OSACA

Release 0.3.2

Oct 20, 2020

Contents:


OSACA

CHAPTER 1

OSACA

1.1 Open Source Architecture Code Analyzer

For an innermost loop kernel in assembly, this tool allows automatic instruction fetching of assembly code and automatic runtime prediction including throughput analysis and detection for critical path and loop-carried dependencies.

CHAPTER 2

Getting started

2.1 Installation

On most systems with python pip and setuputils installed, just run:

```
pip install --user osaca
```

for the latest release.

To build OSACA from source, clone this repository using `git clone https://github.com/RRZE-HPC/OSACA` and run in the root directory:

```
python ./setup.py install
```

After installation, OSACA can be started with the command `osaca` in the CLI.

2.2 Dependencies:

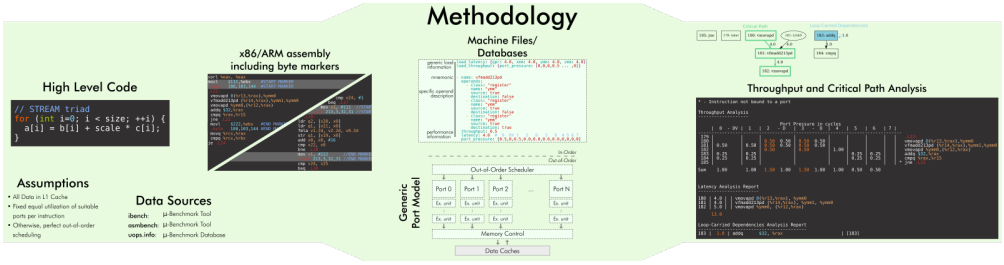
Additional requirements are:

- Python3
- Graphviz for dependency graph creation (minimal dependency is *libgraphviz-dev* on Ubuntu)
- Kerncraft >=v0.8.4 for marker insertion
- ibench or asmbench for throughput/latency measurements

CHAPTER 3

Design

A schematic design of OSACA's workflow is shown below:



CHAPTER 4

Usage

The usage of OSACA can be listed as:

```
osaca [-h] [-V] [--arch ARCH] [--fixed] [--db-check]
      [--import MICROBENCH] [--insert-marker]
      [--export-graph GRAPHNAME] [--ignore-unknown] [--verbose]
      FILEPATH
```

-h, --help prints out the help message.

-V, --version shows the program's version number.

--arch ARCH needs to be replaced with the target architecture abbreviation. Possible options are `SNB`, `IVB`, `HSW`, `BDW`, `SKX` and `CSX` for the latest Intel micro architectures starting from Intel Sandy Bridge and `ZEN1`, `ZEN2` for AMD Zen architectures. Furthermore, `TX2` for Marvell's ARM-based ThunderX2 architecture is available.

--fixed Run the throughput analysis with fixed port utilization for all suitable ports per instruction. Otherwise, OSACA will print out the optimal port utilization for the kernel.

--db-check Run a sanity check on the by "–arch" specified database. The output depends on the verbosity level. Keep in mind you have to provide an existing (dummy) filename in anyway.

--import MICROBENCH Import a given microbenchmark output file into the corresponding architecture instruction database. Define the type of microbenchmark either as "ibench" or "asmbench".

--insert-marker OSACA calls the Kerncraft module for the interactively insertion of IACA byte markers or OSACA AArch64 byte markers in suggested assembly blocks.

--export-graph EXPORT_PATH Output path for .dot file export. If "." is given, the file will be stored as "./osaca_dg.dot". After the file was created, you can convert it to a PDF file using dot.

--ignore-unknown	Force OSACA to apply a throughput and latency of 0.0 cy for all unknown instruction forms. If not specified, a warning will be printed instead if one ore more isntruction form is unknown to OSACA.
-v, --verbose	Increases verbosity level

The **FILEPATH** describes the filepath to the file to work with and is always necessary

---

Hereinafter OSACA's scope of function will be described.

4.1 Throughput & Latency analysis

As main functionality of OSACA, the tool starts the analysis on a marked assembly file by running the following command with one or more of the optional parameters:

```
osaca --arch ARCH [--fixed] [--ignore-unknown]
                  [--export-graph EXPORT_PATH]
      file
```

The `file` parameter specifies the target assembly file and is always mandatory.

The parameter `ARCH` is positional for the analysis and must be replaced by the target architecture abbreviation.

OSACA assumes an optimal scheduling for all instructions and assumes the processor to be able to schedule instructions in a way that it achieves a minimal reciprocal throughput. However, in older versions (<=v0.2.2) of OSACA, a fixed probability for port utilization was assumed. This means, instructions with *N* available ports for execution were scheduled with a probability of *1/N* to each of the ports. This behavior can be enforced by using the `--fixed` flag.

If one or more instruction forms are unknown to OSACA, it refuses to print an overall throughput, CP and LCD analysis and marks all unknown instruction forms with `X` next to the mnemonic. This is done so the user does not miss out on this unrecognized instruction and might assume an incorrect runtime prediction. To force OSACA to apply a throughput and latency of 0.0 cy for all unknown instruction forms, the flag `--ignore-unknown` can be specified.

To get a visualization of the analyzed kernel and its dependency chains, OSACA provides the option to additionally produce a graph as DOT file, which represents the kernel and all register dependencies inside of it. The tool highlights all LCDs and the CP. The graph generation is done by running OSACA with the `--export-graph EXPORT_GRAPH` flag. OSACA stores the DOT file either at the by `EXPORT_GRAPH` specified filepath or uses the default filename "osaca_dg.dot" in the current working directory. Subsequently, the DOT-graph can be adjusted in its appearance and converted to various output formats such as PDF, SVG, or PNG using the dot command, e.g., `dot -Tpdf osaca_dg.dot -o graph.pdf` to generate a PDF document.

4.2 Marker insertion

For extracting the right kernel, one has to mark it in beforehand. Currently, only the detection of markers in the assembly code and therefore the analysis of assembly files is supported by OSACA.

Marking a kernel means to insert the byte markers in the assembly file in before and after the loop. For this, the start marker has to be inserted right in front of the loop label and the end marker directly after the jump instruction. IACA requires byte markers since it operates on opcode-level. To provide a trade-off between reusability for such tool and convenient usability, OSACA supports both byte markers and comment line markers. While the byte markers for x86 are equivalent to IACA byte markers, the comment keywords `OSACA-BEGIN` and `OSACA-END` are based on LLVM-MCA's markers.

4.2.1 x86 markers

Byte markers

```
  movl    $111,%ebx       #IACA/OSACA START MARKER
  .byte   100,103,144     #IACA/OSACA START MARKER
.loop:
  # loop body
  jb      .loop
  movl    $222,%ebx       #IACA/OSACA END MARKER
  .byte   100,103,144     #IACA/OSACA END MARKER
```

Comment line markers

```
  # OSACA-BEGIN
.loop:
  # loop body
  jb      .loop
  # OSACA-END
```

4.2.2 AArch64 markers

Byte markers

```
  mov      x1, #111         // OSACA START
  .byte    213,3,32,31      // OSACA START
.loop:
  // loop body
  b.ne     .loop
  mov      x1, #222         // OSACA END
  .byte    213,3,32,31      // OSACA END
```

Comment line markers

```
  // OSACA-BEGIN
.loop:
  // loop body
  b.ne     .loop
  // OSACA-END
```

OSACA in combination with Kerncraft provides a functionality for the automatic detection of possible loop kernels and inserting markers. This can be done by using the `--insert-marker` flag together with the path to the target assembly file and the target architecture.

4.3 Benchmark import

OSACA supports the automatic integration of new instruction forms by parsing the output of the micro- benchmark tools asmbench and ibench. This can be achieved by running OSACA with the command line option `--import MICROBENCH`:

```
osaca --arch ARCH --import MICROBENCH file
```

`MICROBENCH` specifies one of the currently supported benchmark tools, i.e., "asmbench" or "ibench". `ARCH` defines the abbreviation of the target architecture for which the instructions will be added and file must be the path to the

generated output file of the benchmark. The format of this file has to match either the basic command line output of ibench, e.g.,

```
[INSTRUCTION FORM]-TP:     0.500 (clock cycles)     [DEBUG - result: 1.000000]
[INSTRUCTION FORM]-LT:     4.000 (clock cycles)     [DEBUG - result: 1.000000]
```

or the command line output of asmbench including the name of the instruction form in a separate line at the beginning, e.g.:

```
[INSTRUCTION FORM]
Latency: 4.00 cycle
Throughput: 0.50 cycle
```

Note that there must be an empty line after each throughput measurement as part of the output so that one instruction form entry consists of four (4) lines.

To let OSACA import the instruction form with the correct operands, the naming conventions for the instruction form name must be followed:

- The first part of the name is the mnemonic and ends with the character "–" (not part of the mnemonic in the DB).
- The second part of the name are the operands. Each operand must be separated from another operand by the character "_".
- For each **x86** operand, one of the following symbols must be used:
  - "r" for general purpose registers (rax, edi, r9, ...)
  - "x", "y", or "z" for xmm, ymm, or zmm registers, respectively
  - "i" for immediates
  - "m" for a memory address. Add "b" if the memory address contains a base register, "o" if it contains an offset, "i" if it contains an index register, and "s" if the index register additionally has a scale factor of *more* than 1.
- For each **AArch64** operand, one of the following symbols must be used:
  - "w", "x", "b", "h", "s", "d", or "q" for registers with the corresponding prefix.
  - "v" followed by a single character ("b", "h", "s", or "d") for vector registers with the corresponding lane width of the second character. If no second character is given, OSACA assumes a lane width of 64 bit (d) as default.
  - "i" for immediates
  - "m" for a memory address. Add "b" if the memory address contains a base register, "o" if it contains an offset, "i" if it contains an index register, and "s" if the index register additionally has a scale factor of *more* than 1. Add "r" if the address format uses pre-indexing and "p" if it uses post-indexing.

Valid instruction form examples for x86 are `vaddpd-x_x_x`, `mov-r_mboi`, and `vfmadd213pd-mbis_y_y`.

Valid instruction form examples for AArch64 are `fadd-vd_vd_v`, `ldp-d_d_mo`, and `fmov-s_i`.

Note that the options to define operands are limited, therefore, one might need to adjust the instruction forms in the architecture DB after importing. OSACA parses the output for an arbitrary number of instruction forms and adds them as entries to the architecture DB. The user must edit the ISA DB in case the instruction form shows irregular source and destination operands for its ISA syntax. OSACA applies the following rules by default:

- If there is only one operand, it is considered as source operand
- In case of multiple operands the target operand (depending on the ISA syntax the last or first one) is considered to be the destination operand, all others are considered as source operands.

4.4 Database check

Since a manual adjustment of the ISA DB is currently indispensable when adding new instruction forms, OSACA provides a database sanity check using the –db-check flag. It can be executed via:

```
osaca --arch ARCH --db-check [-v] file
```

`ARCH` defines the abbreviation of the target architecture of the database to check. The `file` argument needs to be specified as it is positional but may be any existing dummy path. When called, OSACA prints a summary of database information containing the amount of missing throughput values, latency values or μ-ops assignments for an instruction form. Furthermore, it shows the amount of duplicate instruction forms in both the architecture DB and the ISA DB and checks how many instruction forms in the ISA DB are non-existent in the architecture DB. Finally, it checks via simple heuristics how many of the instruction forms contained in the architecture DB might miss an ISA DB entry. Running the database check including the `-v` verbosity flag, OSACA prints in addition the specific name of the identified instruction forms so that the user can check the mentioned incidents.

CHAPTER 5

Examples

For clarifying the functionality of OSACA a sample kernel is analyzed for an Intel CSX core hereafter:

```
double a[N], double b[N];
double s;

// loop
for(int i = 0; i < N; ++i)
    a[i] = s * b[i];
```

The code shows a simple scalar multiplication of a vector `b` and a floating-point number `s`. The result is written in vector `a`. After including the OSACA byte marker into the assembly, one can start the analysis typing

```
osaca --arch CSX PATH/TO/FILE
```

in the command line.

The output is:

```
Open Source Architecture Code Analyzer (OSACA) - v0.3
Analyzed file:      scale.s.csx.O3.s
Architecture:       csx
Timestamp:          2019-10-03 23:36:21

 P - Throughput of LOAD operation can be hidden behind a past or future STORE␣
↪instruction
 * - Instruction micro-ops not bound to a port
 X - No throughput/latency information for this instruction in data file


    Combined Analysis Report
    ------------------------
                                        Port pressure in cycles
         |  0   - 0DV  |  1   |  2   -  2D  |  3   -  3D  |  4   |  5   |  6   |  7  ␣
↪||  CP   | LCD  |
    ---------------------------------------------------------------------------------
↪---------------
```

(continues on next page)

(continued from previous page)

```
     170 |             |      |             |             |      |      |      |
↪||      |      |   .L22:
     171 | 0.50        | 0.50 | 0.50   0.50 | 0.50   0.50 |      |      |      |
↪||  8.0 |      |   vmulpd    (%r12,%rax), %ymm1, %ymm0
     172 |             |      | 0.50        | 0.50        | 1.00 |      |      |
↪||  5.0 |      |   vmovapd   %ymm0, 0(%r13,%rax)
     173 | 0.25        | 0.25 |             |             |      | 0.25 | 0.25 |
↪||      |  1.0 |   addq      $32, %rax
     174 | 0.00        | 0.00 |             |             |      | 0.50 | 0.50 |
↪||      |      |   cmpq      %rax, %r14
     175 |             |      |             |             |      |      |      |
↪||      |      | * jne       .L22

           0.75          0.75   1.00   0.50   1.00   0.50   1.00   0.75   0.75
↪   13.0   1.0


    Loop-Carried Dependencies Analysis Report
    -----------------------------------------
     173 |  1.0 | addq      $32, %rax                     | [173]
```

It shows the whole kernel together with the optimized port pressure of each instruction form and the overall port binding. Furthermore, in the two columns on the right, the critical path (CP) and the longest loop-carried dependency (LCD) of the loop kernel. In the bottom, all loop-carried dependencies are shown, each with a list of line numbers being part of this dependency chain on the right.

You can find more (already marked) examples and sample outputs for various architectures in the examples directory.

CHAPTER 6

Credits

Implementation: Jan Laukemann

CHAPTER 7

License

AGPL-3.0

CHAPTER 8

API Reference

8.1 osaca package

8.1.1 Subpackages

osaca.api package

Provides interfaces to other tools.

osaca.api.kerncraft_interface module

class Capturing
Bases: list

class KerncraftAPI(*arch*, *code*)
Bases: object

create_output(*verbose=False*)

get_unmatched_instruction_ratio()

get_port_occupation_cycles()

get_total_throughput()

get_latency()

get_cp()

get_lcd()

Module contents

APIs for handling interfaces to kerncraft, etc.

Only the classes below will be exported, so please add new semantic tools to __all__.

class KerncraftAPI(*arch*, *code*)
Bases: object

create_output(*verbose=False*)

get_unmatched_instruction_ratio()

get_port_occupation_cycles()

get_total_throughput()

get_latency()

get_cp()

get_lcd()

osaca.parser package

Parser module for parsing the assembly code.

osaca.parser.attr_dict module

Attribute Dictionary to access dictionary entries as attributes.

class AttrDict(**args*, ***kwargs*)
Bases: dict

static convert_dict(*dictionary*)
Convert given dictionary to *AttrDict*.

Parameters **dictionary** (*dict*) – *dict* to be converted

Returns *AttrDict* representation of dictionary

osaca.parser.base_parser module

Parser superclass of specific parsers.

class BaseParser
Bases: object

COMMENT_ID = 'comment'

DIRECTIVE_ID = 'directive'

IMMEDIATE_ID = 'immediate'

LABEL_ID = 'label'

MEMORY_ID = 'memory'

REGISTER_ID = 'register'

SEGMENT_EXT_ID = 'segment_extension'

INSTRUCTION_ID = 'instruction'

OPERANDS_ID = 'operands'

parse_file(*file_content*, *start_line=0*)
Parse assembly file. This includes *not* extracting of the marked kernel and the parsing of the instruction forms.

Parameters

- **file_content** (*str*) – assembly code
- **start_line** (*int*) – offset, if first line in file_content is meant to be not 1

Returns list of instruction forms

parse_line(*line*, *line_number=None*)

parse_instruction(*instruction*)

parse_register(*register_string*)

is_gpr(*register*)

is_vector_register(*register*)

get_reg_type(*register*)

construct_parser()

process_operand(*operand*)

get_full_reg_name(*register*)

normalize_imd(*imd*)

is_reg_dependend_of(*reg_a*, *reg_b*)

osaca.parser.parser_AArch64v81 module

class ParserAArch64v81
Bases: *osaca.parser.base_parser.BaseParser*

construct_parser()
Create parser for ARM AArch64 ISA.

parse_line(*line*, *line_number=None*)
Parse line and return instruction form.

Parameters

- **line** (*str*) – line of assembly code
- **line_number** (*int, optional*) – identifier of instruction form, defautls to None

Returns *dict* – parsed asm line (comment, label, directive or instruction form)

parse_instruction(*instruction*)
Parse instruction in asm line.

Parameters **instruction** (*str*) – Assembly line string.

Returns *dict* – parsed instruction form

process_operand(*operand*)
Post-process operand

process_memory_address(*memory_address*)
Post-process memory address operand

process_sp_register(*register*)
Post-process stack pointer register

process_register_list(*register_list*)
Post-process register lists (e.g., {r0,r3,r5}) and register ranges (e.g., {r0-r7})

process_immediate(*immediate*)
Post-process immediate operand

process_label(*label*)
Post-process label asm line

get_full_reg_name(*register*)
Return one register name string including all attributes

normalize_imd(*imd*)
Normalize immediate to decimal based representation

ieee_to_float(*ieee_val*)
Convert IEEE representation to python float

parse_register(*register_string*)

is_gpr(*register*)
Check if register is a general purpose register

is_vector_register(*register*)
Check if register is a vector register

is_flag_dependend_of(*flag_a*, *flag_b*)
Check if `flag_a` is dependent on `flag_b`

is_reg_dependend_of(*reg_a*, *reg_b*)
Check if `reg_a` is dependent on `reg_b`

get_reg_type(*register*)
Get register type

osaca.parser.parser_x86att module

class ParserX86ATT
Bases: *osaca.parser.base_parser.BaseParser*

construct_parser()
Create parser for ARM AArch64 ISA.

parse_register(*register_string*)
Parse register string

parse_line(*line*, *line_number=None*)
Parse line and return instruction form.

Parameters

- **line** (*str*) – line of assembly code
- **line_number** (*int, optional*) – default None, identifier of instruction form

Returns `dict` – parsed asm line (comment, label, directive or instruction form)

parse_instruction(*instruction*)
Parse instruction in asm line.

Parameters **instruction** (*str*) – Assembly line string.

Returns *dict* – parsed instruction form

process_operand(*operand*)
Post-process operand

process_memory_address(*memory_address*)
Post-process memory address operand

process_label(*label*)
Post-process label asm line

process_immediate(*immediate*)
Post-process immediate operand

get_full_reg_name(*register*)
Return one register name string including all attributes

normalize_imd(*imd*)
Normalize immediate to decimal based representation

is_flag_dependend_of(*flag_a*, *flag_b*)
Check if `flag_a` is dependent on `flag_b`

is_reg_dependend_of(*reg_a*, *reg_b*)
Check if `reg_a` is dependent on `reg_b`

is_basic_gpr(*register*)
Check if register is a basic general purpose register (ebi, rax, . . .)

is_gpr(*register*)
Check if register is a general purpose register

is_vector_register(*register*)
Check if register is a vector register

get_reg_type(*register*)
Ger register type

Module contents

Collection of parsers supported by OSACA.

Only the parser below will be exported, so please add new parsers to __all__.

get_parser(*isa*)

class AttrDict(**args*, ***kwargs*)
Bases: `dict`

static convert_dict(*dictionary*)
Convert given dictionary to *AttrDict*.

Parameters **dictionary** (*dict*) – *dict* to be converted

Returns *AttrDict* representation of `dictionary`

class BaseParser
Bases: `object`

COMMENT_ID = 'comment'

DIRECTIVE_ID = 'directive'

IMMEDIATE_ID = 'immediate'

LABEL_ID = 'label'

MEMORY_ID = 'memory'

REGISTER_ID = 'register'

SEGMENT_EXT_ID = 'segment_extension'

INSTRUCTION_ID = 'instruction'

OPERANDS_ID = 'operands'

parse_file(*file_content*, *start_line=0*)
Parse assembly file. This includes *not* extracting of the marked kernel and the parsing of the instruction forms.

Parameters

- **file_content** (*str*) – assembly code
- **start_line** (*int*) – offset, if first line in file_content is meant to be not 1

Returns list of instruction forms

parse_line(*line*, *line_number=None*)

parse_instruction(*instruction*)

parse_register(*register_string*)

is_gpr(*register*)

is_vector_register(*register*)

get_reg_type(*register*)

construct_parser()

process_operand(*operand*)

get_full_reg_name(*register*)

normalize_imd(*imd*)

is_reg_dependend_of(*reg_a*, *reg_b*)

class ParserX86ATT
Bases: *osaca.parser.base_parser.BaseParser*

construct_parser()
Create parser for ARM AArch64 ISA.

parse_register(*register_string*)
Parse register string

parse_line(*line*, *line_number=None*)
Parse line and return instruction form.

Parameters

- **line** (*str*) – line of assembly code
- **line_number** (*int, optional*) – default None, identifier of instruction form

Returns `dict` – parsed asm line (comment, label, directive or instruction form)

parse_instruction(*instruction*)
Parse instruction in asm line.

Parameters **instruction** (*str*) – Assembly line string.

Returns *dict* – parsed instruction form

process_operand(*operand*)
Post-process operand

process_memory_address(*memory_address*)
Post-process memory address operand

process_label(*label*)
Post-process label asm line

process_immediate(*immediate*)
Post-process immediate operand

get_full_reg_name(*register*)
Return one register name string including all attributes

normalize_imd(*imd*)
Normalize immediate to decimal based representation

is_flag_dependend_of(*flag_a*, *flag_b*)
Check if `flag_a` is dependent on `flag_b`

is_reg_dependend_of(*reg_a*, *reg_b*)
Check if `reg_a` is dependent on `reg_b`

is_basic_gpr(*register*)
Check if register is a basic general purpose register (ebi, rax, ...)

is_gpr(*register*)
Check if register is a general purpose register

is_vector_register(*register*)
Check if register is a vector register

get_reg_type(*register*)
Ger register type

class ParserAArch64v81
Bases: *osaca.parser.base_parser.BaseParser*

construct_parser()
Create parser for ARM AArch64 ISA.

parse_line(*line*, *line_number=None*)
Parse line and return instruction form.

Parameters

- **line** (*str*) – line of assembly code
- **line_number** (*int, optional*) – identifier of instruction form, defautls to None

Returns *dict* – parsed asm line (comment, label, directive or instruction form)

parse_instruction(*instruction*)
Parse instruction in asm line.

Parameters **instruction** (*str*) – Assembly line string.

Returns *dict* – parsed instruction form

process_operand(*operand*)
Post-process operand

process_memory_address(*memory_address*)
Post-process memory address operand

process_sp_register(*register*)
Post-process stack pointer register

process_register_list(*register_list*)
Post-process register lists (e.g., {r0,r3,r5}) and register ranges (e.g., {r0-r7})

process_immediate(*immediate*)
Post-process immediate operand

process_label(*label*)
Post-process label asm line

get_full_reg_name(*register*)
Return one register name string including all attributes

normalize_imd(*imd*)
Normalize immediate to decimal based representation

ieee_to_float(*ieee_val*)
Convert IEEE representation to python float

parse_register(*register_string*)

is_gpr(*register*)
Check if register is a general purpose register

is_vector_register(*register*)
Check if register is a vector register

is_flag_dependend_of(*flag_a*, *flag_b*)
Check if `flag_a` is dependent on `flag_b`

is_reg_dependend_of(*reg_a*, *reg_b*)
Check if `reg_a` is dependent on `reg_b`

get_reg_type(*register*)
Get register type

osaca.semantics package

Semantic part of OSACA.

osaca.semantics.arch_semantics module

Semantics opbject responsible for architecture specific semantic operations

class ArchSemantics(*machine_model: osaca.semantics.hw_model.MachineModel*, *path_to_yaml=None*)
Bases: *osaca.semantics.isa_semantics.ISASemantics*

GAS_SUFFIXES = 'bswlqt'

add_semantics(*kernel*)
Applies performance data (throughput, latency, port pressure) and source/destination distribution to each instruction of a given kernel.

Parameters **kernel** (*list*) – kernel to apply semantics

assign_optimal_throughput(*kernel*)
Assign optimal throughput port pressure to a kernel. This is done in steps of 0.01cy.

Parameters **kernel** (*list*) – kernel to apply optimal port utilization

set_hidden_loads(*kernel*)
Hide loads behind stores if architecture supports hidden loads (depricated)

assign_tp_lt(*instruction_form*)
Assign throughput and latency to an instruction form.

substitute_mem_address(*operands*)
Create memory wildcard for all memory operands

convert_op_to_reg(*reg_type*, *reg_id='0'*)
Create register operand for a memory addressing operand

static get_throughput_sum(*kernel*)
Get the overall throughput sum separated by port of all instructions of a kernel.

osaca.semantics.hw_model module

class MachineModel(*arch=None*, *path_to_yaml=None*, *isa=None*, *lazy=False*)
Bases: object

WILDCARD = '*'

get_instruction(*name*, *operands*)
Find and return instruction data from name and operands.

get_instruction_from_dict(*name*, *operands*)
Find and return instruction data from name and operands stored in dictionary.

average_port_pressure(*port_pressure*)
Construct average port pressure list from instruction data.

set_instruction(*name*, *operands=None*, *latency=None*, *port_pressure=None*, *throughput=None*, *uops=None*)
Import instruction form information.

set_instruction_entry(*entry*)
Import instruction as entry object form information.

add_port(*port*)
Add port in port model of current machine model.

get_ISA()
Return ISA of *MachineModel*.

get_arch()
Return micro-architecture code of *MachineModel*.

get_ports()
Return port model of *MachineModel*.

has_hidden_loads()
Return if model has hidden loads.

get_load_latency(*reg_type*)
Return load latency for given register type.

get_load_throughput(*memory*)
Return load thorughput for given register type.

get_store_latency(*reg_type*)
Return store latency for given register type.

get_store_throughput(*memory*)
Return store throughput for given register type.

get_data_ports()
Return all data ports (i.e., ports with D-suffix) of current model.

static get_full_instruction_name(*instruction_form*)
Get one instruction name string including the mnemonic and all operands.

static get_isa_for_arch(*arch*)
Return ISA for given micro-arch `arch`.

dump(*stream=None*)
Dump machine model to stream or return it as a `str` if no stream is given.

osaca.semantics.isa_semantics module

class INSTR_FLAGS
Bases: `object`

Flags used for unknown or special instructions

LD = 'is_load_instruction'

TP_UNKWN = 'tp_unknown'

LT_UNKWN = 'lt_unknown'

NOT_BOUND = 'not_bound'

HIDDEN_LD = 'hidden_load'

HAS_LD = 'performs_load'

HAS_ST = 'performs_store'

class ISASemantics(*isa*, *path_to_yaml=None*)
Bases: `object`

GAS_SUFFIXES = 'bswlqt'

process(*instruction_forms*)
Process a list of instruction forms.

assign_src_dst(*instruction_form*)
Update instruction form dictionary with source, destination and flag information.

osaca.semantics.kernel_dg module

class KernelDG(*parsed_kernel*, *parser*, *hw_model: osaca.semantics.hw_model.MachineModel*)
Bases: `networkx.classes.digraph.DiGraph`

create_DG(*kernel*)
Create directed graph from given kernel

Parameters **kernel** – Parsed asm kernel with assigned semantic information

Returns DiGraph – directed graph object

check_for_loopcarried_dep(*kernel*)
Try to find loop-carried dependencies in given kernel.

Parameters **kernel** (*list*) – Parsed asm kernel with assigned semantic information

Returns *dict* – dependency dictionary with all cyclic LCDs

get_critical_path()
Find and return critical path after the creation of a directed graph.

get_loopcarried_dependencies()
Return all LCDs from kernel (after *check_for_loopcarried_dep()* was run)

find_depending(*instruction_form*, *kernel*, *include_write=False*, *flag_dependencies=False*)
Find instructions in kernel depending on a given instruction form.

Parameters

- **instruction_form** (*dict*) – instruction form to check for dependencies
- **kernel** (*list*) – kernel containing the instructions to check
- **include_write** (*boolean, optional*) – indicating if instruction ending the dependency chain should be included, defaults to *False*
- **flag_dependencies** (*boolean, optional*) – indicating if dependencies of flags should be considered, defaults to *False*

Returns iterator if all directly dependent instruction forms

get_dependent_instruction_forms(*instr_form=None*, *line_number=None*)
Returns iterator

is_read(*register*, *instruction_form*)
Check if instruction form reads from given register

is_written(*register*, *instruction_form*)
Check if instruction form writes in given register

export_graph(*filepath=None*)
Export graph with highlighted CP and LCDs as DOT file. Writes it to 'osaca_dg.dot' if no other path is given.

Parameters **filepath** (*str, optional*) – path to write DOT file, defaults to None.

osaca.semantics.marker_utils module

reduce_to_section(*kernel*, *isa*)
Finds OSACA markers in given kernel and returns marked section

Parameters

- **kernel** (*list*) – kernel to check
- **isa** (*str*) – ISA of given kernel

Returns *list* – marked section of kernel as list of instruction forms

find_marked_kernel_AArch64(*lines*)
Find marked section for AArch64

Parameters **lines** (*list*) – kernel

Returns *tuple of int* – start and end line of marked section

find_marked_kernel_x86ATT(*lines*)
Find marked section for x86

Parameters **lines** (*list*) – kernel

Returns *tuple of int* – start and end line of marked section

get_marker(*isa*, *comment=''*)
Return tuple of start and end marker lines.

find_marked_section(*lines*, *parser*, *mov_instr*, *mov_reg*, *mov_vals*, *nop_bytes*, *reverse=False*, *comments=None*)
Return indexes of marked section

Parameters

- **lines** (*list*) – kernel
- **parser** (BaseParser) – parser to use for checking
- **mov_instr** (*list of str*) – all MOV instruction possible for the marker
- **mov_reg** (*str*) – register used for the marker
- **mov_vals** (*list of int*) – values needed to be moved to mov_reg for valid marker
- **nop_bytes** (*list of int*) – bytes representing opcode of NOP
- **reverse** (*boolean, optional*) – indicating if ISA syntax requires reverse operand order, defaults to *False*
- **comments** (*dict, optional*) – dictionary with start and end markers in comment format, defaults to None

Returns *tuple of int* – start and end line of marked section

match_bytes(*lines*, *index*, *byte_list*)
Match bytes directives of markers

find_jump_labels(*lines*)
Find and return all labels which are followed by instructions until the next label

Returns OrderedDict of mapping from label name to associated line index

find_basic_blocks(*lines*)
Find and return basic blocks (asm sections which can only be executed as complete block).

Blocks always start at a label and end at the next jump/break possibility.

Returns OrderedDict with labels as keys and list of lines as value

find_basic_loop_bodies(*lines*)
Find and return basic loop bodies (asm section which loop back on itself with no other egress).

Returns OrderedDict with labels as keys and list of lines as value

Module contents

Tools for semantic analysis of parser result.

Only the classes below will be exported, so please add new semantic tools to __all__.

class MachineModel(*arch=None*, *path_to_yaml=None*, *isa=None*, *lazy=False*)
Bases: `object`

WILDCARD = '*'

get_instruction(*name*, *operands*)
Find and return instruction data from name and operands.

get_instruction_from_dict(*name*, *operands*)
Find and return instruction data from name and operands stored in dictionary.

average_port_pressure(*port_pressure*)
Construct average port pressure list from instruction data.

set_instruction(*name*, *operands=None*, *latency=None*, *port_pressure=None*, *throughput=None*, *uops=None*)
Import instruction form information.

set_instruction_entry(*entry*)
Import instruction as entry object form information.

add_port(*port*)
Add port in port model of current machine model.

get_ISA()
Return ISA of *MachineModel*.

get_arch()
Return micro-architecture code of *MachineModel*.

get_ports()
Return port model of *MachineModel*.

has_hidden_loads()
Return if model has hidden loads.

get_load_latency(*reg_type*)
Return load latency for given register type.

get_load_throughput(*memory*)
Return load thorughput for given register type.

get_store_latency(*reg_type*)
Return store latency for given register type.

get_store_throughput(*memory*)
Return store throughput for given register type.

get_data_ports()
Return all data ports (i.e., ports with D-suffix) of current model.

static get_full_instruction_name(*instruction_form*)
Get one instruction name string including the mnemonic and all operands.

static get_isa_for_arch(*arch*)
Return ISA for given micro-arch `arch`.

dump(*stream=None*)
Dump machine model to stream or return it as a `str` if no stream is given.

class KernelDG(*parsed_kernel*, *parser*, *hw_model: osaca.semantics.hw_model.MachineModel*)
Bases: `networkx.classes.digraph.DiGraph`

create_DG(*kernel*)
Create directed graph from given kernel

Parameters **kernel** – Parsed asm kernel with assigned semantic information

Returns `DiGraph` – directed graph object

check_for_loopcarried_dep(*kernel*)
Try to find loop-carried dependencies in given kernel.

Parameters **kernel** (*list*) – Parsed asm kernel with assigned semantic information

Returns *dict* – dependency dictionary with all cyclic LCDs

get_critical_path()
Find and return critical path after the creation of a directed graph.

get_loopcarried_dependencies()
Return all LCDs from kernel (after *check_for_loopcarried_dep()* was run)

find_depending(*instruction_form*, *kernel*, *include_write=False*, *flag_dependencies=False*)
Find instructions in kernel depending on a given instruction form.

Parameters

- **instruction_form** (*dict*) – instruction form to check for dependencies
- **kernel** (*list*) – kernel containing the instructions to check
- **include_write** (*boolean, optional*) – indicating if instruction ending the dependency chain should be included, defaults to *False*
- **flag_dependencies** (*boolean, optional*) – indicating if dependencies of flags should be considered, defaults to *False*

Returns iterator if all directly dependent instruction forms

get_dependent_instruction_forms(*instr_form=None*, *line_number=None*)
Returns iterator

is_read(*register*, *instruction_form*)
Check if instruction form reads from given register

is_written(*register*, *instruction_form*)
Check if instruction form writes in given register

export_graph(*filepath=None*)
Export graph with highlighted CP and LCDs as DOT file. Writes it to 'osaca_dg.dot' if no other path is given.

Parameters **filepath** (*str, optional*) – path to write DOT file, defaults to None.

reduce_to_section(*kernel*, *isa*)
Finds OSACA markers in given kernel and returns marked section

Parameters

- **kernel** (*list*) – kernel to check
- **isa** (*str*) – ISA of given kernel

Returns *list* – marked section of kernel as list of instruction forms

class ArchSemantics(*machine_model: osaca.semantics.hw_model.MachineModel*, *path_to_yaml=None*)

Bases: *osaca.semantics.isa_semantics.ISASemantics*

GAS_SUFFIXES = 'bswlqt'

add_semantics(*kernel*)

Applies performance data (throughput, latency, port pressure) and source/destination distribution to each instruction of a given kernel.

Parameters **kernel** (*list*) – kernel to apply semantics

assign_optimal_throughput(*kernel*)

Assign optimal throughput port pressure to a kernel. This is done in steps of `0.01cy`.

Parameters **kernel** (*list*) – kernel to apply optimal port utilization

set_hidden_loads(*kernel*)

Hide loads behind stores if architecture supports hidden loads (depricated)

assign_tp_lt(*instruction_form*)

Assign throughput and latency to an instruction form.

substitute_mem_address(*operands*)

Create memory wildcard for all memory operands

convert_op_to_reg(*reg_type*, *reg_id='0'*)

Create register operand for a memory addressing operand

static get_throughput_sum(*kernel*)

Get the overall throughput sum separated by port of all instructions of a kernel.

class ISASemantics(*isa*, *path_to_yaml=None*)

Bases: `object`

GAS_SUFFIXES = 'bswlqt'

process(*instruction_forms*)

Process a list of instruction forms.

assign_src_dst(*instruction_form*)

Update instruction form dictionary with source, destination and flag information.

class INSTR_FLAGS

Bases: `object`

Flags used for unknown or special instructions

LD = 'is_load_instruction'

TP_UNKWN = 'tp_unknown'

LT_UNKWN = 'lt_unknown'

NOT_BOUND = 'not_bound'

HIDDEN_LD = 'hidden_load'

HAS_LD = 'performs_load'

HAS_ST = 'performs_store'

find_basic_blocks(*lines*)

Find and return basic blocks (asm sections which can only be executed as complete block).

Blocks always start at a label and end at the next jump/break possibility.

Returns OrderedDict with labels as keys and list of lines as value

find_basic_loop_bodies(*lines*)
Find and return basic loop bodies (asm section which loop back on itself with no other egress).

Returns OrderedDict with labels as keys and list of lines as value

find_jump_labels(*lines*)
Find and return all labels which are followed by instructions until the next label

Returns OrderedDict of mapping from label name to associated line index

8.1.2 Submodules

8.1.3 osaca.db_interface module

sanity_check(*arch: str*, *verbose=False*, *output_file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)
Checks the database for missing TP/LT values, instructions might missing int the ISA DB and duplicate instructions.

Parameters

- **arch** (*str*) – micro-arch key to define DB to check
- **verbose** (*bool, optional*) – verbose output flag, defaults to *False*
- **output_file** (*stream, optional*) – output stream specifying where to write output, defaults to `sys. stdout`

import_benchmark_output(*arch*, *bench_type*, *filepath*, *output=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)
Import benchmark results from micro-benchmarks.

Parameters

- **arch** (*str*) – target architecture key
- **bench_type** (*str*) – key for defining type of benchmark output
- **filepath** (*str*) – filepath to the output file
- **output** (*stream*) – output stream to dump, defaults to sys.stdout

8.1.4 osaca.frontend module

Frontend interface for OSACA. Does everything necessary for analysis report generation.

class Frontend(*filename=''*, *arch=None*, *path_to_yaml=None*)
Bases: `object`

throughput_analysis(*kernel*, *show_lineno=False*, *show_cmnts=True*)
Build throughput analysis only.

Parameters

- **kernel** (*list*) – Kernel to build throughput analysis for.
- **show_lineno** (*bool, optional*) – flag for showing the line number of instructions, defaults to *False*

- **show_cmnts** (*bool, optional*) – flag for showing comment-only lines in kernel, defaults to *True*

latency_analysis(*cp_kernel*, *separator='|'*)
Build a list-based CP analysis report.

Parameters **cp_kernel** (*list*) – loop kernel containing the CP information for each instruction form

Separator separator symbol for the columns, defaults to '|'

loopcarried_dependencies(*dep_dict*, *separator='|'*)
Print a list-based LCD analysis to the terminal.

Parameters **dep_dict** (*dict*) – dictionary with first instruction in LCD as key and the deps as value

Separator separator symbol for the columns, defaults to '|'

full_analysis(*kernel*, *kernel_dg: osaca.semantics.kernel_dg.KernelDG*, *ignore_unknown=False*, *verbose=False*)
Build the full analysis report including header, the symbol map, the combined TP/CP/LCD view and the list based LCD view.

Parameters

- **kernel** (*list*) – kernel to report on
- **kernel_dg** (*KernelDG*) – directed graph containing CP and LCD
- **ignore_unknown** (*boolean, optional*) – flag for ignore warning if performance data is missing, defaults to *False*
- **verbose** (*boolean, optional*) – flag for verbosity level, defaults to False

combined_view(*kernel*, *cp_kernel: osaca.semantics.kernel_dg.KernelDG*, *dep_dict*, *ignore_unknown=False*, *show_cmnts=True*)
Build combined view of kernel including port pressure (TP), a CP column and a LCD column.

Parameters

- **kernel** (*list*) – kernel to report on
- **kernel_dg** (*KernelDG*) – directed graph containing CP and LCD
- **dep_dict** (*dict*) – dictionary with first instruction in LCD as key and the deps as value
- **ignore_unknown** (*bool, optional*) – flag for showing result despite of missing instructions, defaults to *False*
- **show_cmnts** (*bool, optional*) – flag for showing comment-only lines in kernel, defaults to *True*

8.1.5 osaca.osaca module

CLI for OSACA

get_version()
Gets the current OSACA version stated in the __init__ file

Returns str – the version string.

create_parser(*parser=None*)
Return argparse parser.

Parameters **parser** (ArgumentParser) – Existing parser object to add the arguments, defaults to *None*

Returns The newly created ArgumentParser object.

check_arguments(*args*, *parser*)
Check arguments passed by user that are not checked by argparse itself.

Parameters

- **args** – arguments given from ArgumentParser after parsing
- **parser** – ArgumentParser object

import_data(*benchmark_type*, *arch*, *filepath*, *output_file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)
Imports benchmark results from micro-benchmarks.

Parameters

- **benchmark_type** (*str*) – key for defining type of benchmark output
- **arch** (*str*) – target architecture to put the data into the right database
- **filepath** (*str*) – filepath of the output file"
- **output_file** (*stream, optional*) – output stream specifying where to write output, defaults to sys.stdout

insert_byte_marker(*args*)
Inserts byte markers into an assembly file using kerncraft.

Parameters **args** – arguments given from ArgumentParser after parsing

inspect(*args*, *output_file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)
Does the actual throughput and critical path analysis of OSACA and prints it to the terminal.

Parameters

- **args** – arguments given from ArgumentParser after parsing
- **output_file** (*stream, optional*) – Define the stream for output, defaults to sys.stdout

run(*args*, *output_file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)
Main entry point for OSACAs workflow. Decides whether to run an analysis or other things.

Parameters

- **args** – arguments given from ArgumentParser after parsing
- **output_file** (*stream, optional*) – Define the stream for output, defaults to sys.stdout

get_asm_parser(*arch*) → osaca.parser.base_parser.BaseParser
Helper function to create the right parser for a specific architecture.

Parameters **arch** (*str*) – architecture code

Returns *BaseParser* object

get_unmatched_instruction_ratio(*kernel*)
Return ratio of unmatched from total instructions in kernel.

main()
Initialize and run command line interface.

8.1.6 osaca.utils module

find_file(*name*)
Check for existence of name in user or package data folders and return path.

exists_cached_file(*name*)
Check for existence of file in cache dir. Returns path if it exists and False otherwise.

Python Module Index

o

Index

H

I

K

L

M

N

O

P

R

S

T

W