Applying the Roofline Model

Ruedi Steinmann

March 8, 2012

Abstract

This is the paper's abstract ...

Contents

1 Introduction

The roofline model [5] premises to be an insightful model for analyzing program performance. To our knowledge, there is no tool capable of easily creating roofline plots. During this master thesis, we would like to create

such a tool. Emphasis lies on correctness of the results produced by the tool and on it's maintainability.

Code to be measured can come either in the form of a relatively isolated routine, typically containing a single kernel, or in the form of a larger program, containing multiple different kernels. While routines are treated as atomic units, it is desirable to split the larger program into it's different algorithms and do the analysis for each algorithm separately. In the rest of this paper, we will use the term kernel for a routine or part of a program which is analyzed on it's own.

To obtain the peak performance lines for the roofline model, we need to run micro benchmarks. To get the data points for a kernel, we need to measure the performance and the operational intensity.

Performance is defined as amount of work per unit of time. The amount of work and how it is defined is determined by the actual kernel and may not be need to be measured directly by the measurement tool. The time required to do the work has to be measured.

Operational intensity is defined as operations per byte of data traffic. Since the operational intensity cannot be measured directly, we have to measure the operation count and the data traffic. Depending on the problem, different definitions of operation and data traffic make sense.

An operation could be a floating point operation, resulting int the well known "flops" unit, either single or double precision. But an operation could as well be defined as machine instruction, integer operation or others.

The point where data traffic is measured is typically between the last level cache of the processor and the DRAM, but it could be measured as well between the different cache levels or between L1 cache and processor.

The micro benchmarks are typically designed to either transfer as much data per unit of time or to execute as may operations per unit of time. Thus the measurement capabilities needed to evaluate the actual problems can be reused for the benchmarks.

In summary, we need to measure the following quantities:

- execution time
- memory traffic
- operation count

On current processors, measuring these quantities should be possible. For the execution time either the cycle counter or the system timer can be used. The memory traffic can be determined using the performance counters for counting cach misses and write back operations. An other option is to use the

counters for bus transfers. Measuring the operation count depends heavily on the definition of operation, but is typically measurable with the performance counters, too.

2 Effects Affecting the Measurement of Multiple Quantities

While each of the measurements required to produce a roofline plot has it's own subtleties, there are some effects affecting multiple measurements, which we will discuss in the following sections.

2.1 Context Switches

Unless a special operating system is used, we have to deal with context switches during measurement. On current operating systems, a context switch typically occurs every 10ms due to the timer interrupt. Following the lines of [1], there are two ways to deal with them:

If the execution time of the kernel is small, the operating system will eventually execute the whole kernel without interruption. This can be exploited using the best k measurement scheme. The kernel is repeatedly executed until the k measurements with the smallest execution times show a variation below a certain threshold. These executions were apparently not affected by a context switch, since a context switch would have increased execution time. The HW_INT_RCV performance counter could possibly be used as well to detect context switches.

If a single execution of the kernel takes longer than the period of time between two timer interrupts, it will always be affected by a context switch. In this case, the proposed solution is to execute the kernel in a loop until many context switches occurred. The effect of the context switches can be compensated for by reducing the measured time be a certain factor. The factor depends on the actual system.

The linux kernel offers the possibility to count performance events only during the time a thread actually executes, omitting events happening during kernel execution or while other threads run. This was not treated in the book above, and offers interesting new options.

2.2 Cache

Specially for short running kernels, the initial state of the cache can have a big impact on the memory traffic and execution time. [?] The impact can

be controlled by making sure the caches are either warm or cold.

Getting cold caches can be quite tricky. See [4] section '3. CACHE FLUSHING METHODS WHEN TIMING ONE INVOCATION' for details.

For our tool, we chose a combined approach. We access a large memory buffer, execute more than 32KB code to flush the L1 code cache. In addition, the kernels report the buffers they allocated and the clflush instruction is executed on them.

To get a warm cache, the kernel is usually executed once before starting the measurement. This causes the working set to be loaded in the caches, if it fits, and the code to be in the cache as well.

2.3 System Load

Specially if the measurement includes context switches, the system load has a big impact on the measurement result. But due to caching effects, even the k best measurement scheme might be affected by the system load. Since we can control the system the measurements are performed on, we can control system load. But none the less, it should be recorded along with the measurement, in case something goes wrong with the measurement system setup.

2.4 Multi Threading

Our kernels and benchmarks will use multi threading. This has to be supported by the measurement tool. Since some caches as well as part of the memory bandwidth might be shared among different cores, multi threading can have various effects on the amount of transferred memory and the available bandwidth.

In case of hyper threading, some functional units of a processor core are shared among two threads. This influences the peak performance of the core.

It is possible that the operating system moves a thread from one core to another. Since the overhead of a switching the core is large, it is generally avoided by the scheduler of the operating system. But it can occur, and we have to be prepared for it.

2.5 Frequency Scaling

The core frequency is not constant in current processors. Since the memory latencies and throughputs do not scale with the core frequency, a lower core frequency will generally cause a higher percentage of the peak performance to be reached by the kernel.

On Linux, frequency scaling is controlled so called governors. They can be set through the sys file system. The tool offers the possibility to set the governor. In addition, it is possible to read the current frequency. If the frequency changes during a measurement, this is recored as well.

3 Single Threaded Validation and Reproducibility

To validate the results of single threaded measurements, and to evaluate the reproducibility, we designed three groups of measurements. Each group is focused on one specific quantity, namely time, transferred bytes and operation count. We measured on an idle system only. All measurements are executed with cold caches.

In each group, we measured with the following kernels:

ADD Repeat adding a constant to an accumulator. Everything is performed in registers. We use multiple accumulators and unroll the loop.

Read Read a buffer from memory

Write Overwrite a buffer in memory

Triad Perform $a_i = b_i + k * c_i$. This involves reading a, b and c and writing a back.

3.1 Transferred Bytes

We tried to estimate the amount of memory that has to be transferred. For pure reading, we expect to observe the whole buffer beeing transferred to the core. When writing is involved, we expect some of the writes to be held back in the cache. It is hard to estimate the number of writes which is actually held back, therfore we just presume that the whole cache is used for write back.

Figure 1 shows that we roughly observe the expected amount of memory transfer. The errors (see fig. 2) decrease for growing buffer sizes.

3.2 Time

The first two graphs show the result of measuring the execution time of the ADD kernel. Figure 3 shows that the execution time is around one cycle per operation and does not differ between SSE and x87, which is consistent with

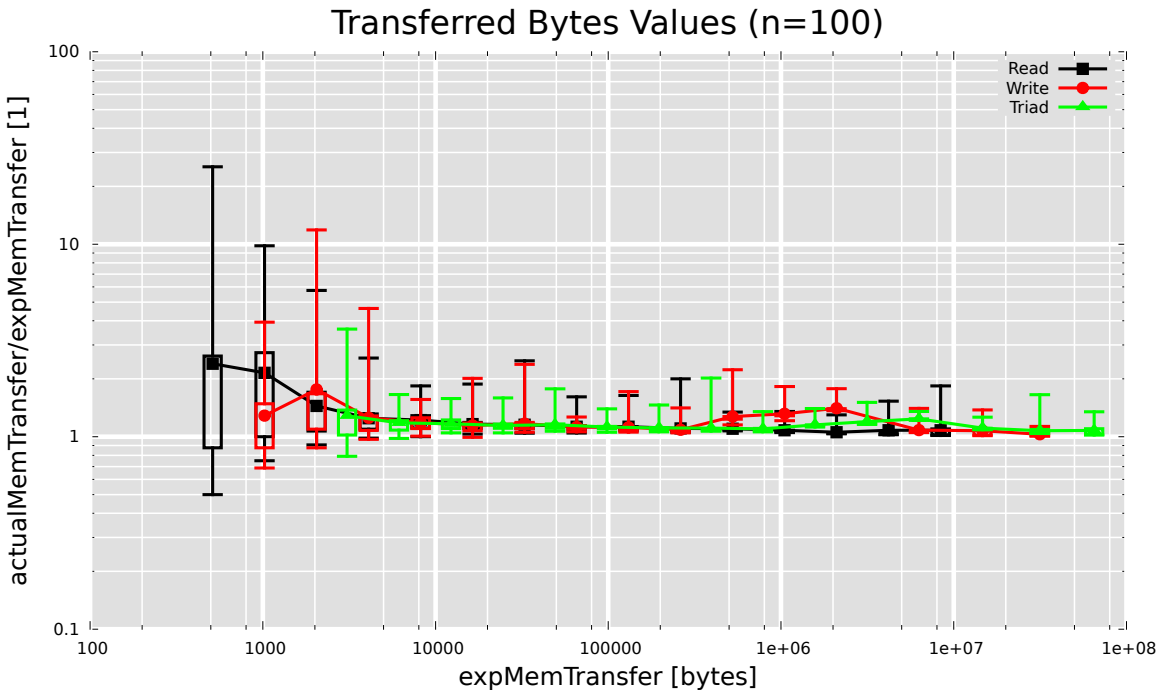


Figure 1: Yonah: Transferred Bytes

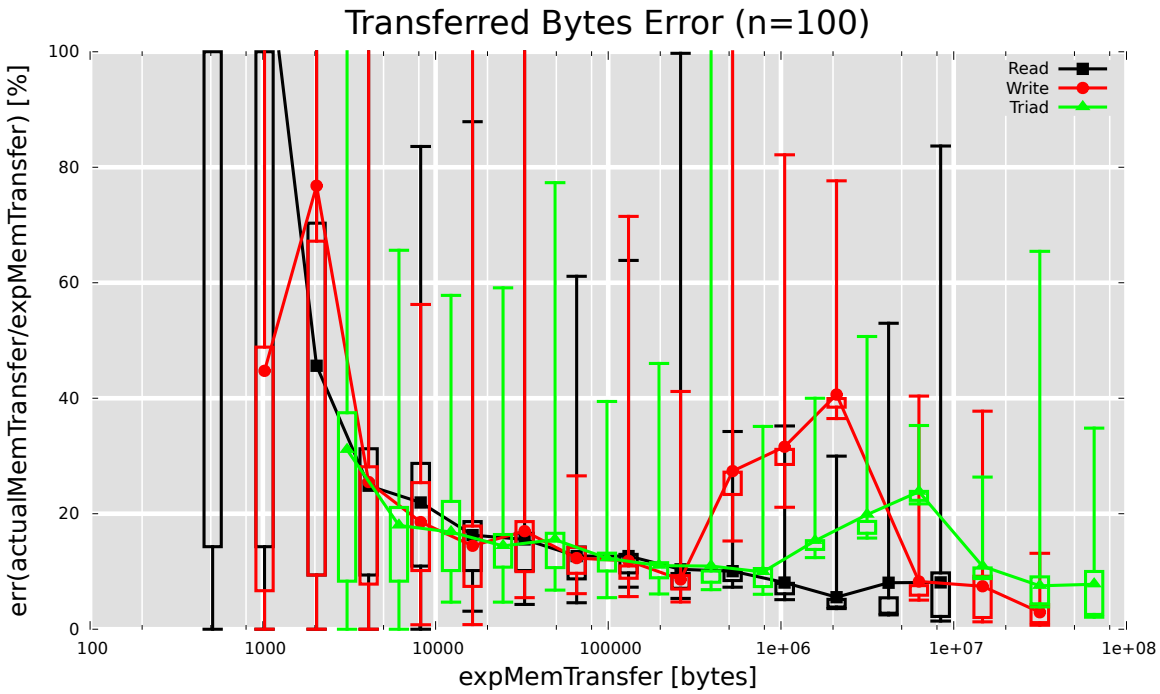


Figure 2: Yonah: Error of Transferred Bytes

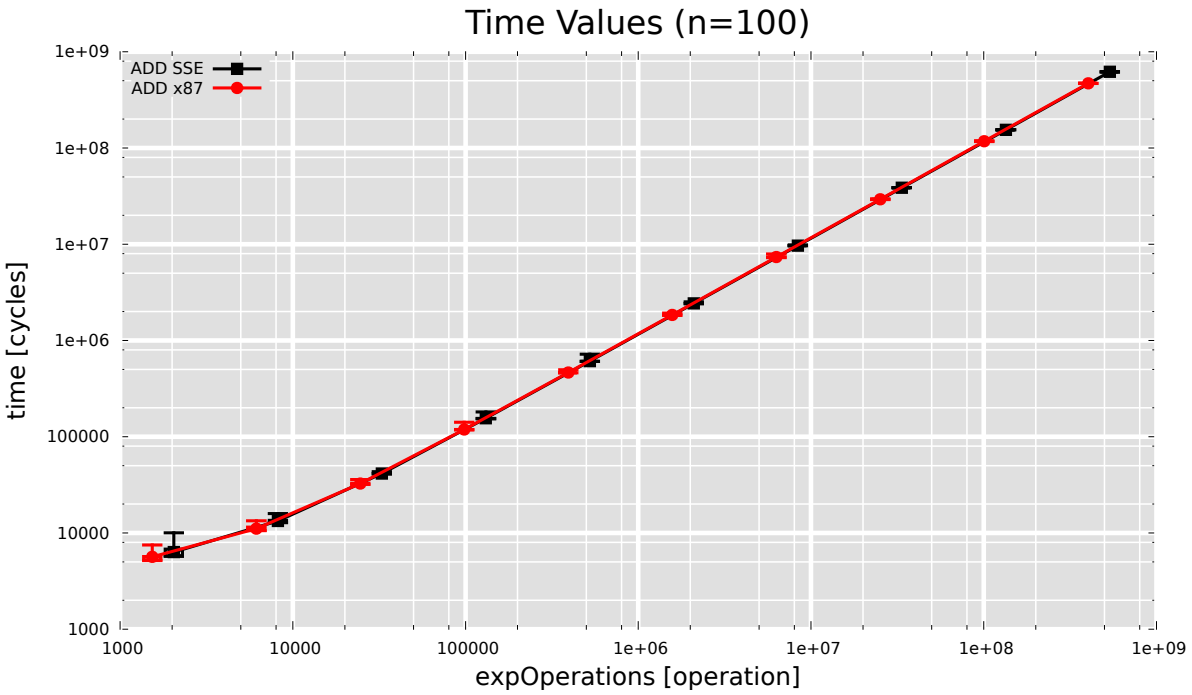


Figure 3: Yonah: Execution Time of the ADD kernel

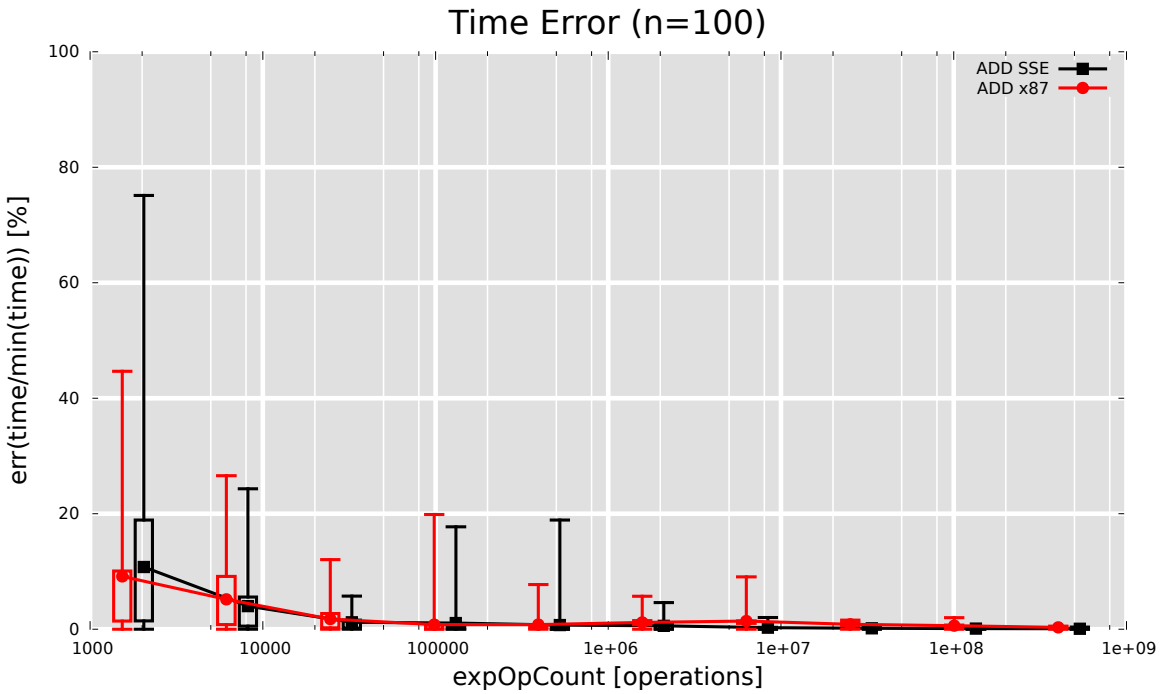


Figure 4: Yonah: Error of the Execution Time of the ADD kernel

the information found in the Intel manual. Figure 4 shows that the errors are small for operation counts above 10000.

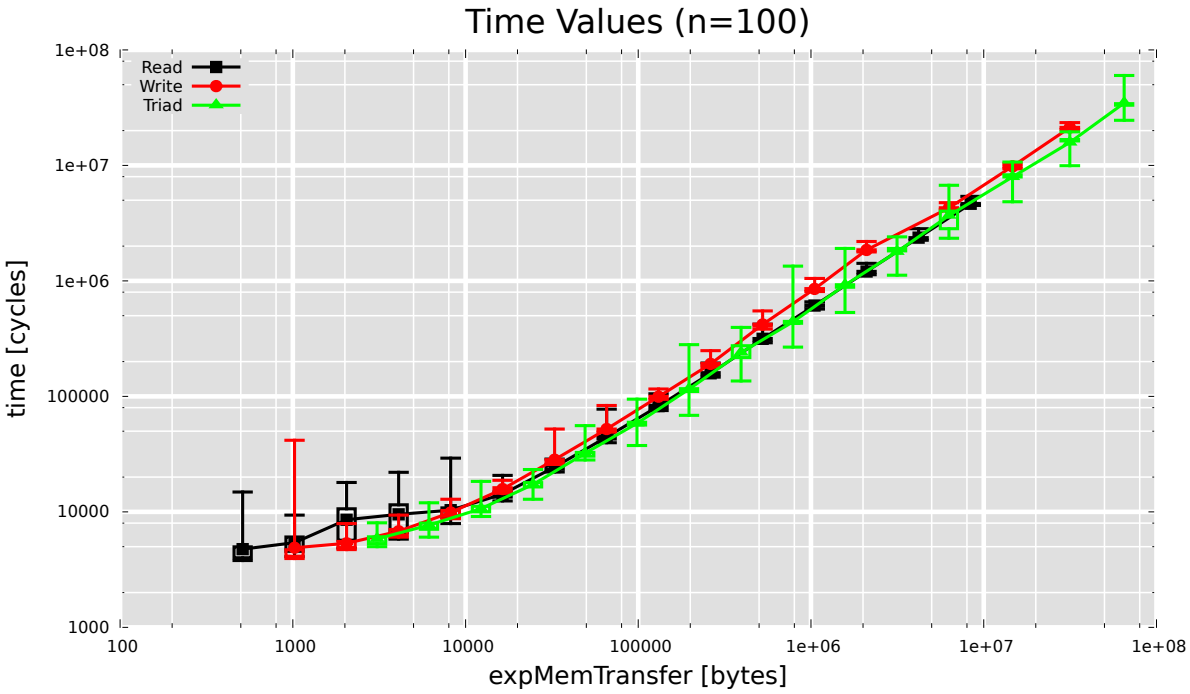


Figure 5: Execution Time of the memory kernels

sdf

For the memory intesive kernels, we show the expected memory transfer on the x axis. (fig. 5)

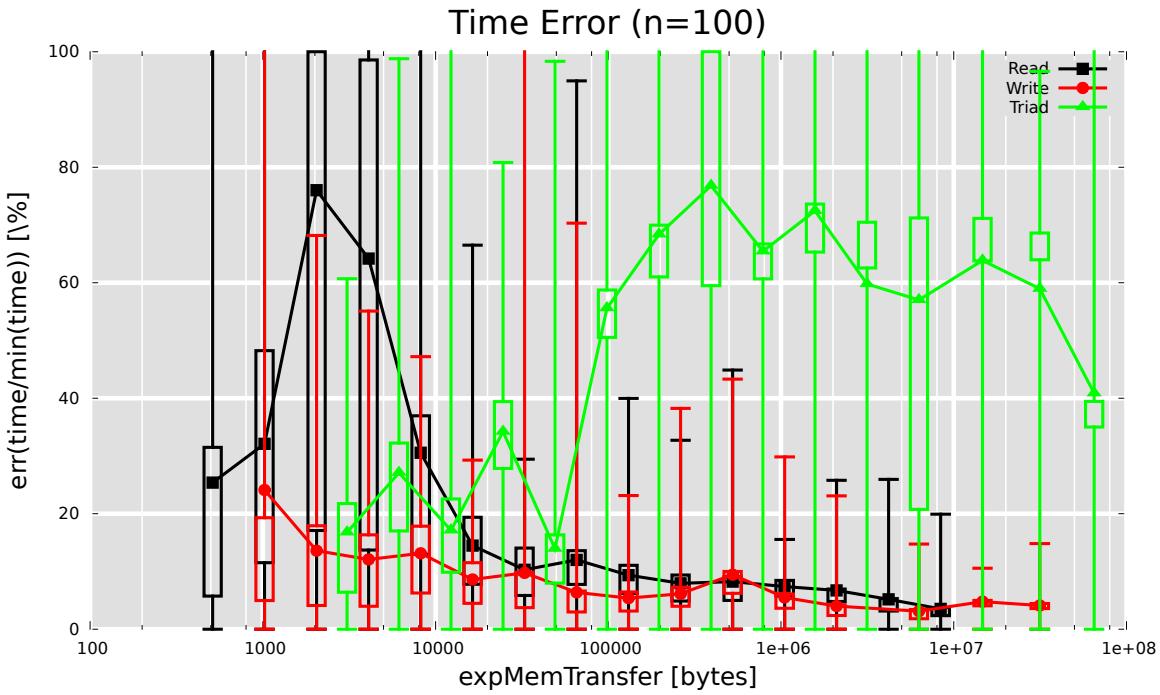


Figure 6: Error of the Execution Time of the memory kernels

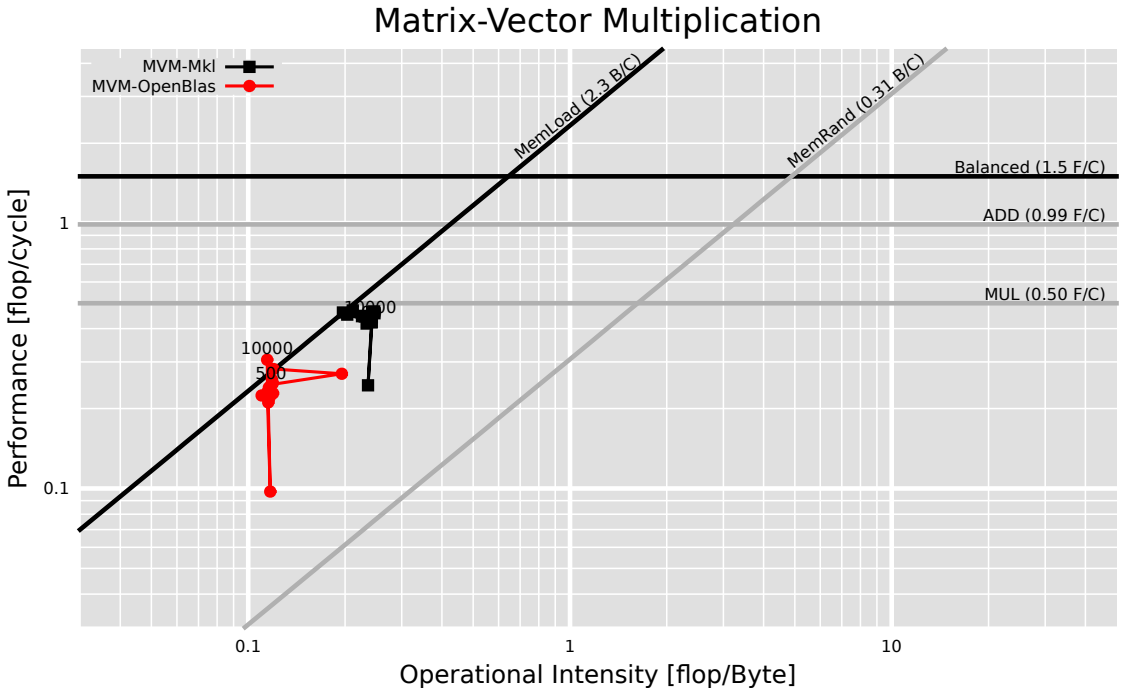


Figure 7: bar

3.3 Operation Count

4 Measurement Tool Architecture

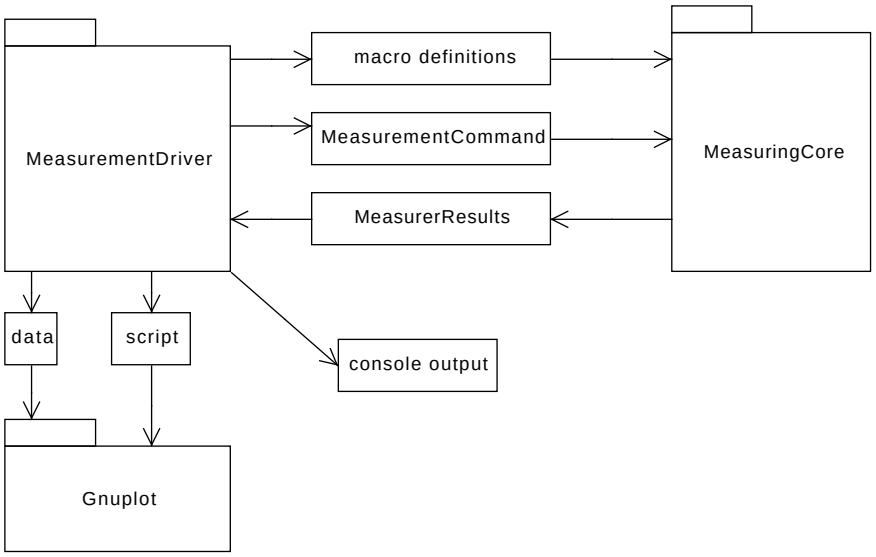
The tool consists of two main components: The Measuring Core, which performs the actual measurements, and the Measurement Driver, which controls the core.

High performance code is generally written in C or C++. Therefore the core is written in that language. To simplify development and maintenance, we tried to keep the amount of C++ code as small as possible. Therefore the measurement driver is written in Java.

A measurement is controlled by a measurement specific routine (one per measurement), which iterates through all parameter points to be examined, compiles and starts the Measuring Core and processes the results. This should result in straight forward code for controlling the measurements and, since the control code is written in Java, a minimal amount of new concepts has to be learned.

During the development of measurement control routines, it is expected that many changes do not affect the parameter points. To speed up repeated measurements after changes to the measurement driver, the measurement results are cached. Thus, as long as the measurement parameters are not changed, the measurement does not need to be repeated.

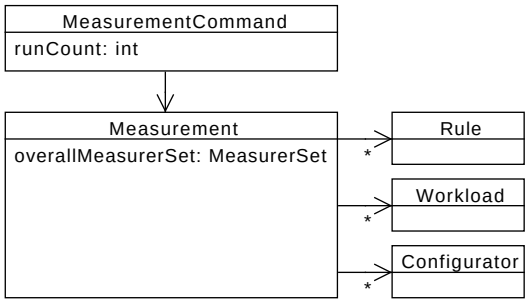
The measurement tool is used to generate and display measurement results. Often, the measurement results lead to changes to the tool itself. Thus, switches between using the tool and developing the tool are frequent. To support these switches, a frontend program is provided. It compiles the measurement driver and executes it. It can be started using a shell script called "rot". The result files of a measurement are placed in the current working directory.



4.1 Component Collaboration

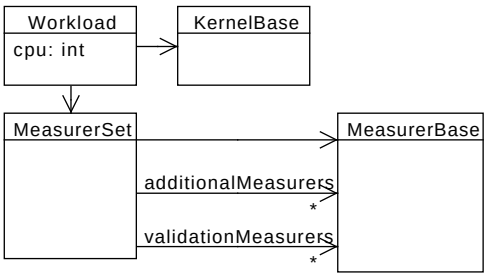
Data transfer between the measuring core and the measurement driver is achieved using serialized objects stored in files. Classes of these objects are used both from C++ and Java and therefore shared entities. They are described using XML. Source code for both languages is generated. For details, see 4.3

The root class for describing a measurement is a MeasurementCommand. It contains the number of times the measurement should be repeated, as well as the actual Measurement. The kernels are contained within the workloads, and rules allow to respond to various events happening during the measurement.

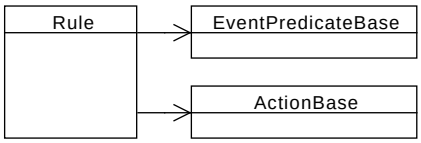


A workload describes what should be run and measured on one core. Each workload is run within a separate thread, which is optionally pinned to a fixed core. In this thread, the validation measurers are started, the caches are warmed up, the additional measurers are started, followed by the main

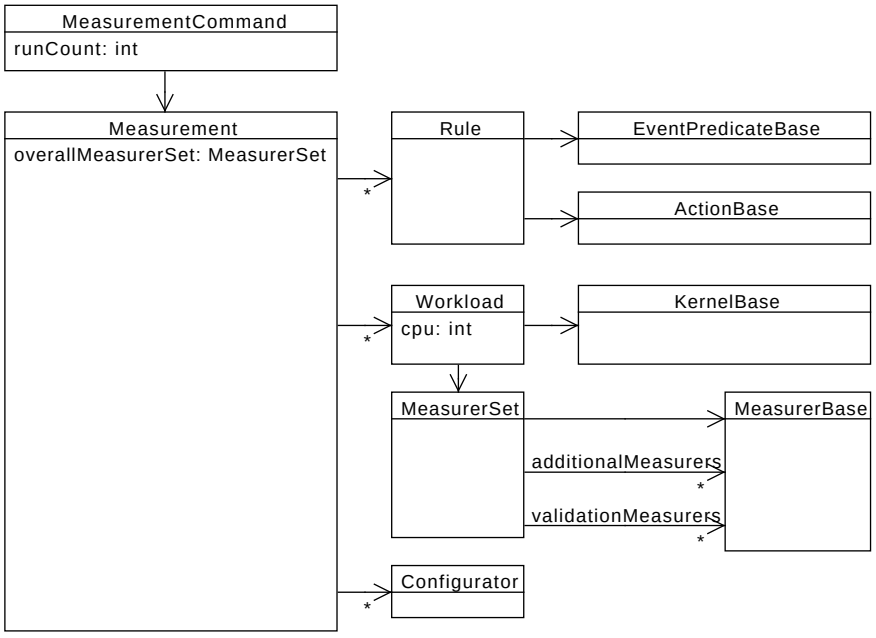
measurer. Then the kernel is run and the measurers are stopped.



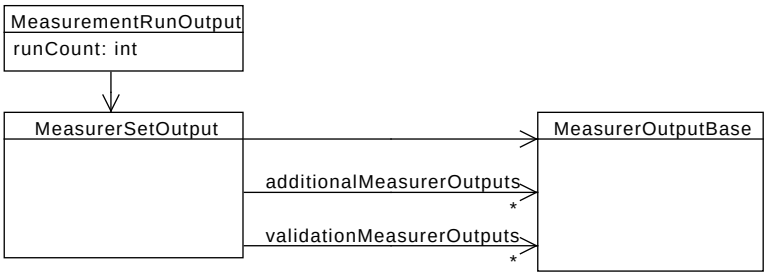
During the measurement, events are raised. For example: start of a workload, end of a workload, start of a thread etc. These events are matched against the event predicates stored in the rules. If a predicate matches, the action of the rule is executed.



The following diagram shows all classes describing a measurement together:

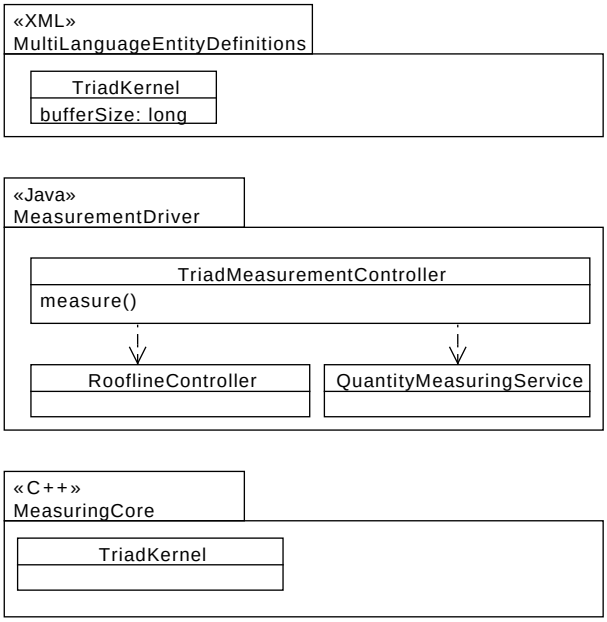


A measurement is usually repeated multiple times, to get an idea of the distribution of the results. Each repetition is called measurement run. In each run, the outputs of all measurers are collected. At the end of the measurement, the core serializes the results of all runs into a single file, which is read by the driver.



4.2 Tour of a Measurement

In this section, we'll look at the components specific to a measurement. To make sure you don't get lost, here is the tour map:



First, we'll look at the kernel. It is defined in an XML file:

```
1 <?xml version="1.0" encoding="UTF-8"?>
  <derivedClass
```

```
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-
            instance"
        xsi:noNamespaceSchemaLocation="../shared.xsd"
5       name="TriadKernel" <!-- name of the class -->
        baseType="KernelBase"
7       cSuffix="Data"
        comment="Kernel performing a=b+k*d
9           on a memory buffer">
        <field
11          name="bufferSize"
            type="long"
13          comment="The size of the buffer"/>
</derivedClass>
```

Kernels are always derived from KernelBase, hence the derivedClass element on line 2 and the base type defined on line 6.

The cSuffix is given as 'Data' on line 7. This causes the generated class to be named 'TriadKernelData'. The measuring core implements 'TriadKernel', which derives from TriadKernelData. The serialization service will instantiate the derived class. This mechanism allows to use a derived class, optionally with additional code and data, to be used in the measuring core. This is how the actual algorithm is implemented. We'll look at this later.

On line 10 starts a field definition. Fields and getters/setters are generated for the field.

Next comes the class controlling the whole measurement.

```
package ch.ethz.ruediste.roofline.measurementDriver.
    measurementControllers;
2
public class TriadMeasurementController implements
    IMeasurementController {
4
    public String getName() {
6       return "triad";
    }
8
    public String getDescription() {
10      return "runs the triad kernel";
    }
12
    @Inject
14  public QuantityMeasuringService
```

```
        quantityMeasuringService;

16      @Inject
        public RooflineController rooflineController;
18
        public void measure(String outputName) throws
            IOException {
20      ...
        }
22  }
```

The class implements IMeasurementController and has to be placed in the measurementControllers package. The measure command will instantiate the class and call the measure() method. The getName() method returns the name of the measurement, which is used to identify the measurement.

The two fields with the @Inject attribute are initialzed by the dependency injection framework when the class is instantiated. The quantity measuring service allows to measure quantities like operation count, transferred bytes, performance etc. The roofline controller manages a roofline plot. We will see how these facilities are used when we look at the body of the measure function:

```
public void measure(String outputName){
2       rooflineController.setTitle("Triad");
        rooflineController.addDefaultPeaks();
4
        for (long size = 10000; size < 100000; size +=
            10000) {
6           // initialize kernel
            TriadKernel kernel = new TriadKernel();
8           kernel.setBufferSize(size);
            kernel.setOptimization("-O3");
10
            // add a roofline point
12          rooflineController.addRooflinePoint(
                "Triad", Long.toString(size),
14              kernel, Operation.CompInstr,
                MemoryTransferBorder.LlcRam);
16
            // measure the throughput
18          Throughput throughput =
                quantityMeasuringService.measureThroughput(
```

```
                kernel, MemoryTransferBorder.LlcRam,
                    ClockType.CoreCycles);

            // measure the operation count
            OperationCount operations =
                quantityMeasuringService
                    .measureOperationCount(kernel, Operation.
                        CompInstr);

            // print throughput and operation count
            System.out.printf("size %d: throughput: %s
                operations: %s\n", size,
                    throughput, operations);
        }

        rooflineController.plot();
    }
```

First the roofline plot is initialized with the title and the default peaks. Then, for each buffer size, the kernel is initialized. For each kernel, the optimization flags used to compile the kernel have to be specified.

Then the roofline controller is instructed to add a roofline point to the plot. The first argument is the series name, next the label of the data point. Points with the same series name are connected with a line in the plot. The rest of the arguments specify the kernel and how the required quantities should be measured.

In the rest of the loop, the throughput and the operation count are measured and printed to the console. This is an example of how to use the quantity measuring service.

The last statement of the measure() body causes the plot to be output to a file in the current directory. This involves the invocation of gnuplot.

During the invocation of addRooflinePoint() and the quantity measuring service a lot was going on under the hood. First a measurement was created from the kernel and the measurers required to measure the requested quantities. Then was checked if there is already a result for the measurement in the cache. If not, the measurement was serialized, the measuring core was configured, built and started. Then the result of the core was parsed and stored in the cache. And finally, the requested quantities were caculated.

The only measurement specific part involved in this process is the implementation of the kernel. First the header:

```
class TriadKernel : public TriadKernelData{
```

```
    double *a,*b,*c;

protected:
    std::vector<std::pair<void*,long> > getBuffers();

public:
    void initialize();
    void run();
    void dispose();
};
```

The kernel requires three buffers. All declared methods override methods from the KernelBase. The buffers are allocated and initialized in initialize() and freed in dispose(). getBuffers() returns the buffers along with their sizes. This is used to clear the or warm the caches. run() contains the actual algorithm.

```
void TriadKernel::initialize() {
    srand48(0);
    size_t size = getBufferSize() * sizeof(double);

    // allocate the buffers
    a = (double*) malloc(size);
    b = (double*) malloc(size);
    c = (double*) malloc(size);

    // initialize the buffers
    for (long i=0; i<getBufferSize(); i++){
        a[i]=drand48();
        b[i]=drand48();
        c[i]=drand48();
    }
}

std::vector<std::pair<void*, long> > TriadKernel::
   getBuffers() {
    size_t size = getBufferSize() * sizeof(double);

    std::vector<std::pair<void*, long> > result;
    result.push_back(std::make_pair((void*) a, size));
    result.push_back(std::make_pair((void*) b, size));
    result.push_back(std::make_pair((void*) c, size));
```

```
25      return result;
    }
27
    void TriadKernel::run() {
29      for (long p = 0; p < 1; p++) {
            for (long i = 0; i < getBufferSize(); i++) {
31              a[i] = b[i] + 2.34 * c[i];
            }
33      }
    }
35
    void TriadKernel::dispose() {
37      free(a);
        free(b);
39      free(c);
    }
```

4.3 Multi Language Infrastructure

The shared entities are used from both C++ and Java. To avoid having to manually synchronize two versions of the same class, the source code for the C++ and the Java implementation is generated from an XML definition by the Shared Entity Generator. The XML definition contains class and field definitions only, no code. If class specific code is needed, it has to be implemented separately for each language and merged with the field definitions using inheritance.

The shared entity definitions, written in XML, are parsed using a serialization library called XStream. XStream maps classes to an XML representation. The classes used to define the shared entities are shown in Figure 8.

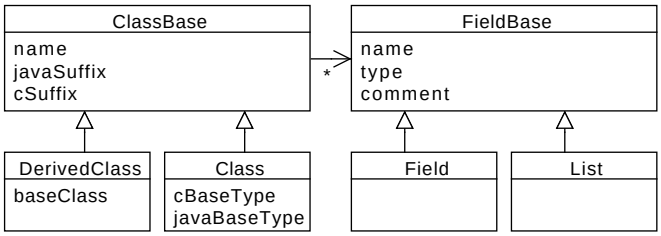


Figure 8: Classes representing a multi language class definition

The following is an example of a class definition:

```
<?xml version="1.0" encoding="UTF-8"?>
<class name="MultiLanguageTestClass"
  cBaseType="MultiLanguageObjectBase"
  javaBaseType=""
  comment="Multi Language Class used for unit tests">

  <field
    name="longField"
    type="long"
    comment="test field with type 'long'"/>
  <list
    name="referenceList"
    type="MultiLanguageTestClass"
    comment="list referencing full classes"/>
  <field
    name="referenceField"
    type="MultiLanguageTestClass"
    comment="field referencing another class"/>
</class>
```

After the definitions are loaded, Velocity templates are used to generate all source code.

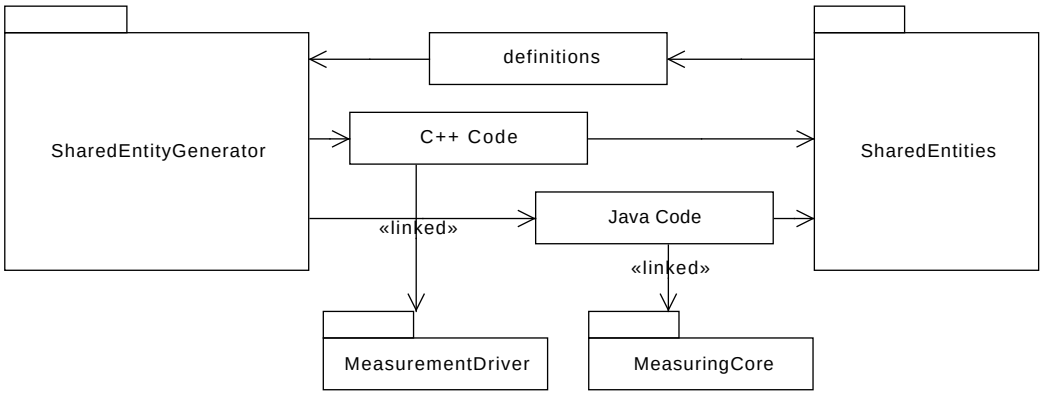
A normal entity has a C and a Java base type. The C base type has to directly or indirectly inherit from SharedEntityBase, which is a polymorphic class. This allows to use the RTTI (RunTime Type Information). Java base types have no such constraint (due to the implicit common base class Object). The base types are just included in the generated source code, but have no other effect on the code generation.

A derived entity names another entity as base type. The C and Java base types are set to that class. The fields of the base class are included in the serialization process. If just the C and Java base types would be set to the shared entity used as base class, the generated class would still derive from the base class, but the fields of the base class would not be included in the serialization process.

Often it is necessary to mix hand written code with the generated code. To support this, a suffix can be specified, which is added to the name of the generated class. Only the name of the generated class is affected, not the type name used for references to the class. A class named without the specified suffix has to be provided manually, and should derive from the generated

class. Any additional code as well as additional fields can be included in the hand written class.

The class definitions and the generated code is located in the Multi Language Classes project. The generated Java code is linked by the Measurement Driver project. The generated C code is linked by the Measuring Core. The following Diagram shows these dependencies:



4.3.1 Serialization and Deserialization

Along with the source code for each class, a service serializing and deserializing multi language objects to/from a simple text based format is generated for both languages. It supports the following primitive types:

- double
- integer
- long integer
- boolean
- string

References to other shared entities are supported. The serializer does not recognize if the same object can be reached multiple times within the same object graph. Each time it encounters an object, the object is serialized instead of referencing the previous serialization.

Lists containing one of the supported primitive types as well as containing references to other shared entities are supported.

The service implementations for both languages follow the same structure. Each has two methods, one for serialization and one for deserialization.

The serialization method receives an object and an output stream. The method body contains an if for each known serializable class, which checks if the class of the object received is equal to the serializable class. If true, the value of all fields of the class and it's base classes get serialized. For references, the serialization method is called recursively with the same output stream and the referenced object as parameters.

The deserialization method works analogous to the serialization method. It receives an input stream. The method body contains an if for each known serializable class, which checks if the next line of the input names the serializable class. If true, a new instance of the class is created and the value of all fields are read from the input and set on the created instance, including all fields declared in a base class. If a reference is encountered, the deserialization method is called recursively with the same input, and the returned instance is used as field value.

4.4 How To

4.4.1 Install

see INSTALL file in tool directory

4.4.2 Create New Kernels

- create the xml description of your kernel description in multiLanguageClasses/definitions/kernels
- run "rmt help" to generate the multi language code from you description
- create your kernel implementation in measuringCore/kernels. Subclass "Kernel", parameterized to the description class you've created. Implement "initialize()", "run()" and "dispose()".
- register your kernel class. In the cpp file of your kernel implementation, include "typeRegistry/TypeRegisterer.h" and instantiate a static global variable of type "TypeRegisterer", parameterized to your kernel implementation.
- use your kernel from a measurement

4.4.3 Create New Measurement

- create new class in measurementDriver/measurements

- implement IMeasurement

4.4.4 Add Configuration Key

The configuration is used to set various flags in the measurement driver.

- add public static field of type ConfigurationKey to any class within the measurement driver.

4.4.5 Generate Annotated Assembly

- in Eclipse, hit build (Ctrl+B)
- change the kernel header file (make it recompile)
- build again
- from the console window, copy the compiler invocation for MeasurementSchemeRegistration.cpp
- open a terminal and go to tool/measuringCore/Debug.
- paste the compiler invocation
- insert "-Wa,-ahl=ass.s", check optimization flags
- issue command
- the annotated assembly code can be found in tool/measuringCore/Debug/ass.s
- open the annotated assembly code in Eclipse

4.5 Frontend

The measurement tool is a console tool controlled using command line options. Measurement results are either directly dispalyed on the console, dumped to a data file or processed, usually for generating a graph. The graph is typically stored as a file. But unlike normal tools, the source code is expected to change frequently, and the user likely switches often between coding and using the tool.

To support this usage pattern, the build process has been integrated into the normal tool operation. The frontend is used to first trigger the build process and then invoke the measurement driver. Otherwise, the user would have to keep to console windowses open, one for building and one for

measuring, and not to forget building to see the changes made to the source code.

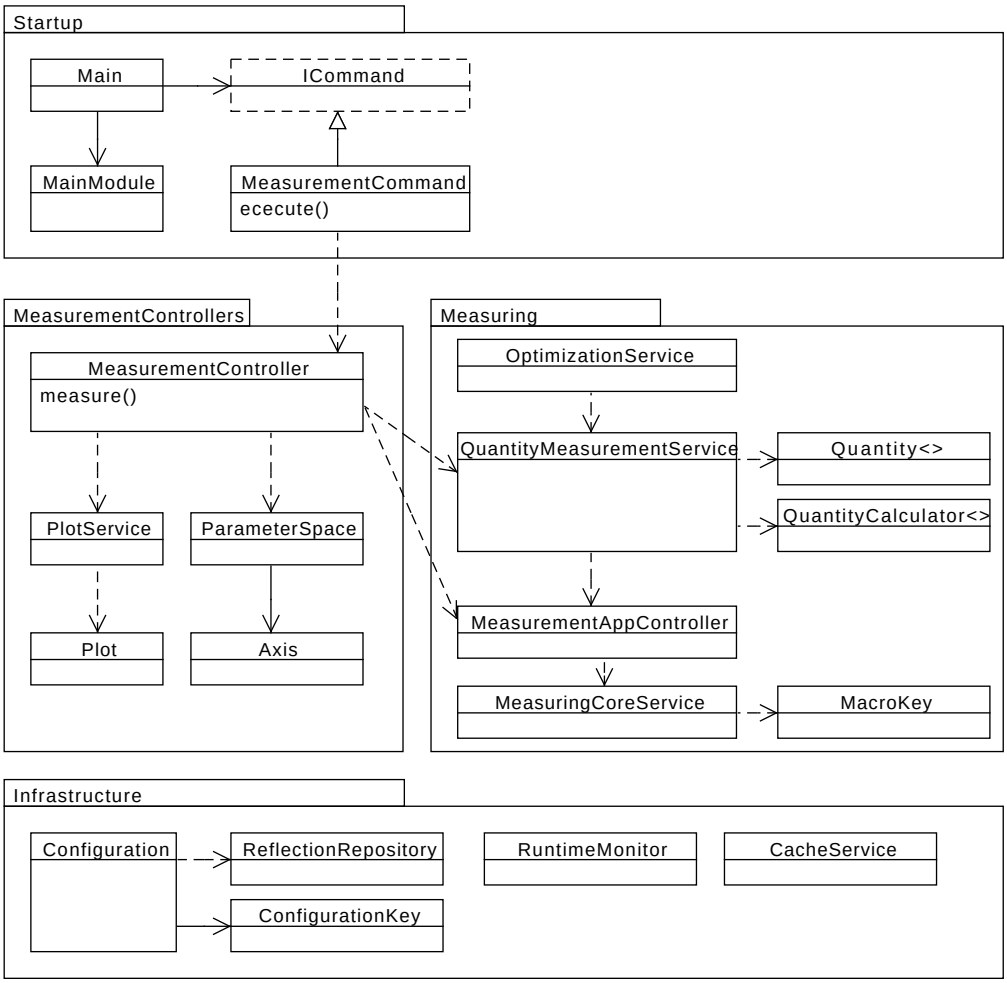
After building, the frontend starts the the measurement driver, forwarding it's own command line. Certain flags are used to control the operation of the fronted. These are not forwarded.

The frontend has a configuration system. The known configuration keys are defined at the top of the Main class. There are three configuration sources. The default configuration stored in a configuration file. It contains templates which are expanded during the build process. The result is included as resource in the generated JAR file. Flags of the default configuration can be overwritten using a user configuration file, which is located by default under " /.roofline/frontendconfig". This location can be changed using a command line argument. Finally, the command line options known by the frontend are used to modify the configuration flags after they have been loaded.

4.6 Measurement Driver

For the design of the measurement driver, we used software engineering best practice, namely unit testing (junit), mocks (jmock), and dependency injection (guice). Further a domain model (DOM), controllers, repositories and stateless services as described in [2]. Describing all these concepts lies beyond the scope of this report. It is assumed that the reader has a basic understanding of the mentioned concepts.

The following diagram shows an overview of the driver:



In the following paragraphs, we will have a quick look at the look at the different parts.

The entry point of the driver is the **Main** class. First, the dependency injection framework is initialized. This is accomplished using the **MainModule**. It's configure() method uses the **ClassFinder** to find all compiled classes and configures how they are instantiated, mainly based on naming conventions.

Then the command line arguments are parsed. If command line auto completion is desired (indicated by the '-autocomplete' flag), the auto completion process starts.

Otherwise the configuration is initialized, using the flags specified at the command line and the configuration files (default configuration stored in the jar and user configuration from the home directory of the user).

The last step in the initialization sequence is to set up log4j, the logging

framework.

Then the class for the command given on the command line is instantiated and the execute() method on the resulting **ICommand** is called.

When a 'measure' command is given, a **MeasurementCommandController** is instantiated. The command controller looks a the next command line argument, instantiates the corresponding measurement controller and calls the measure() method.

The measurement controller will typically create multiple **Measurement**s with different parameters. The **ParameterSpace** facilitates iterating over all possible parameter combinations. Each parameter is associated to an **Axis**. For each axis, one or multiple values can be given. The getAllPoints() returns a **Coordinate** for each possible value combination. Itearating over the points in the space, the measurement controller can construct a measurement for each point.

The results of the constructed measurements can be either printed to the console, or stored in one of various **Plot**s. When all data is gathered, the plot can be rendered and written to an output file using the **PlotService**.

Although it is possible to directly create the **Measurer**s required to measure something, most of the time the intent is to measure a certain **Quantity** (**OperationCount**, **TransferredBytes**, **Performance** etc). The **QuantityMeasuringService** can be used to obtain a **QuantityCalculator** for a quantity. The calculator can be queried for the list of measurers which are required to calculate the quantity. When the results of all required measurers are known, they can be passed to the calculator, which will return the desired quantity. In addition, the quantity measuring service provides convenience methods for working with the quantity calculators.

Once the **Measurement** is constructed, the measure() method of the **MeasurementAppController** is used to perform the measurement. First it is checked if a cached result is available for the measurement (using the **CacheService**). If not, the **MeasuringCoreService** is used to build the core for the measurement and to start the core.

The **Kernel**s can define macros. During build preparation, all macro definitions present in the measurement are collected and written to generated header files within the core.

During the execution of the measurement driver, the run time used for the various tasks is collected using the **RuntimeMonitor**. At the end of the execution, the times spent for the tasks is printed to the console.

4.6.1 Dependency Injection Configuration

Generally, a convention over configuration approach was chosen for the configuration of the dependency injection. The conventions as well as optional exceptions are defined in the MainModule. The conventions are:

Services all classes in the services packages are bound to themselves as singletons

Repositories all classes in the repositories packages are bound to themselves as singletons

Application Controllers all classes in the 'appControllers' package are bound to themselves as singletons

Measurement Series all classes deriving form IMeasurementSeries in the measurement series package are bound to the IMeasurementSeries interface annotated with their name

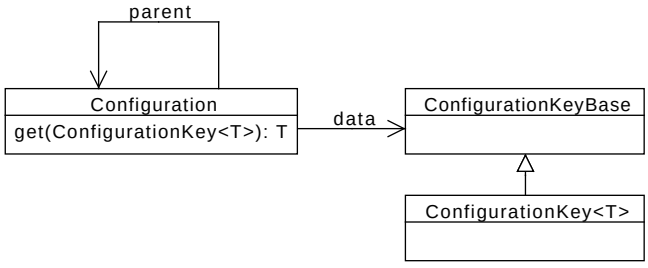
Commands all classes deriving from ICommand in the commands package are bound to the ICommand interface annotated with their name

Measurement Controllers all classes deriving from IMeasurementController in the measurement controller package are bound to the IMeasurementController interface annotated with their name

4.6.2 Configuration

The design goal was to create a configuration system which

- allows to set configuration flags from the command line and from configuration files
- can manage some form of comment for the flags
- makes the available flags transparent
- supports user specific configuration



The central class of our solution is the **ConfigurationKey**. A configuration key contains a string key which identifies the configuration flag it represents. In addition, it contains a description and the default value of the flag. It has a template parameter which defines the data type of the flag. This removes the necessity to use type casts when reading configuration flags. Configuration keys should be stored in public static variables. The help command scans all classes of the measurement driver for such configuration keys and prints the key string and the description. After the configuration is loaded, it is checked if a configuration key is defined for each configuration flag specified. If a configuration key for a flag is missing (or more likely, a configuration flag has been misspelled in the configuration) an error is generated.

The values associated with configuration keys are stored in **Configuration**s. Configurations can be chained together using the parent links. If no value is found in a configuration or all of its ancestors, the default value stored in the configuration key is used.

The state of a configuration kan be saved on a stack using push() and restored using pop(). All modifications to a configuration after a push are undone by the pop. This can be used to temporarily change the configuration.

The following paragraphs describe the sources of configuration flag definitions in order of decreasing precedence.

The command line is scanned for arguments starting with a dash. Such arguments are expected to be in the form of "-<flag key>=<value>" and specifies that the configuration with the specified flag key should have the specified value. Configuration flag definitions on the command line have highest precedence.

Next come two configuration files. They both have the same format: each line consists of the flag key, followed by an equal sign and the flag value.

The first file is the user configuration file. By default it is located under ~/.roofline/config, but this can be changed using the "userConfigFile" configuration flag, in particular by overwriting the flag on the command line.

The second file is the default configuration. It is located in the source code of the measurement driver, and can be loaded from the classpath. It contains some placeholders, which are expanded during the build process.

Finally, the flag definitions with lowest precedence are the default values given in the configuration keys.

4.6.3 Auto Completion

4.6.4 Commands

A command is represented by a class deriving from ICommandController and should be placed in the commands package. A command has a name and a description, which should be the return value of the getName() respectively getDescription() methods of the command. The measurement driver expects a command name as first argument. The name is matched against the names of all available commands. If a command matches, the execute() method of a new instance of the corresponding class is called with the remaining command line arguments as parameter.

4.6.5 Measurement Controllers

The operation of the measurement driver is controlled by the measurement controllers. They define which measurements to perform and how to process the output. The measure command instantiates a measurement controller and calls the measure() method.

4.6.6 Parameter Space

When implementing measurement controllers, one often has to iterate over all possible combinations of some parameters. The **ParameterSpace** was designed to support this.

Every parameter is identified by an **Axis**. For each axis, one or multiple values are specified. After the desired values are specified, all possible parameter combinations can be generated, represented by **Coordinate** objects. The points are generated implicitely when iterating over the space.

Example:

```
space.add(systemLoadAxis, SystemLoad.Idle);
space.add(systemLoadAxis, SystemLoad.DiskOther);
space.add(systemLoadAxis, SystemLoad.DiskAll);
space.add(systemLoadAxis, SystemLoad.AddOther);
space.add(systemLoadAxis, SystemLoad.AddAll);

space.add(clockTypeAxis, ClockType.CoreCycles);
space.add(clockTypeAxis, ClockType.ReferenceCycles);
space.add(clockTypeAxis, ClockType.uSecs);

for (Coordinate coordinate : space) {
```

```
13      ClockType clockType
            = coordinate.get(clockTypeAxis);
15      SystemLoad systemLoad
            = coordinate.get(systemLoadAxis);
17      ...
}
```

To faciliate the initialization of measurements, the classes of the measurement description have an intialize() method which takes a coordinate as parameter. Depending on the kernel or measurer at hand, some fields are set to the value of an axis given by the coordinate.

The most common axes are defined in the **Axes** class.

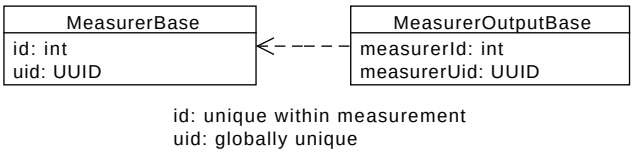
4.6.7 Retrieving Outputs

To process the results of a measurement, it is frequently necessary to retrieve the output of a specific measurer.

The straight forward approach would be to give each measurer an unique id, and to store the id of the measurer with the measurer output. Measurers are newly created with each invocation of the measurement driver, possibly leading to new ids. But for caching, the ids of the measurers do not matter.

To overcome these problems, two ids are generated for each measurer. One identifier uniquely identifying each instantiated measurer. And an id which is unique within one measurement. When loading a result from cache, the unique identifiers of the loaded result are set to the identifiers of the measurement at hand.

To retrieve the output of a measurer, the **MeasurementResult** and the **MeasurementRunOutput** provide several methods which take a measurer as argument and return it's output.



4.6.8 Plotting

4.6.9 The MeasurementAppController

The measurement application controller is the entry point for performing measurements. It is the sole client to the **MeasuringCoreService**, which

provides the low level control over the measuring core, and keeps track of the measurement the core is compiled for.

In addition, it uses the **MeasurementHashRepository** to keep track of measurements which have equal measuring cores.

The main method of the controller is measure(). Functionality in pseudo Code:

```
MeasurementResult measure(measurement, numberOfRuns)
"prepare measurement"
runOutputs=[]
if (useCachedResult || "measurement has been seen")
    loaded="load stored results"
    if (loaded!=null)
        if (!shouldCheckCoreHash
            || currentCoreHash==loaded.coreHash)
            runOutputs=loaded

if ("more results needed")
    newResult=performMeasurement()
    "merge and store loaded and new run outputs"

"build and return MeasurementResult with the desired
    number of run outputs"
```

The method first tries to load a measurement result from the result cache. If not enough run outputs are loaded (or none at all), the measurement is performed to get the remaining measurement run outputs.

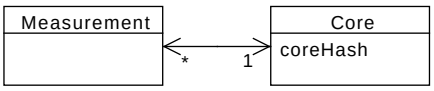
Finally, a measurement result with exactly the requested number of runs is constructed and returned.

It is possible to disable loading stored results by setting the useCachedResults configuration key to false. The measurement is performed and existing results are overwritten.

The hash code of the core is stored along with the measurement result. This allows to check if the currently compiled core is equal to the core which was used during the measurement. By default, if the core changed since the results were generated, the results are not used and new results are generated using the current measuring core. By setting shouldCheckCoreHash to false, this check can be skipped.

Preparing and building the measuring core are expensive operations in term of runtime. Therefore, the measurement application controller keeps track of as much information about the measurements and the cores needed to perform them as possible. The **MeasurementHashRepository** is used

for this purpose. It has the following internal Model:



The **Core** class is private and does not leave the repository.
The model is exposed through the following methods

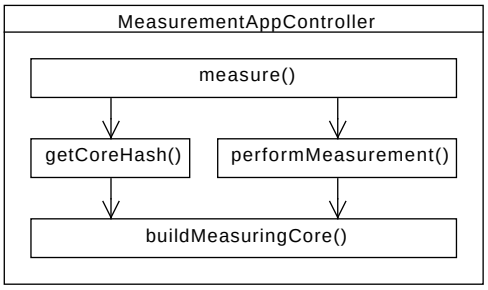
- areCoresEqual(measurementA, measurementB): bool
- setHaveEqualCores(measurementA, measurementB)
- setCoreHash(measurement,coreHash)
- getCoreHash(measurement): CoreHash

Before building, the controller asks the repository if the core for the new measurement is the same as the currently built one. (using areCoresEqual()) If the cores are the same, no building is required.

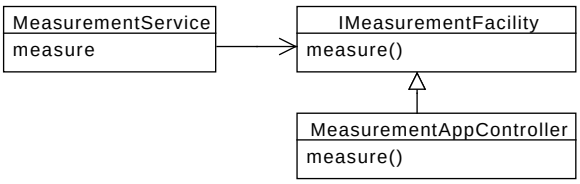
During the build preparation the controller and the **MeasuringCore-Service** monitor changes to the core. If no changes were necessary, the repository is notified using setHaveEqualCores(). The cores of the two measurements are merged.

When a core hash is required (for example to check if the current core is the same as the one used to generate a stored result), the controller first asks the repository for it using getCoreHash(). If the hash is not known, the core is built for the measurement, the hash is calculated from the core and stored in the repository using setCoreHash(). This could again lead to a merging of two cores, if a core with the same hash is present already.

Building the measuring core can become necessary when the core hash of a measurement has to be known, or when a measurement is to be performed. This is reflected in the call graph of the methods within the application controller:



Since it makes sense that services can start measurements, the **MeasurementService** provides a measure() method. The service knows an instance of **IMeasurementFacility** which provides measure(), and forwards all calls to its own measure() method to the measurement facility. The facility is is implemented by the **MeasurementAppController** controller. Therefore the service ultimately forwards all calls to measure() to the application controller.



4.6.10 Architecture Specific Behavior

The measurement driver supports multiple system architectures. Currently, the system architecture is identified by the available PMUs. When a performance event is to be read, a list with the event for each architecture is passed to the getAvailableEvent() method of the **SystemInfoService**. The method returns the available event. In other cases, the presence of a PMU is checked directly.

For further development, it might become beneficial to use the specification pattern.

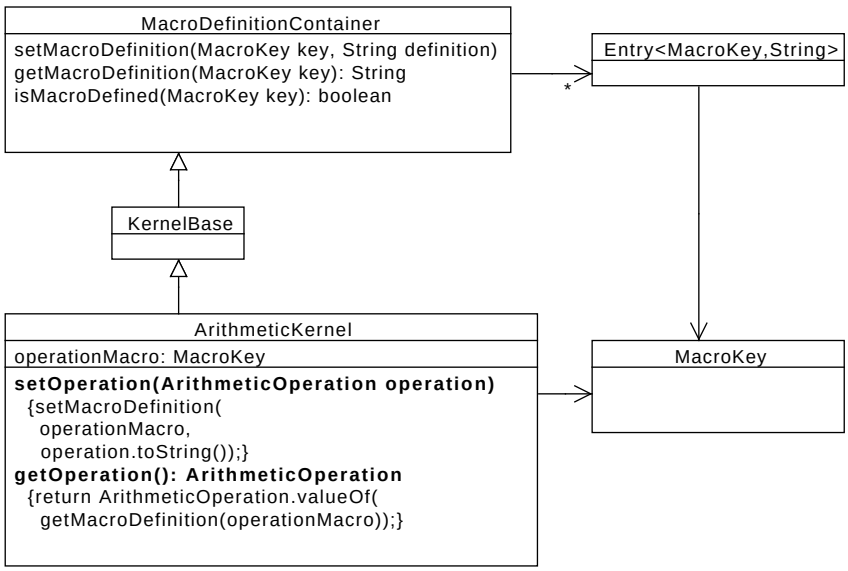
4.6.11 Preprocessor Macros

Preprocessor macros are used to allow flexible compile time parameterization of the measuring core. The macros are defined by the measurement driver. Before the compilation of the measuring core, the measurement driver writes the definition of each macro to a separate include file. This allows the build system to track macro definition changes for and to recompile only the required parts.

In the measurement driver, each macro is identified by a macro key, which contains the macro name, a description and the default value. The macro definitions are stored in classes deriving from MacroDefinitionContainer. The classes should define macro keys by placing them in private static variables. To access the macro definition, getters and setters have to be provided.

When the measuring core is configured to perform a measurement, the macro keys are collected from the classes of the measurement driver using reflection. Then the macro definitions are extracted from the measurement definition and referenced objects. If no definition is given for a macro, the

default definition found in the macro key is used. If contradicting definitions are found, an error is raised.



4.6.12 Measurement Result Caching

The measurement controllers mix the definition of the measurement parameters and the processing of the output. Thus, if the output processing logic needs to be modified, the measurements have to be performed again. This causes a delay, which is avoided by caching the measurement results.

All parameters of a measurement are contained within the measurement description and the referenced objects. Therefore, if the measurement description is identical to a measurement description of a previous measurement, the result of the previous measurement can be reused.

The cache mechanism works using a hash function on the XML representation of the measurement description. After a measurement has been performed, a file named after the hash value of the measurement description of the measurement is created and the measurement results are stored therein. Before a measurement is performed, the hash value is computed. If a corresponding file is found, the previous measurement results are reused.

4.7 Measuring Core

The measuring core is based on the object graph constructed by the driver. The classes are extended with code and data.

4.7.1 Core Architecture

To fully utilize multi core systems, applications have to be implemented with multiple threads or processes. Measuring such applications is considerably more difficult than measuring single threaded applications. If the thread management is implemented specifically for the measurement at hand, the measuring code can be weaved into it by hand. But if the threads or processes are created within legacy or closed source code, the measuring tool has to take care of detecting the creation of threads and install the necessary measurers. For the roofline measuring tool we will only consider multi threaded applications.

To gain full control over the kernel code, the measurement tool starts the kernel within a child process. The parent process attaches to the child using ptrace. This causes the child to be stopped when certain events occur and the parent is notified. The events include thread creation and breakpoints.

Every kernel thread can raise events at any point. The events include thread creation, breakpoints, starting and stopping of workloads etc. Whenever an event occurs, the rule list has to be searched and the matching actions have to be executed. The rule list is always searched in the thread which raised the event. If the event caused the thread to be stopped and the parent thread to be notified, the parent restarts the thread with a notification of the event which occurred. The thread will search the rule list and continue execution.

Actions can be associated with a thread. In this case, the action is always executed within that thread. Measurers can be associated with a thread, too. In that case, all functions of the measurer have to be called within the specified thread. It is the responsibility of the caller to respect such an association. If necessary, an intermediate action has to be used.

The technically most challenging is to interrupt another thread and make it execute some action. There are two approaches to this problem, either using a signal handler of the thread or using ptrace to call code within the thread. Since installing a signal handler in a thread could cause unwanted side effects, we use the ptrace approach. SIGTRAP is sent to the target thread, which will cause it to be stopped. The parent modifies the thread state of the stopped thread. When the thread resumes execution, it executes some event handling code and then return to the location the thread was interrupted.

4.7.2 Child Thread Lifetime

The parent process manages a thread table, which contains the state of each child thread. When a child thread is cloned, the parent receives a signal by ptrace. It will create a new entry in the thread table and set the state to "new". In this state, the child cannot receive messages yet. When the thread is started, the parent process will receive a SIGSTOP from the child. Upon receiving this signal from a thread in the new state, the parent will send the "process" command and mark the thread as processing. As soon as the child is done with the initialization, it will send "done" to the parent. The process is put into the running state. When the child receives a message, the parent will possibly send process to it again. If it exits, it will go the the exit state.

4.7.3 Building

Each measurement can be performed with different compiler optimization flags and macro definitions. Therefore, the measuring core has to be rebuilt for each measurement, which makes rebuilding the measuring core a frequent operation. It should therefore be as fast as possible. This is achieved by carefully tracking all build dependencies and by using ccache.

CCache is a compiler cache. Whenever the compiler is run, ccache hashes all input files, together with the compiler flags. It then checks if it's cache already contains an entry for the hash value. If this is not the case, ccache runs the compiler and stores the output together with the hash value of the input in it's cache. If the hash value is present already, it does not run the compiler but uses the compiler output stored in it's cache. This considerably speeds up recompilations.

But ccache still has to build the hash values and copy the compiler output, which takes some time. This is where tracking the build dependencies comes in. The following parameters can change between measurements:

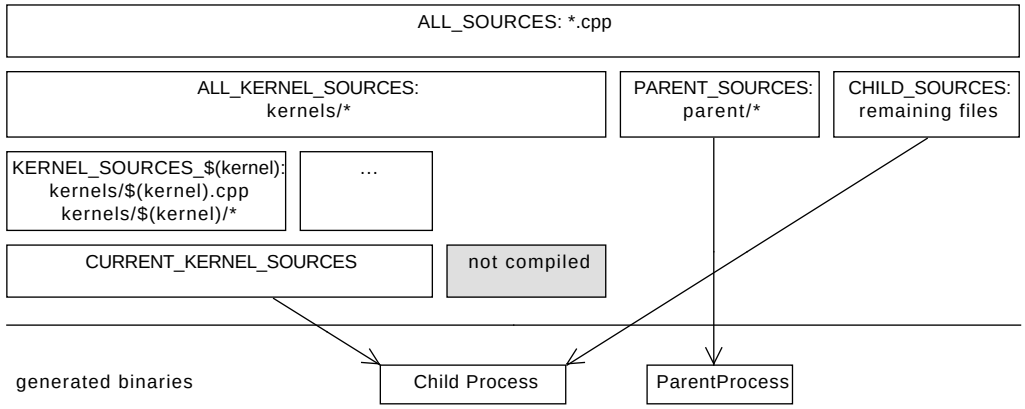
Macro definitions Each macro definition is stored in a separate file, which is only updated by the measurement driver if the macro definition changes. Every source file which needs a macro definition includes the corresponding file. These inclusions are tracked and allow to only recompile the affected source files.

Compiler flags Each kernel specifies it's own optimization flags. The compiler flags used for the rest of the measuring core do not affect the measurement results. The compiler flags are stored in a separate file. Whenever it is changed, the parts affected are recompiled.

Compiled Kernels For each measurement, only the kernels actually used are compiled. The measurement driver writes the kernel names into a separate file. The child binary is recompiled when it changes.

The build process is controlled using the gnu make utility [3]. Make automatically determines which parts of a program have to be recompiled, based on rules stored in a makefile. Each rule consists of target files, prerequisite files and a recipe. Make checks the modification times of the target and the prerequisite files. If any prerequisite is newer than any target, the recipe is executed in order to update the target files. The recipe is a sequence of shell commands.

The following diagram shows how the source files are categorized and compiled:



The makefile used for the measuring core first instructs make to use the find utility to get a list of all source files (with .cpp extension) in the measuringCore/src and measuringCore/generated directories (ALL_SOURCES).

Using the filter functions of make, the kernel sources are set to the subset of ALL_SOURCES which is located in src/kernels or generated/kernels. These are all sources related to kernels. (ALL_KERNEL_SOURCES).

The sources of the parent process are the subset of ALL_SOURCES which is located in src/parent(PARENT_SOURCES).

The source files of ALL_SOURCES not contained in ALL_KERNEL_SOURCES or PARENT_SOURCES are stored in CHILD_SOURCES.

The names of all present kernels are stored by the measurement driver in generated/kernelNames.mk. For each of the kernels named there, the source file named after the kernel and all source files in the subdirectory named after the kernel collected in a variable (KERNEL_SOURCES_$(kernel)). They are

compiled with the optimization flags of the compiler, wich are stored under generated/kernelOptimization.

The sources of all current sources are collected and form, together with the CHILD_SOURCES, the sources of the child process.

The parent process is compiled from the PARENT_SOURCES.

There is a rule without recipe with the kernel objects as target and the file containing the measurement specific optimization flags as prerequisite. This causes the file containing the optimization flags to be added as prerequisite for each kernel object file.

In the C programming language, it is possible to include other files in a source file. Of course, the compiled code depends on the contents of the included files, too. To track these build dependencies, the compiler is instructed to generate rules without recipes with the object file as target and the source file together with the included files as prerequisites. The generated rules are stored in .d files in the build directory and are included in the makefile.

If special compilation flags are required for a source file, a rule should be added near the end of the makefile.

4.7.4 System Initialization

We chose a modular approach to initialize the measuring core. Whenever a system part needs to run code when the program starts or shuts down, it can instantiate a class derived from SystemInitializer.

This is preferably achieved by declaring a global static variable named dummy in a .cpp file. Example:

```
// define and register a system initializer.
static class FooInitializer: public SystemInitializer{
  void start(){
    // code to be executed on startup
  }

  void stop(){
    // code to be executed on shutdown
  }
} dummy;
```

It is important to give every initializer subclass an individual name. Use the name of the file the initializer is declared in as prefix. If two initializer classes have the same name, they don't work correctly (instances of the wrong classes are created)

Whenever a SystemInitializer is instantiated, the instance is registered in a static global list. On system startup and shutdown, the start() respective stop() method of all registered SystemInitializers is called.

5 Measuring Execution Time

- 0x40000000 UNHALTED_CORE_CYCLES 30Ah CPU_CLK_UNHALTED.CORE
- 0x40000025 UNHALTED_REFERENCE_CYCLES 30Bh CPU_CLK_UNHALTED.REF
- 0x400000ad THERMAL_TRIP 3Bh C0h
- 0x400000ae CPU_CLK_UNHALTED
- 0x400001a9 ix86arch::UNHALTED_CORE_CYCLES
- 0x400001aa ix86arch::INSTRUCTION_RETIRED
- 0x400001ab ix86arch::UNHALTED_REFERENCE_CYCLES

5.1 Experiments

To gain insight into the accuracy and precision of the timers available, we run the addition kernel with an exponentially increasing iteration count. The largest iteration count should result in a runtime of about 0.5s. We expect the precision to be high for small iteration counts, a peak variance at around 5 ms due to the task switches, and an increasing precision for higher iteration counts. The accuracy of the minimum should be high for small iteration counts, and include a small overhead if multiple task switches occur during the measurement.

5.2 Experiment 1

The goal of this experiment is to show the effects of task switches and disk IO on measuring execution time. As far as possible, we'd like to exclude the effects of shared memory bandwidth.

We measure the arithmetic add kernel. Iteration count is increased until total execution time is about half a second. We measure at both the minimal and maximal CPU frequency. We measure on an idle system, when another arithmetic kernel runs or when heavy disk IO is performed. We let the system

load threads run on all of the system's CPU. Separate measurements for core cycles, bus cycles, gettimeofday(), times (user time)

We estimate the execution time for one iteration. We take the time for the highest iteration count where we can still get measurement runs without any interrupts. This time is divided by the respective iteration count.

Graphs (idle, disk, arithmetic):

- X: exp exec time [ms, cycles] Y: rel error V: box plots
- X: exp exec time [ms, cycles] Y: ints V: box plots
- X: exp exec time [ms, cycles] Y: task sw V: box plots

5.3 Experiment 2

Investigate the effects of task switches on a memory intensive kernel. Repeat experiment 1 with both a memory load and write kernel

5.4 Experiment 3

Measure the influences of shared memory bandwidth on the runtime. Depending on the system architecture, we expect to measure shared bandwidth effects. We measure a memory load and a memory write kernel. We measure on idle system and while running memory load and write kernel on the other CPUs. Buffer sizes are larger than cache size. Measure core cycles. Measure at min and max frequency. Buffer size chosen such that execution time is around half a scheduling time segment.

One plot per kernel combination (load/load, load/write, write/load, write/write)

- X: cpu map Y: time[cycles]

5.5 Experiment 4

Unfortunately, even on a very lightly loaded system, any task can be executed on any other core. Caused by the shared memory bandwidth, this could affect short running kernels, and result in wrong measurement results of a whole measurement series. We show this by starting two memory intensive, short running kernels simultaneously. Solution: introduce sleeps between measurement runs, to make sure they are independently measured.

with and without sleep: X: measurement number Y: time[cycles]

5.6 Experiment 5

For long running measurements, it is sufficient to make sure that the average system load is low. The side effects of code running on other kernels will be smoothed out. This is shown on various average loads (none, 1 2 3 5 10 20 40 50 75 100)

X: system load Y: rel error X: system load Y: variance

We measure using the following counters: - core cycles - bus cycles - gettimeofday() - times(): user time

We measure the following kernels: +- arithmetic:add *- mem: load *- mem: write

We measure at *+- max frequency +- min frequency

We measure use the following kernels to generate system load +- arithmetic: add *- mem: load *- mem: write +- io: file - io: network - io: graphics

We measure at different load levels (*+none, 25, 50, 75, *+100)

Measure at different load switch durations (0.1,1,5,10,15,50,100 ms)

We measure with any core combination. same core / +core2 core *+0 *+1 *+2 *+3

6 Measuring Memory Traffic

- 0x40000027 LLC_MISSES
- 0x40000040 SSE_PRE_EXEC 07h03h SSE_PRE_EXEC.L2
- 0x40000059 L2_DBUS_BUSY_RD
- 0x4000005c L2_LINES_IN 24h
- 0x4000006a L2_M_LINES_OUT 27h
- 0x400000cb SSE_PRE_MISS 4Bh 00h/01h/02h
- 62h BUS_DRDY_CLOCKS

6.1 Experiments

To gain insight into the accuracy and precision of the memory transfer methods, we run the addition kernel with an exponentially increasing iteration count. The largest iteration count should result in a runtime of about 0.5s. We expect the precision to be high for small iteration counts, a peak variance at around 5 ms due to the task switches, and an increasing precision for higher iteration counts. The accuracy of the minimum should be high for

small iteration counts, and include a small overhead if multiple task switches occur during the measurement.

6.2 Experiment 1

The first series of measurements is about the overhead of task switches on the measured memory transfer. We measure an arithmetic kernel and mem load and write, with buffer sizes which fit into the cache and which do not fit. We measure on an idle system, and with an arithmetic kernel on the other CPUs.

X: iteration count Y: rel error X: iteration count Y: ints X: iteration count Y: task switches

6.3 Experiment 2

The second measurement series is about the separation of measuring transfer volume of different threads. We measure mem load and write with large buffers. We measure on an idle system and with mem kernels on the other CPUs. Execution time: about .5 s

X: cpu map Y: rel error

We measure using the following counters: - bus transactions - cache misses / writebacks / non-prefetching load/store instructions

We measure the following kernels: - arithmetic:add - mem: load: small - mem: write: small - mem: load: large - mem: write: large

We measure at - max frequency

We measure use the following kernels to generate system load - arithmetic: add - mem: load - mem: write - io: file - io: network - io: graphics

We measure at different load levels (none, 25, 50, 75, 100)

Measure at different load switch durations (0.1,1,5,10,15,50,100 ms)

We measure with any core combination. same core / +core2 core 0 1 2 3

7 Mesuring Operation Count

- 0x40000156 SIMD_UOP_TYPE_EXEC B3h 01h/20h

List of Figures

References

[1] R. Bryant and D. O'Hallaron. *Computer systems: a programmer's perspective*. Prentice Hall, 2011. 4

[2] E. Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley, 2004. 24

[3] P. D. S. Richard M. Stallman, Roland McGrath. Gnu make version 3.82. Technical report, Free Software Foundation, July 2010. 37

[4] R. C. Whaley and A. M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. *Softw. Pract. Exper.*, 38(15):1621–1642, Dec. 2008. 5

[5] S. W. Williams, A. Waterman, and D. A. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical Report UCB/EECS-2008-134, EECS Department, University of California, Berkeley, Oct 2008. 2