TECHNISCHE UNIVERSITÄT DRESDEN

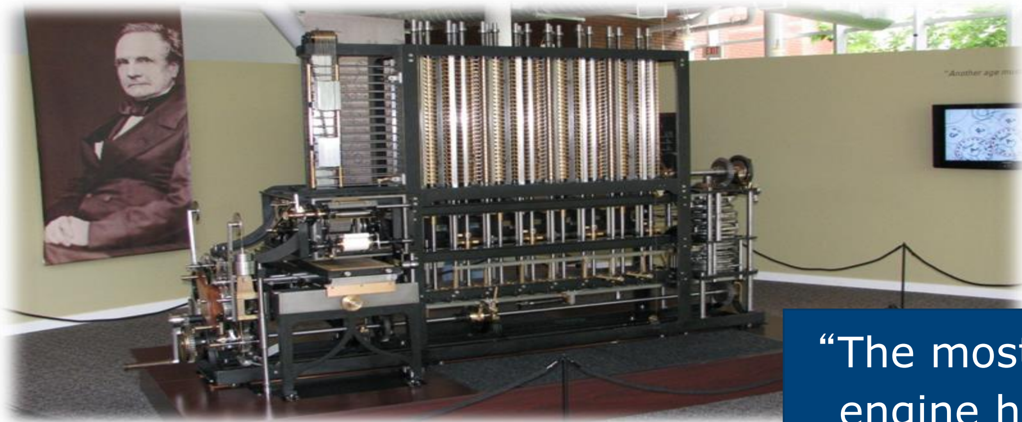
ZIH Center for Information Services & High Performance Computing

LLVM-HPC'17: Fourth Workshop on the LLVM Compiler Infrastructure in HPC

AN LLVM INSTRUMENTATION PLUG-IN FOR SCORE-P

Ronny Tschüter, Johannes Ziegenbalg, Bert Wesarg, Matthias Weber, Christian Herold, Sebastian Döbel, Ronny Brendel

Performance: an old problem



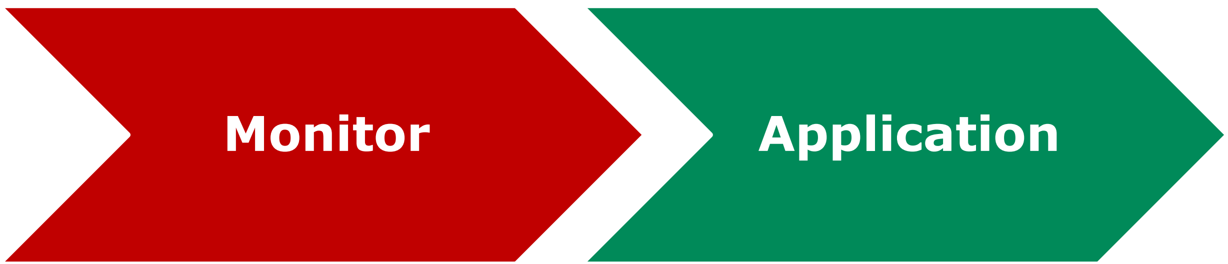
Difference Engine

“The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible.”

Charles Babbage
1791 – 1871

Performance Analysis

- Monitoring infrastructures that capture performance relevant data during application execution



Agenda

- Methodology
- Implementation
- Case Study
- Conclusion

Methodology

- Source code annotations (hooks)
- Hooks invoke the monitor

Source Code Instrumentation

Methodology

```
void func(int i)
{

      if (i>0)
      {
            func(i-1);
      }

}
```

```
void func ( int i)
{
      ENTER ("func");
      if (i>0)
      {
            func(i-1);
      }
      EXIT ("func");
}
```

Methodology

Instrumentation techniques

- Manual
- Automatic
  - Compiler instrumentation (e.g., Clang option *-finstrument-functions*)
  - LLVM compiler pass

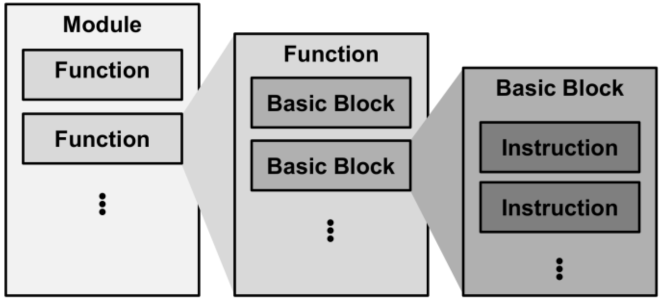
Methodology

Requirements

- Instrumentation of function enter and exit events
- Independence from the programming language of the source code
- Support of filtering options both at compile time and runtime
- Support for user defined filter rules
- Avoid interference with optimizations applied by the compiler
- Internal handling of meta data
- Exception-aware instrumentation

Methodology

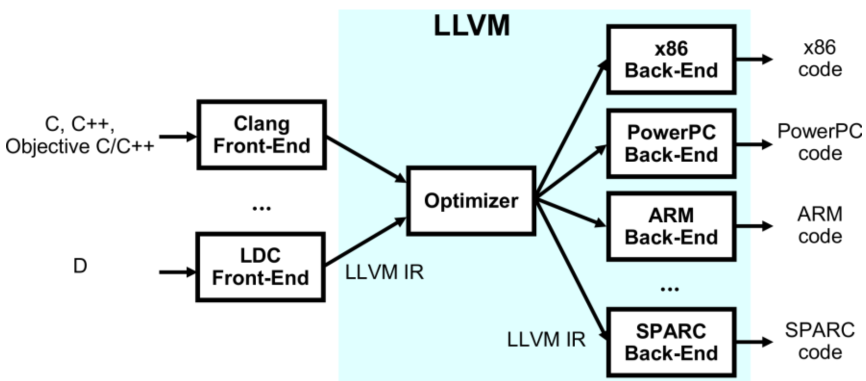
- Implementation of a FunctionPass using the LLVM Pass Framework
- Invoked for each application function
- Insert hooks into the LLVM Intermediate Representation (IR)
- Applying filtering techniques in order to realize selective function instrumentation at compile-time



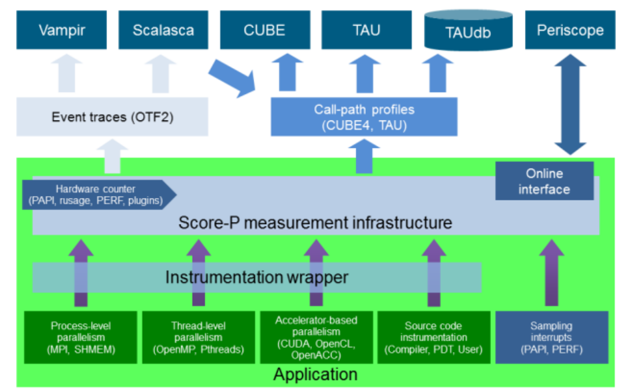
Portion of the LLVM IR relevant for this work

Implementation

- LLVM pass implementation to ensure independence from the programming language of the source code
- Integration in the Score-P monitoring infrastructure



LLVM infrastructure overview



Overview of the Score-P monitoring infrastructure and related analysis tools

Implementation

Override virtual method *runOnFunction(Function &F)* which is called for each function in the processed IR

- Collecting meta data
- Deciding whether a function is instrumented
  - Default filtering rules
  - User defined filtering rule set
- Adding calls to the monitoring infrastructure

Implementation

```
FUNCTION :
static uint32_t handle = INVALID_REGION ;

if ( handle == INVALID_REGION ) register_region( &descr );
if ( handle != FILTERED_REGION ) enter_region( handle );
try {
    /* FUNCTION BODY */
}
finally {
    if ( handle != FILTERED_REGION ) exit_region( handle );
}
```

Implementation

Instrumentation plug-in usage

- Pass is built as a shared library
- Compiler loads this shared library to enable instrumentation at compile-time
- LLVM pass registry manages registration and initialization of the pass subsystem at compiler startup

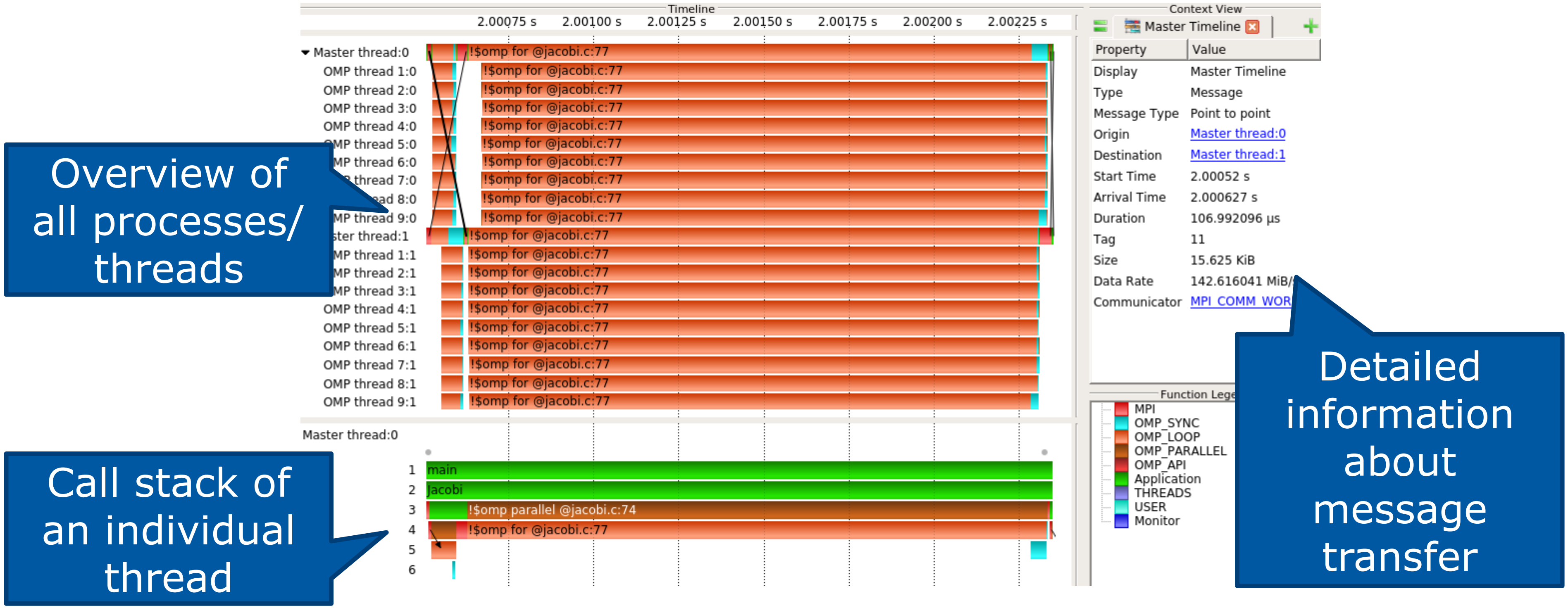
```
clang -Xclang -load -Xclang <instrumenation_pass_library.so>
-c main.c
```

Case Study

Comparison of event sequences

- Instrumentation of a Jacobi solver application (MPI+OpenMP) with
  - Automatic compiler instrumentation
  - LLVM instrumentation plug-in

Case Study – Comparison of Event Sequences



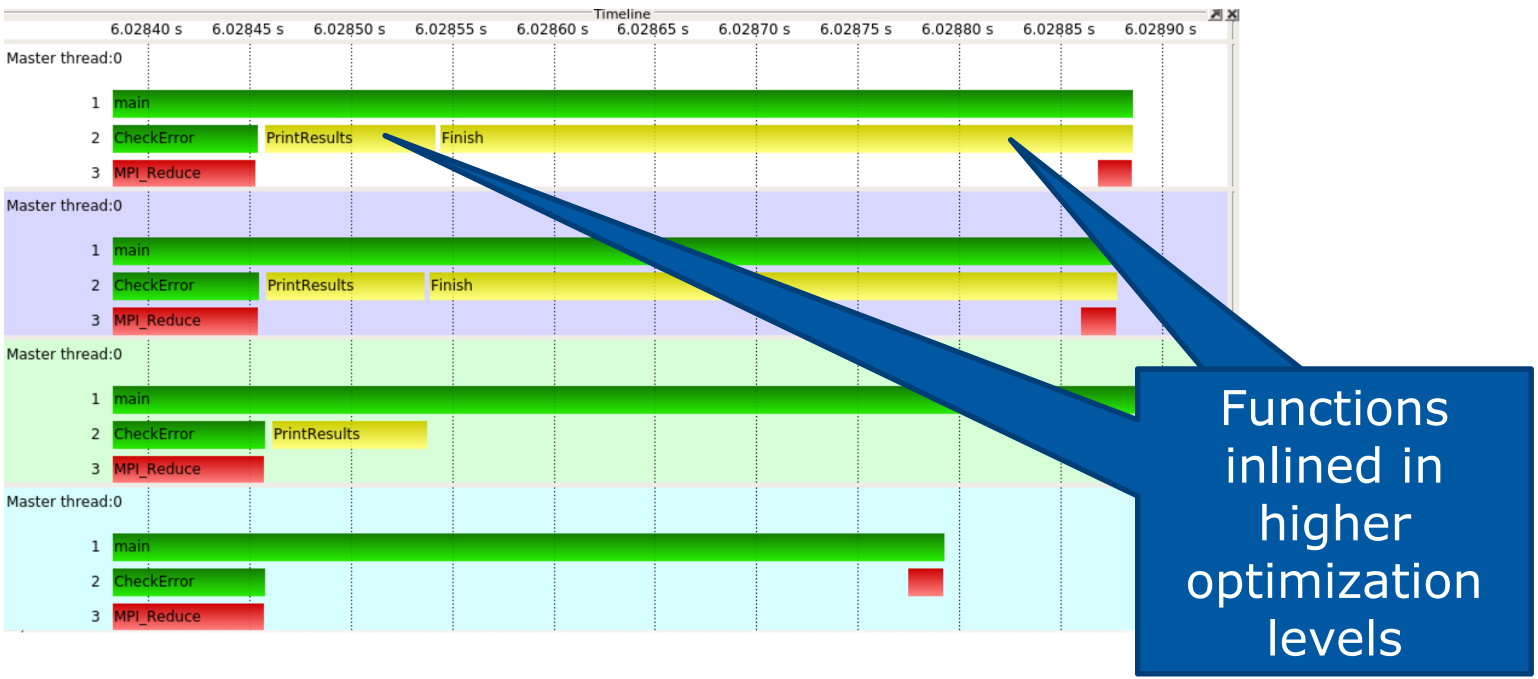
Timeline visualization of the recorded event sequence in Vampir

Case Study – Comparison of Event Sequences

- Number of user function invocations over all processing elements

	Number of user function invocations	
Optimization level	Automatic compiler instrumentation	Instrumentation via plug-in
`-O0`	2014	2014
`-O1`	2014	2014
`-O2`	2014	2010
`-O3`	2014	2008

Case Study – Comparison of Event Sequences



Call stack visualization of the Jacobi application compiled with different optimization levels

Case Study

Comparison of runtime overheads

- Instrumentation of the miniFE application (OpenMP) with
  - Automatic compiler instrumentation
  - LLVM instrumentation plug-in

Case Study - Comparison of Runtime Overheads

- Runtime in seconds of the miniFE experiments
- Each experiment was executed three times, the minimum of these runs is shown

Experiment	Runtime in seconds
Uninstrumented	6
Automatic compiler instrumentation	800
Automatic compiler instrumentation, runtime filter	140
Instrumentation via plug-in	27
Instrumentation via plug-in, compile-time filter	7

Conclusion

- LLVM plug-in supporting
  - Exception-aware instrumentation
  - Selective instrumentation of specific functions at compile-time
  - Runtime filtering

- Feedback
  - Transferring additional information from the Front-End to the Optimizer (source code location, demangled function names, mark internal functions)