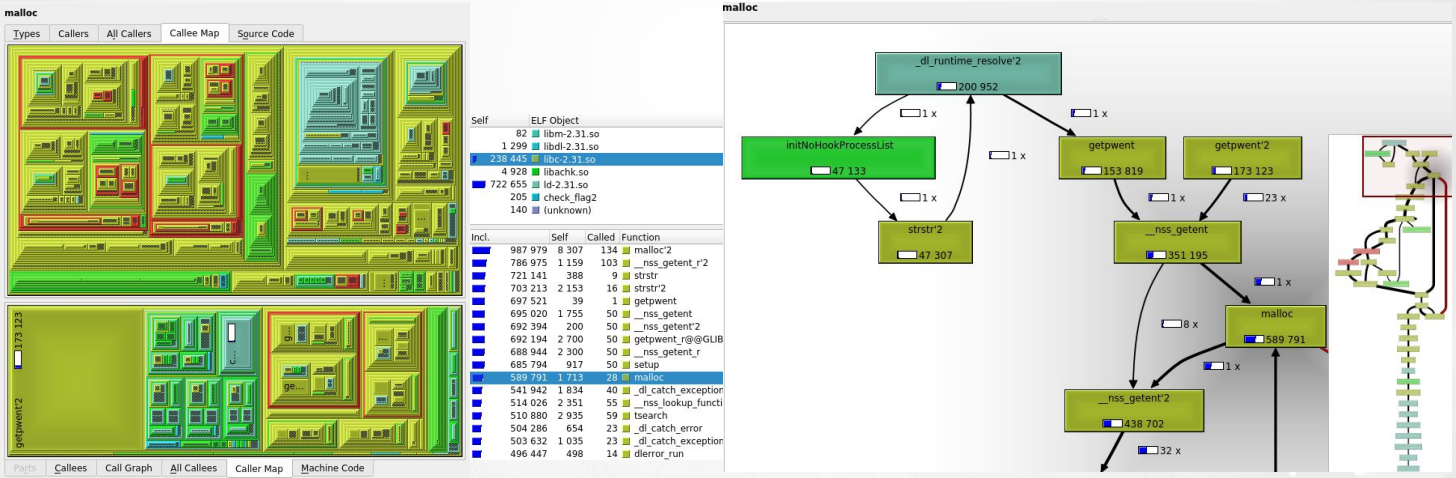
- Existing multi-platform profilers on RISC architectures provide an insufficiently accurate profile.
- Problem of visualizing binary execution trace using Kcachegrind GUI application.



Shurygin A., KCachegrind, Profile Visualization

INTRODUCTION

- Existing multi-platform profilers on RISC architectures provide an insufficiently accurate profile.
- Problem of visualizing binary execution trace using Kcachegrind GUI application.
  - ✔ Ability to visualize an exact profile of an already executed application.
  - ✔ Use for low-level program optimization



Shurygin A., KCachegrind, Profile Visualization

PROFILE GENERATION IMPLEMENTATION

Details of the implementation of the profile generation algorithm

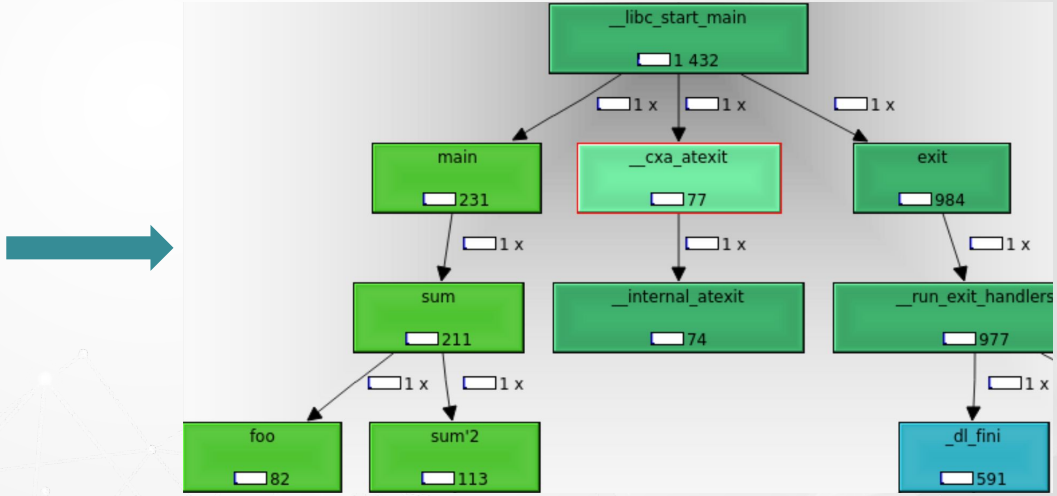
IMPLEMENTATION

- The development of the profile generation algorithm was divided into three logical parts.
  - Reverse engineering of the Callgrind output format;
  - Resolving profiling inaccuracies;
  - Implementation of profile generation algorithm and a set of auxiliary tools.

IMPLEMENTATION

- The development of the profile generation algorithm was divided into three logical parts.
  1. Reverse engineering of the output format of the reference tool for reusing Kcachegrind GUI application according to the Callgrind grammar.

```
ob=/lib/aarch64-linux-gnu/libc-2.27.so
fn=__libc_start_main
0 54
cfn=__cxa_atexit
calls=1 0
0 77
cob=path_to_elf
cfn=__libc_csu_init
calls=1 0
0 52
cfn=_setjmp
calls=1 0
0 34
cob=path_to_elf
cfn=main
calls=1 0
0 231
cfn=exit
calls=10
0 984
```



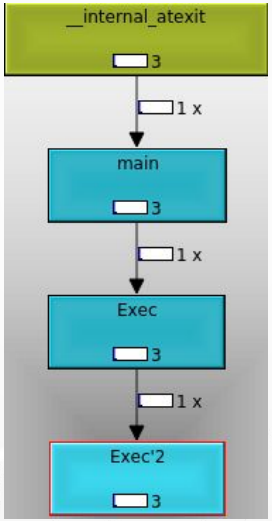
Shurygin. A, KCachegrind, Graph visualization

IMPLEMENTATION

- The development of the profile generation algorithm was divided into three logical parts
  2. Resolving inaccuracies in ARM's handling of control-flow change instructions.

```
void foo1() { int x = 1; return;
}
void Exec(int flag) {
    if (!flag) {
        foo1();
        foo1();
        foo1();
    } else
        foo1();
}
int main() {
    Exec(1);
    Exec(1);
    Exec(0);
}
```

Example of C program

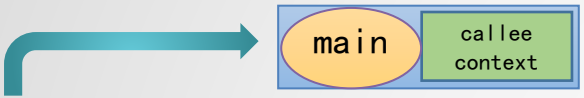


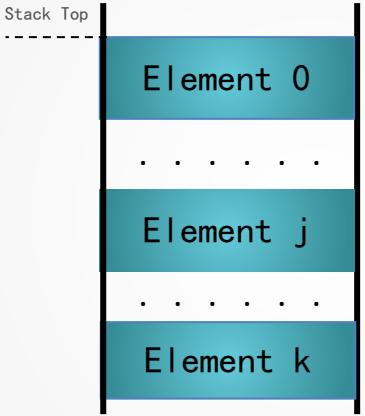
Shurygin A., KCachegrind, Strange Callgraph

```
stp     x29, x30, [sp,#-32]!
mov     x29, sp
str     w0, [sp,#28]
ldr     w0, [sp,#28]
cmp     w0, #0x0
b.ne    75c <Exec+0x28>
bl      71c <foo1>
        1 call(s) to 'foo1' (check_flag2: ch...
bl      71c <foo1>
        1 call(s) to 'foo1' (check_flag2: ch...
bl      71c <foo1>
b       760 <Exec+0x2c>
bl      71c <foo1>
        2 call(s) to 'foo1' (check_flag2: ch...
nop
ldp     x29, x30, [sp],#32
ret
stp     x29, x30, [sp,#-16]!
mov     x29, sp
```

Shurygin A., KCachegrind, Disassemble output

IMPLEMENTATION

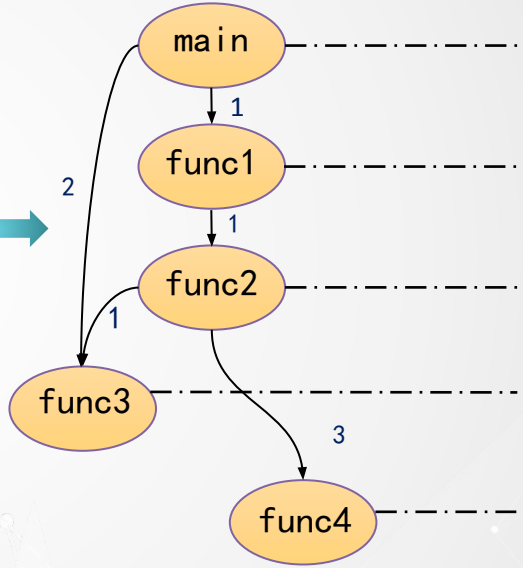




Trace execution

$[N_0 \ldots N_i \; N_{i+1} \ldots N_{n-1} N_n]$

Call Graph (E, V)



	Self cost	Inclusive cost
main	10	54
func1	7	34
func2	7	27
func3	5	5
func4	5	5

$$inclusive\ cost = self\ cost + \sum_{i=1}^{N} callee\ inclusive\ cost$$

IMPLEMENTATION

- However …

```
void foo1() { int x = 1; return;
}
void Exec(int flag) {
    if (!flag) {
        foo1();
        foo1();
        foo1();
    } else
        foo1();
}
int main() {
    Exec(1);
    Exec(1);
    Exec(0);
}
```

Example of C program

Shurygin A., KCachegrind, Callgraph

```
f…  endbr64
55  push    %rbp
4…  mov     %rsp,%rbp
4…  sub     $0x8,%rsp
8…  mov     %edi,-0x4(%rbp)
8…  cmpl    $0x0,-0x4(%rbp)
7…  jne     1170 <Exec+0x35>
                Jump 2 of 3 times to 0x1170
b…  mov     $0x0,%eax
e…  call    1129 <foo1>
                1 call(s) to 'foo1' (example)
b…  mov     $0x0,%eax
e…  call    1129 <foo1>
                1 call(s) to 'foo1' (example)
b…  mov     $0x0,%eax
e…  call    1129 <foo1>
                1 call(s) to 'foo1' (example)
e…  jmp     117a <Exec+0x3f>
                Jump 1 times to 0x117A
b…  mov     $0x0,%eax
e…  call    1129 <foo1>
                2 call(s) to 'foo1' (example)
90  nop
c9  leave
c3  ret
```

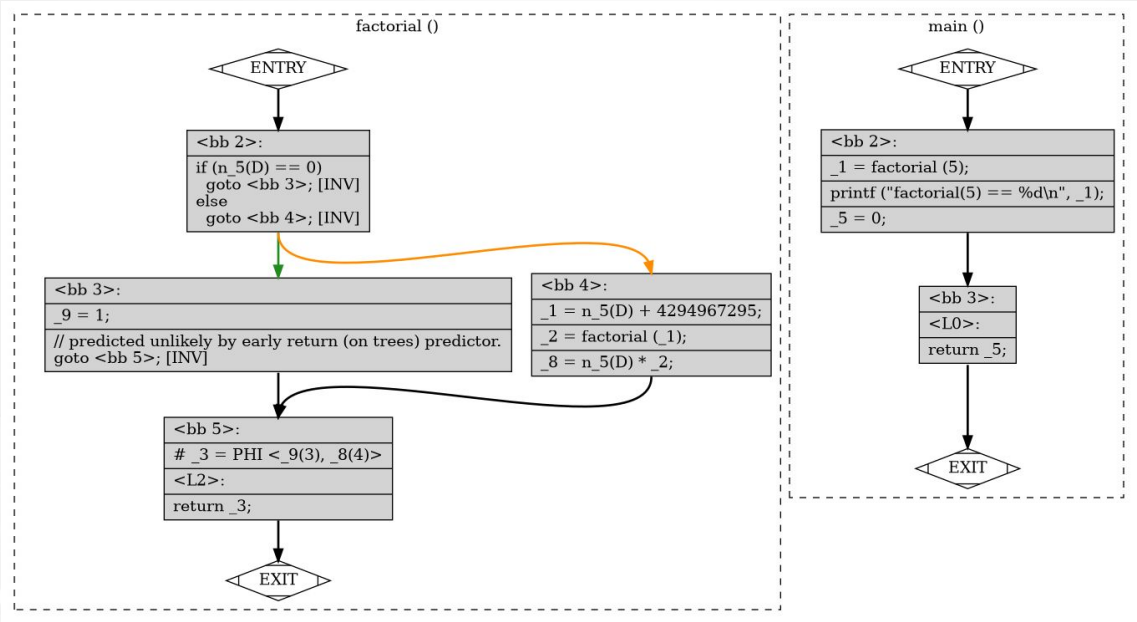
Shurygin A., KCachegrind, Disassemble output

CONTROL-FLOW VISUALIZATION IMPLEMENTATION

Details of the implementation of the control-flow graph visualization algorithm

IMPLEMENTATION

- Visualizing CFG is of great use when analyzing binary execution traces.
- Compiler is able to generate CFG of a translation unit, but it requires source files.



Dolgov A.
Control-flow graph.
Produced by gcc

IMPLEMENTATION

- The development of the control-flow graph construction and visualization algorithm was divided into 4 logical parts:
  - Reverse engineering of the Kcachegrind internals;
  - Integrating classes representing basic block and branch into Kcachegrind;
  - Developing the algorithm that divides functions into basic blocks;
  - Implementing visualization of CFG.

IMPLEMENTATION

- The development of the control-flow graph construction and visualization algorithm was divided into 4 logical parts:

1. Reverse engineering of the Kcachegrind internals.

   Two main classes that are useful for CFG construction.

```
880  /**
881   * A code instruction address of the program.
882   * Consists of a list a TracePartInstr from different trace files
883   * and a list of TraceInstrCalls if there are calls from this address.
884   */
885  class TraceInstr: public TraceListCost
886  {
887  public:
888      TraceInstr();
889      ~TraceInstr() override;
```

Dolgov. A.
Instruction class.
Kcachegrind repo

```
1207 /**
1208  * A traced function
1209  *
1210  * References to functions are stored in
1211  *  (1) a function map in TraceData (by value)
1212  *  (2) a TraceClass
1213  */
1214 class TraceFunction: public TraceCostItem
1215 {
1216 public:
1217     TraceFunction();
1218     TraceFunction(TraceData* data, const QString& name,
1219                   TraceClass* cls, TraceFile* file, TraceObject* object);
1220     ~TraceFunction() override;
```

Dolgov. A.
Function class.
Kcachegrind repo

IMPLEMENTATION

- The development of the control-flow graph construction and visualization algorithm was divided into 4 logical parts:

2. Integrating classes representing basic block and branch into Kcachegrind.

   Two new classes on which CFG is built.

```
1150  class TraceBasicBlock : public TraceListCost
1151  {
1152  public:
1153
1154      using size_type = typename std::vector<TraceInstr*>::size_type;
1155
1156      using iterator = typename std::vector<TraceInstr*>::iterator;
1157      using const_iterator = typename std::vector<TraceInstr*>::const_iterator;
1158
1159      TraceBasicBlock(typename TraceInstrMap::iterator first,
1160                      typename TraceInstrMap::iterator last);
```

Dolgov. A.
Basic block class.
Kcachegrind repo

```
1115  class TraceBranch : public TraceJumpCost
1116  {
1117  public:
1118      enum class Type { invalid, unconditional, indirect, fallThrough, true_, false_ };
1119
1120      TraceBranch(TraceInstr* from, TraceInstr* to, Type type, SubCost execCount);
1121      ~TraceBranch() override = default;
```

Dolgov. A.
Branch class.
Kcachegrind repo

IMPLEMENTATION

- The development of the control-flow graph construction and visualization algorithm was divided into 4 logical parts:

3. Developing the algorithm that divides functions into basic blocks.

Algorithm steps:

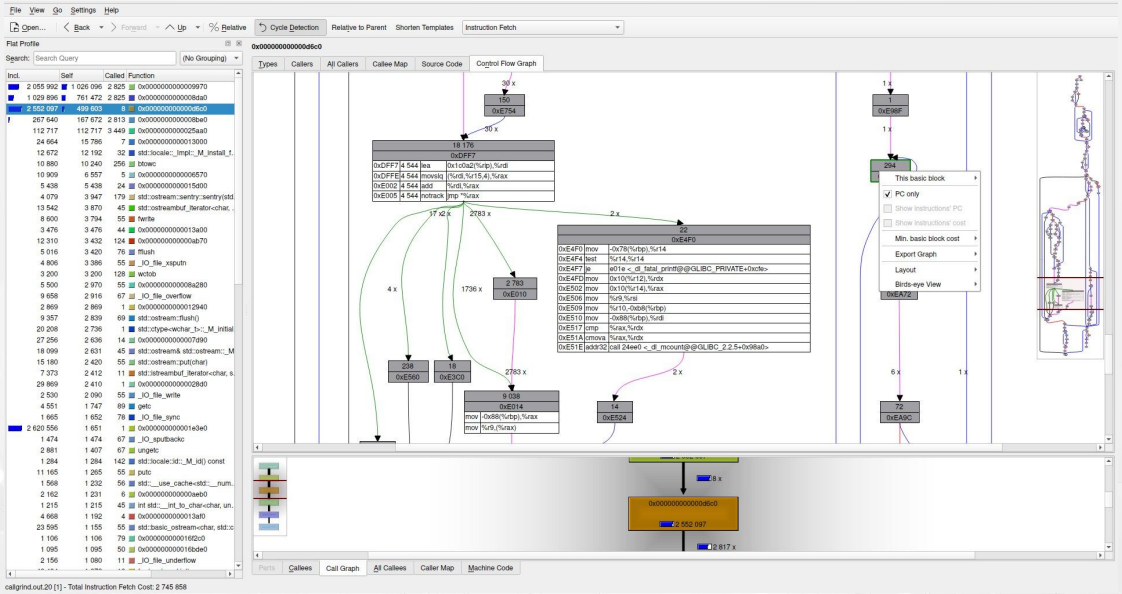
a) Collect all instructions that are destination points for jumps from the given function;
b) Based on collected information divide the function into basic blocks;
c) Handle fall-through branches (ones which source is not a terminator)

IMPLEMENTATION

- The development of the control-flow graph construction and visualization algorithm was divided into 4 logical parts:

4. Implementing visualization of CFG.

Control-flow graph view is a combination of call graph view and machine code view.



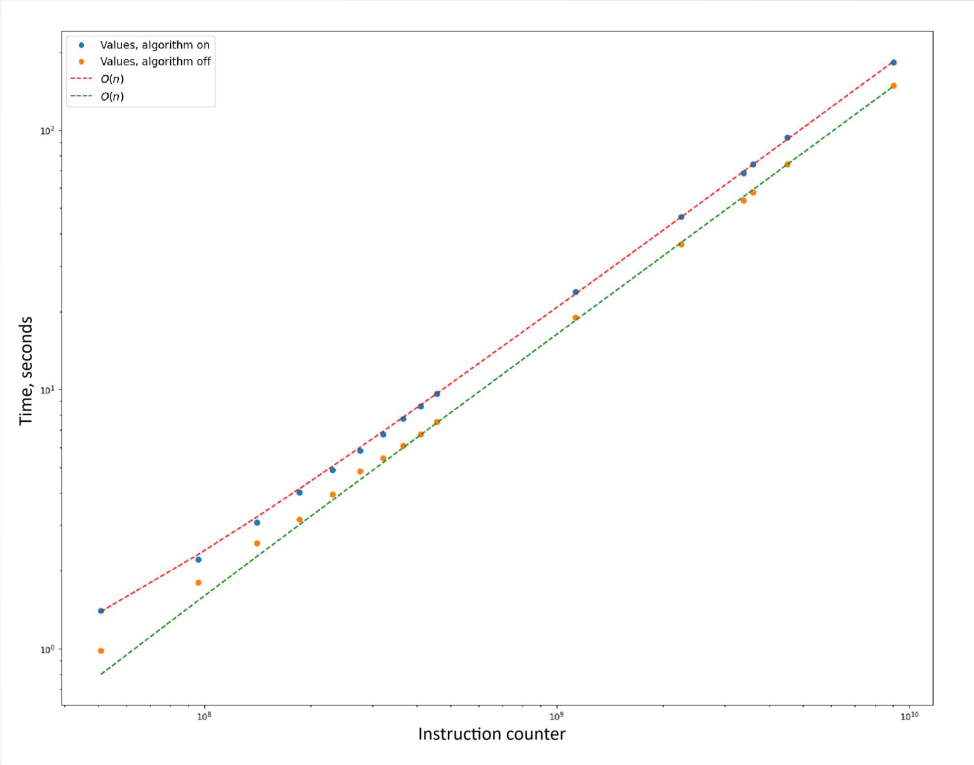
Dolgov. A.
CFG visualization.
Kcachegrind repo

RESULT SUMMARY

Evaluation of the results obtained

SUMMARY

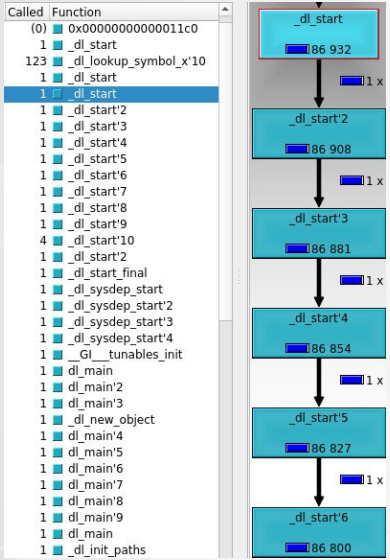
1. Developed algorithm was tested with reference profiles from Callgrind.
2. Linear dependence on the trace size.
3. SPEC CPU 2017 benchmarks run.



Shurygin A., Python (Matplotlib),
Asymptotic complexity of the algorithm

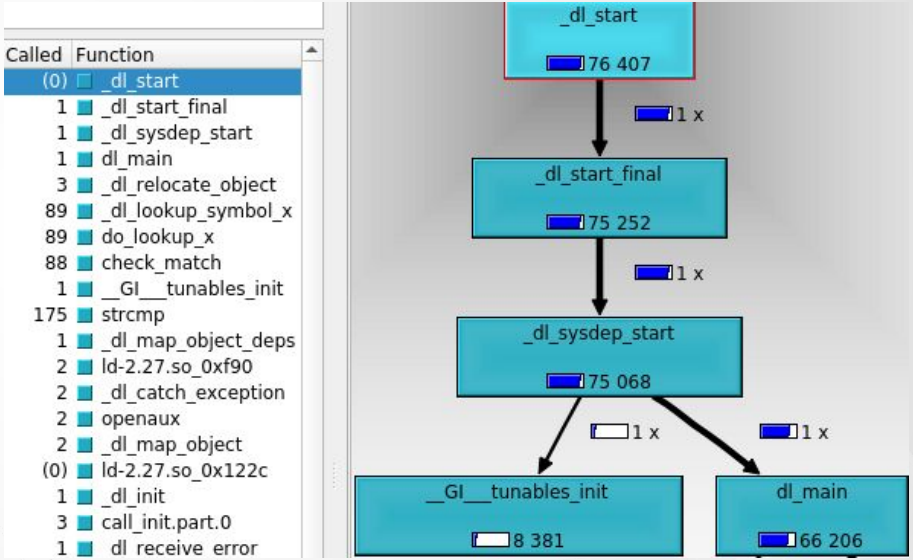
SUMMARY

- The developed algorithm resolves inaccuracies in the construction of the call & controlf-flow graphs for the ARM architecture.



Shurygin A., KCachegrind,
Callgrind profile output

VS

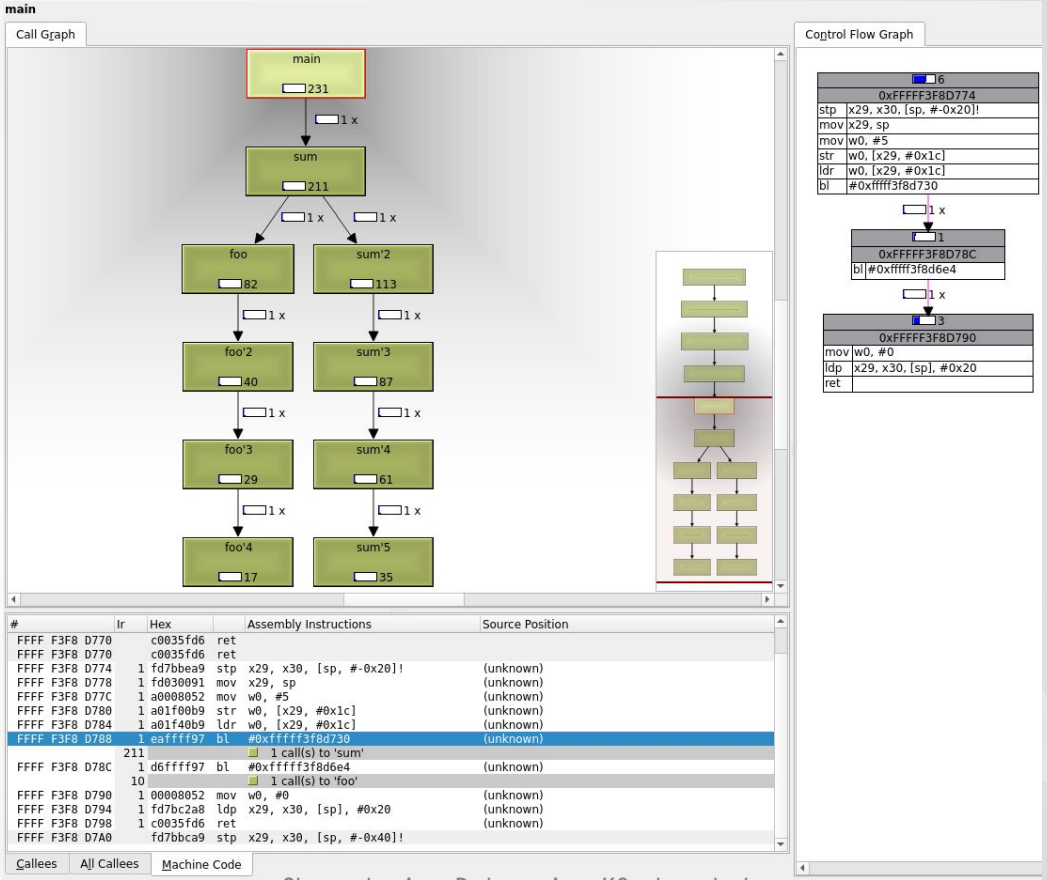


Shurygin A., KCachegrind,
Developed algorithm profile output

SUMMARY

In conclusion:

What did we do?

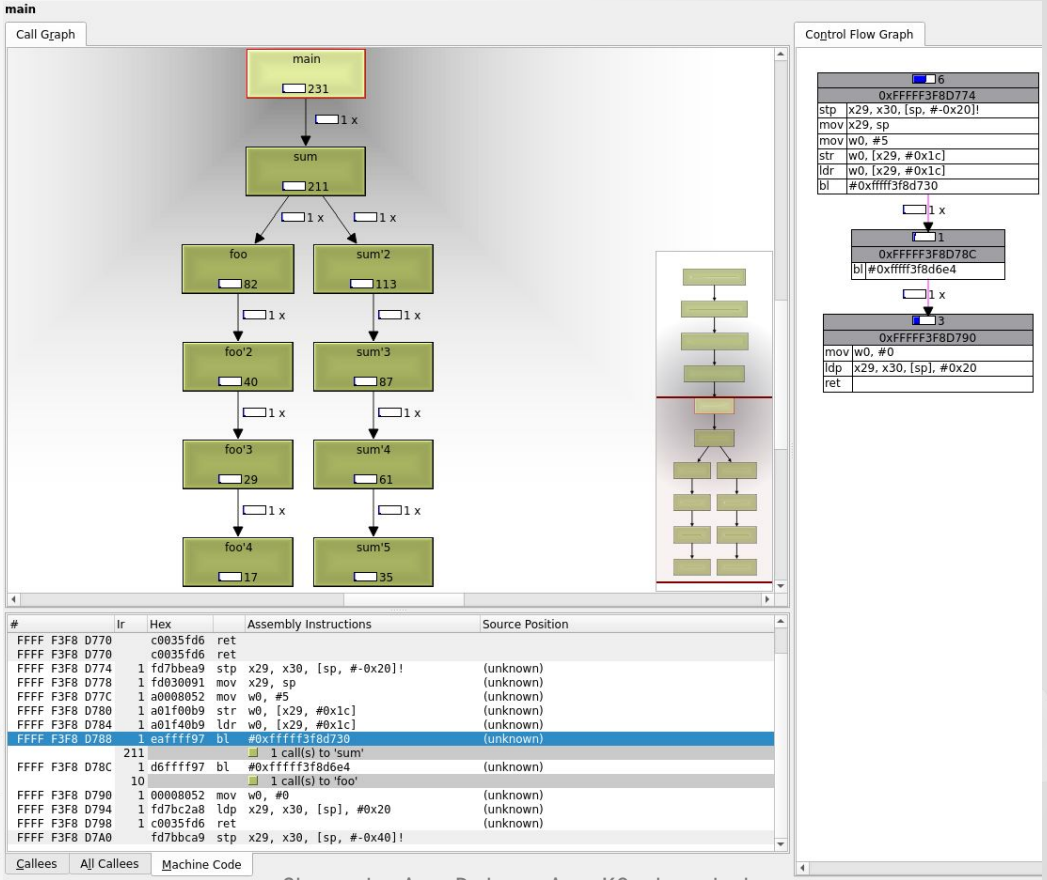


Shurygin A., Dolgov A., KCachegrind, Visualization of call graph by functions and by linear sections of code

SUMMARY

In conclusion:

- Toolkit for a profiling & analysing based on execution traces has been developed.

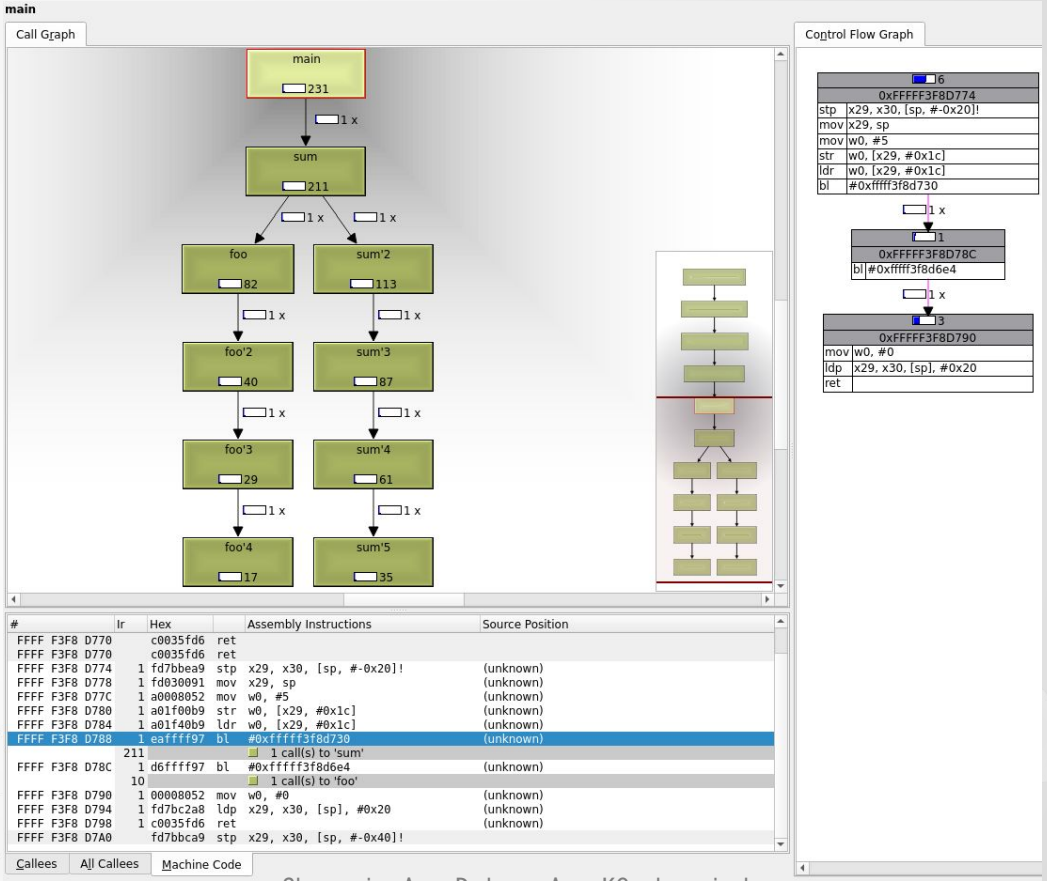


Shurygin A., Dolgov A., KCachegrind, Visualization of call graph by functions and by linear sections of code

SUMMARY

In conclusion:

Novelty of the work?

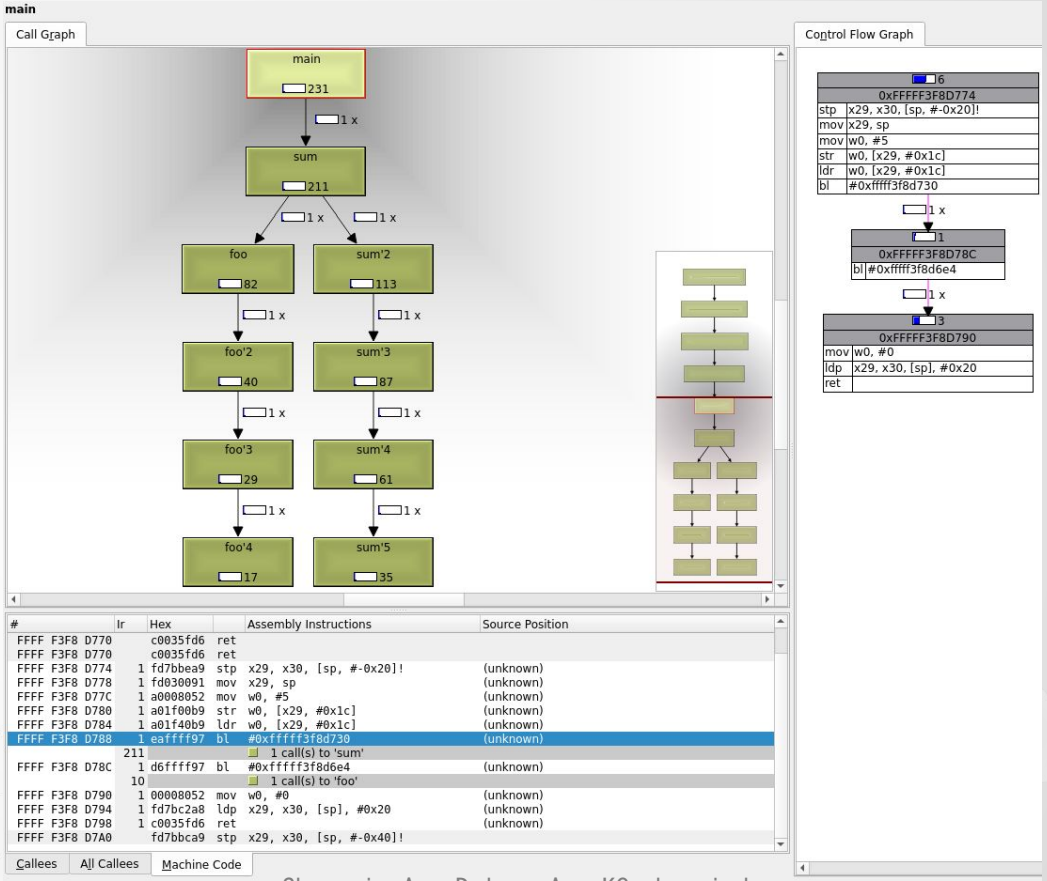


Shurygin A., Dolgov A., KCachegrind, Visualization of call graph by functions and by linear sections of code

SUMMARY

In conclusion:

- Visualize control flow for better understanding of execution.

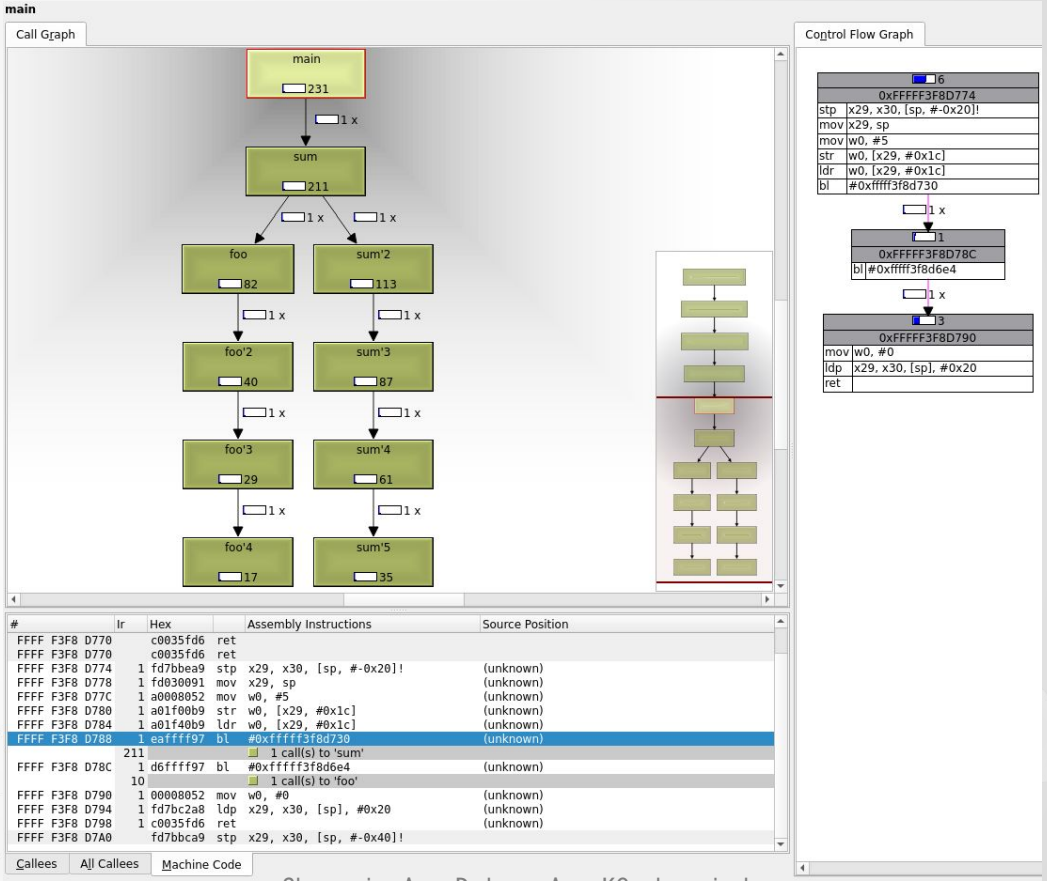


Shurygin A., Dolgov A., KCachegrind, Visualization of call graph by functions and by linear sections of code

SUMMARY

In conclusion:

For whom?

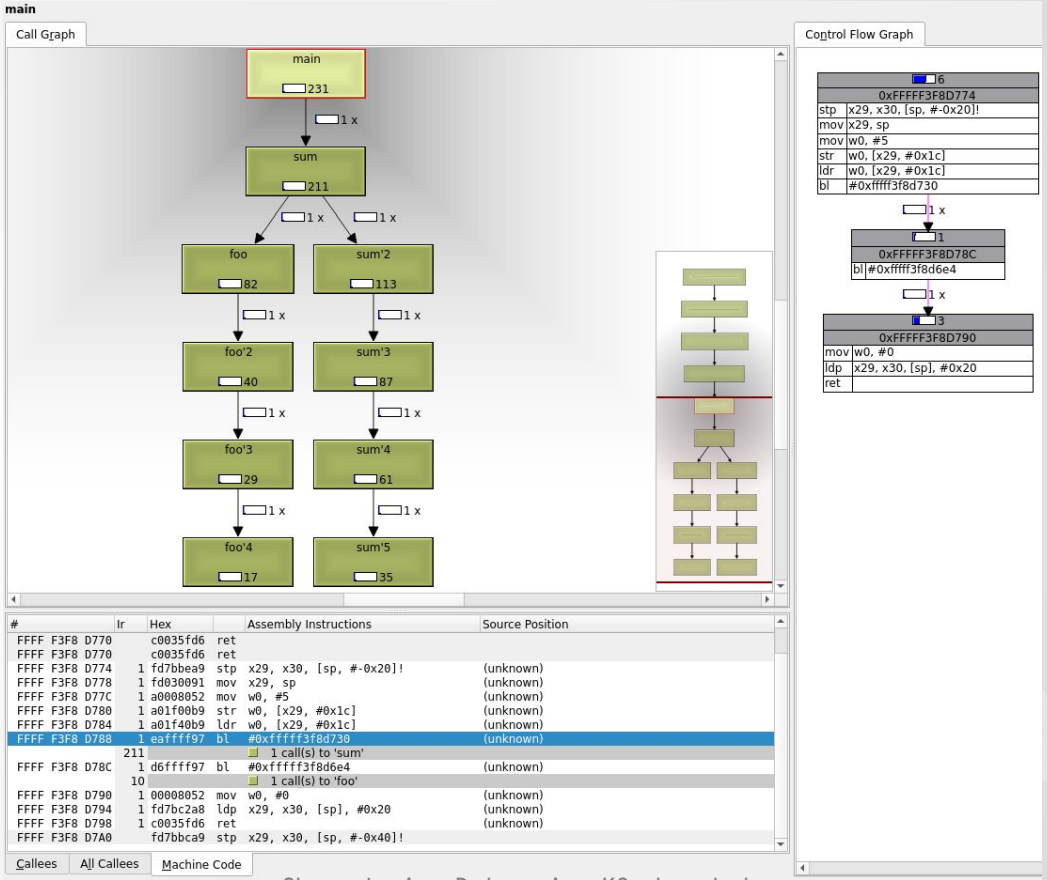


Shurygin A., Dolgov A., KCachegrind, Visualization of call graph by functions and by linear sections of code

SUMMARY

In conclusion:

- Useful for microarchitectural research and performance analysis.

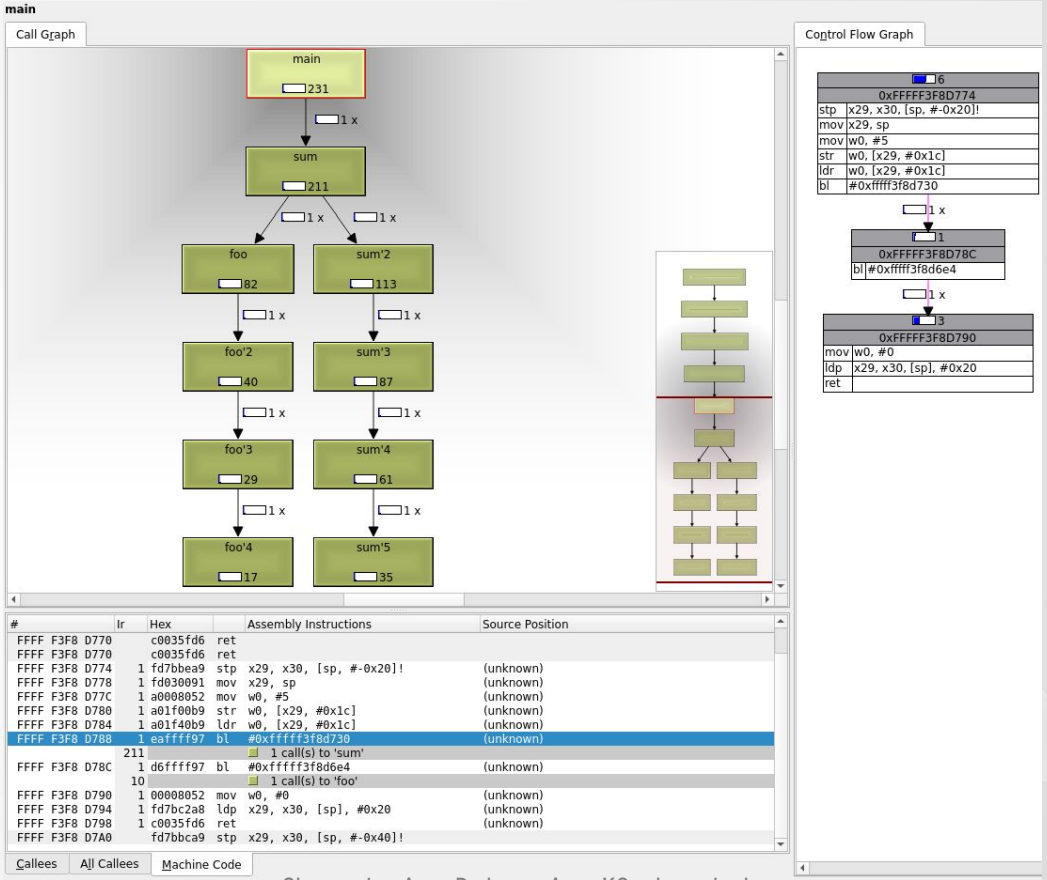


Shurygin A., Dolgov A., KCachegrind, Visualization of call graph by functions and by linear sections of code

SUMMARY

In conclusion:

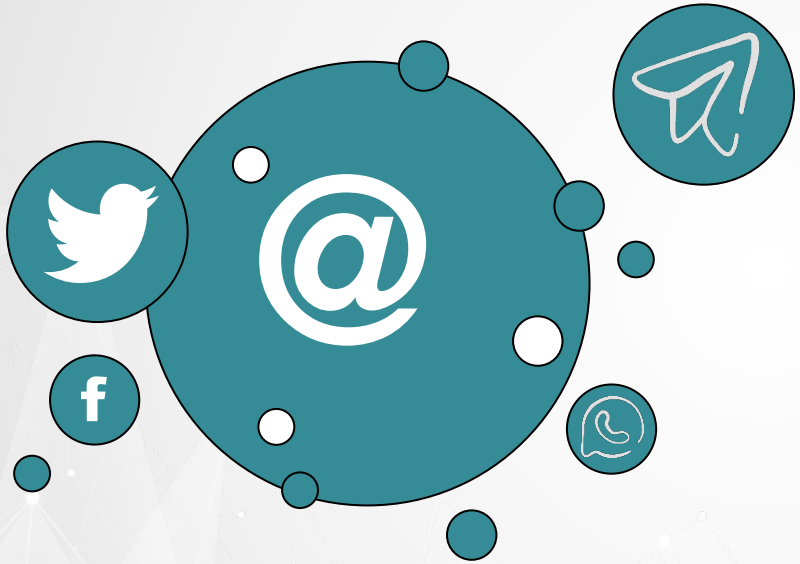
- An algorithm for generating a profile based on traces has been developed.
- Visualize control flow for better understanding of execution.
- Useful for microarchitectural research and performance analysis.



Shurygin A., Dolgov A., KCachegrind, Visualization of call graph by functions and by linear sections of code

CONTACTS

CONTACTS



We are Social!

shurygin.aa@frtk.ru
dolgov.aleksandr@phystech.edu

THANK YOU