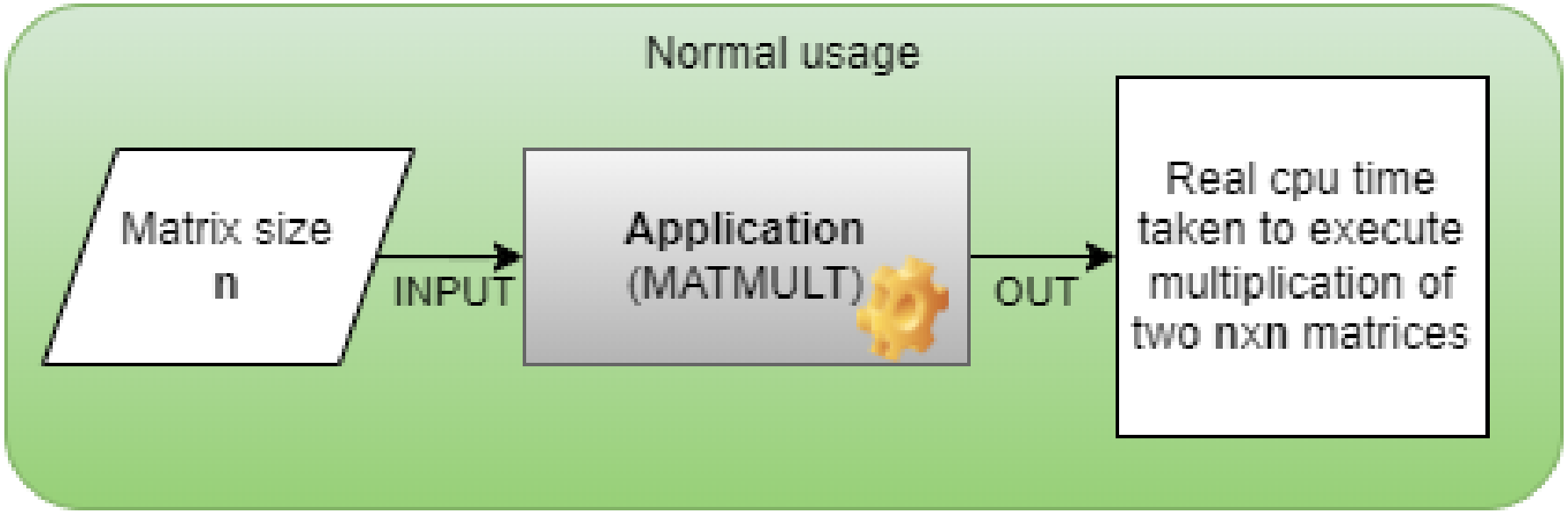
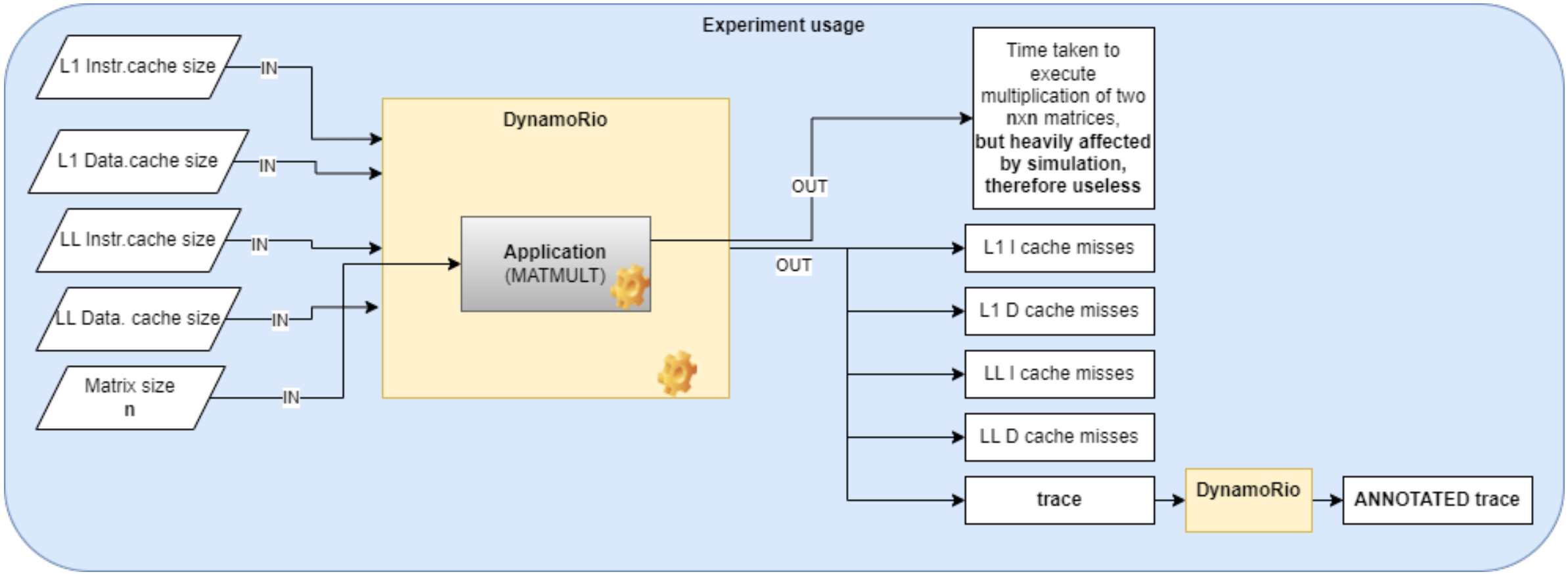
ERICSSON CASE

Progress so far and potential ideas

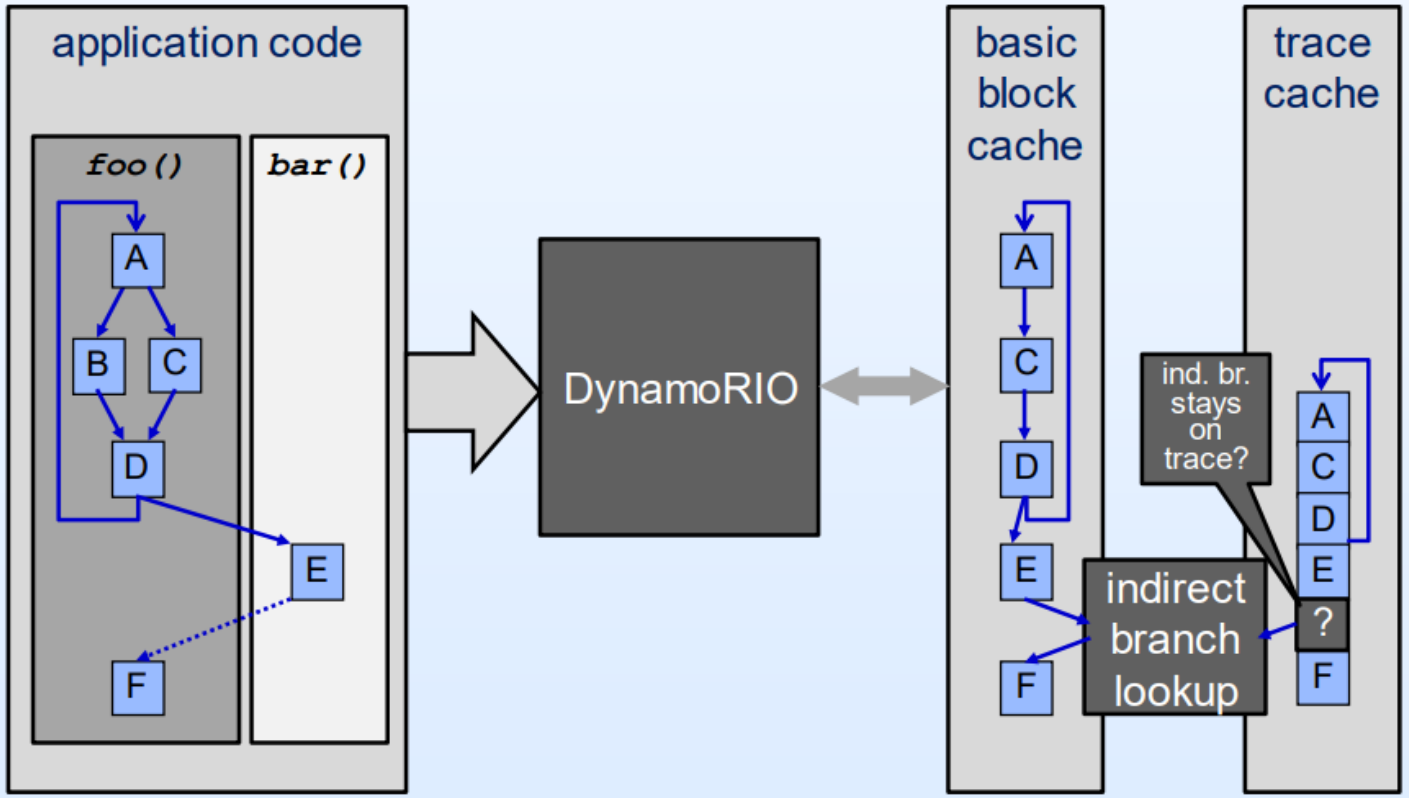
Basic idea – refresher



```
frag7: add  %eax, %ecx
       cmp  $4, %eax
       jle  <frag8>
       jmp  <stub1>
stub0: mov  %eax, eax-slot
       mov  &dstub0, %eax
       jmp  context_switch
stub1: mov  %eax, eax-slot
       mov  &dstub1, %eax
       jmp  context_switch
```



Improvement #4: Trace Building



Slowdown: ~~300x~~ ~~25x~~ ~~3x~~ ~~1.2x~~ 1.1x

Source: „Building Dynamic Tools with DynamoRIO on x86 and ARMv8“, Chris Adeniyi-Jones Edmund Grimley Evans Kevin Zhou

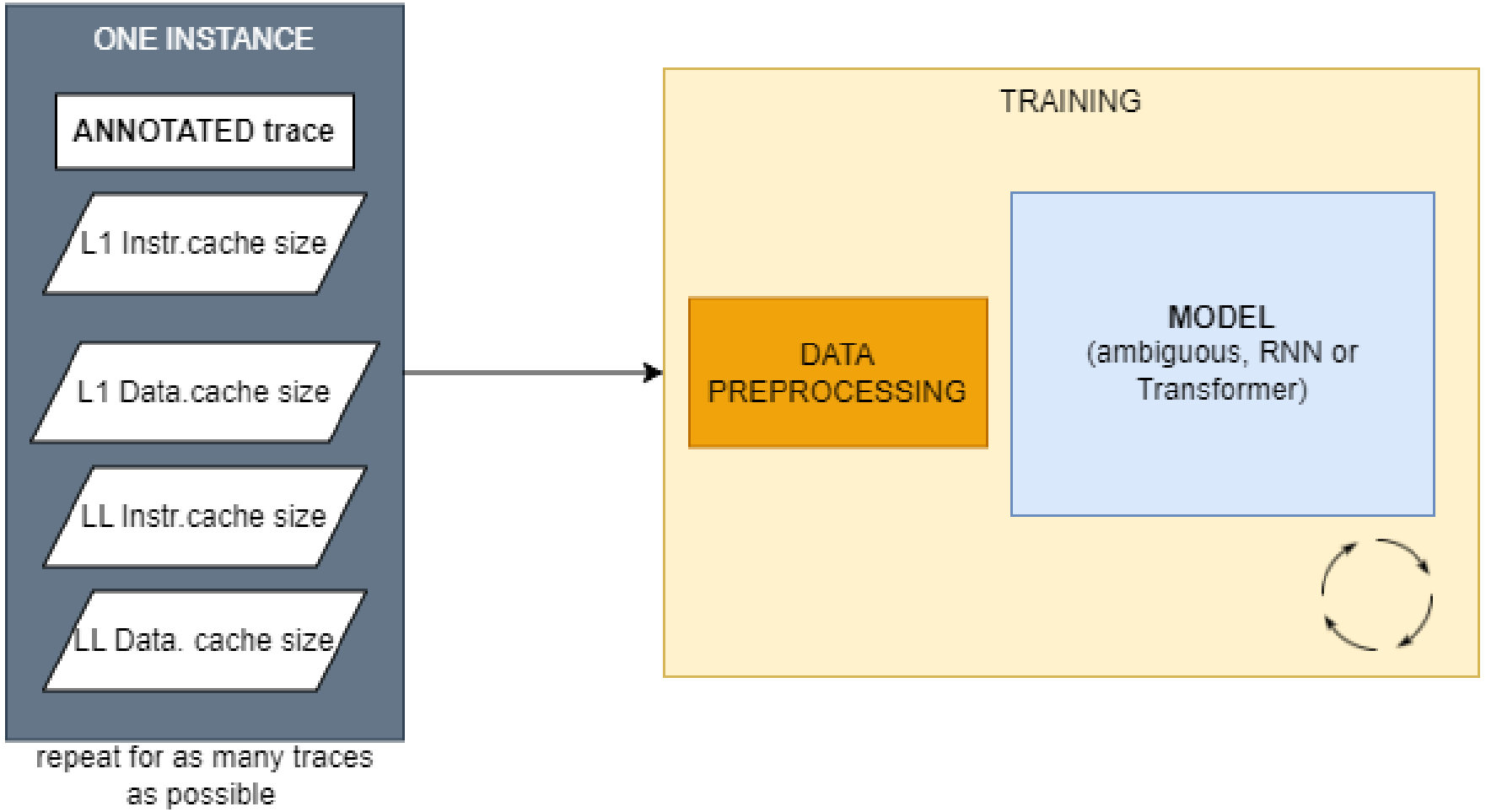
ANNOTATED TRACE

```
[11]ifetch      4 byte(s) @ 0x00007fd7791b1050 f3 0f 1e fa          nop    %edx
INST MISS

[12]ifetch      1 byte(s) @ 0x00007fd7791b1054 55                   push   %rbp %rsp -> %rsp 0xfffffff8(%rsp)[8byte]
[13]write       8 byte(s) @ 0x00007ffe44e2c150 by PC 0x00007fd7791b1054
[14]ifetch      3 byte(s) @ 0x00007fd7791b1055 48 89 e5             mov    %rsp -> %rbp
[15]ifetch      2 byte(s) @ 0x00007fd7791b1058 41 57                push   %r15 %rsp -> %rsp 0xfffffff8(%rsp)[8byte]
[16]write       8 byte(s) @ 0x00007ffe44e2c148 by PC 0x00007fd7791b1058
[17]ifetch      2 byte(s) @ 0x00007fd7791b105a 41 56                push   %r14 %rsp -> %rsp 0xfffffff8(%rsp)[8byte]
[18]write       8 byte(s) @ 0x00007ffe44e2c140 by PC 0x00007fd7791b105a
[19]ifetch      2 byte(s) @ 0x00007fd7791b105c 41 55                push   %r13 %rsp -> %rsp 0xfffffff8(%rsp)[8byte]
[20]write       8 byte(s) @ 0x00007ffe44e2c138 by PC 0x00007fd7791b105c
DATA MISS

[21]ifetch      2 byte(s) @ 0x00007fd7791b105e 41 54                push   %r12 %rsp -> %rsp 0xfffffff8(%rsp)[8byte]
[22]write       8 byte(s) @ 0x00007ffe44e2c130 by PC 0x00007fd7791b105e
[23]ifetch      1 byte(s) @ 0x00007fd7791b1060 53                   push   %rbx %rsp -> %rsp 0xfffffff8(%rsp)[8byte]
[24]write       8 byte(s) @ 0x00007ffe44e2c128 by PC 0x00007fd7791b1060
[25]ifetch      7 byte(s) @ 0x00007fd7791b1061 48 81 ec 88 00 00 00 sub    $0x0000000000000088 %rsp -> %rsp
[26]ifetch      4 byte(s) @ 0x00007fd7791b1068 48 89 7d 88          mov    %rdi -> 0xffffff88(%rbp)[8byte]
[27]write       8 byte(s) @ 0x00007ffe44e2c0d8 by PC 0x00007fd7791b1068
DATA MISS

[28]ifetch      2 byte(s) @ 0x00007fd7791b106c 0f 31                rdtsc   -> %edx %eax
[29]ifetch      7 byte(s) @ 0x00007fd7791b106e 4c 8d 25 8b ef fd ff lea    <rel> 0x00007fd779190000 -> %r12
[30]ifetch      7 byte(s) @ 0x00007fd7791b1075 80 25 92 9d 01 00 df and    $0xdf <rel> 0x00007fd7791cae0e[1byte] -> <rel> 0x00007fd7791cae0e[1byte]
[31]read        1 byte(s) @ 0x00007fd7791cae0e by PC 0x00007fd7791b1075
DATA MISS
```

Important notes

- Issues that happened before: Matmult ran slowly for large n. This was resolved by scaling (Peter‘s idea).
- Namely, n does not matter so much as the ratio of n and the cache sizes. If we scale n so that the cache sizes are filled with the created matrices, then n doesn‘t need to be too big and therefore we can run many more instances in the same timeframe.
- Example: n = 100 means two matrices of 10000 32bit integers = 3.90625 kB
- If we choose L1 D cache >> 4 kB, then we get very few cache misses, since the matrices fill up the cache entirely.
- Alternatively, if, to correct this, we choose a larger n, we unnecessarily slow down the execution.
- Therefore, matrix sizes and cache sizes need to be chosen approprately.

Model training phase



Two approaches to the annotated trace

- The data in the annotated trace is sequential in nature.
- To extract the features we want, Peter and I came up with two strategies.
- 1) Block strategy, where we take separate the annotated trace into blocks (e.g. of 100 instructions) and we count the data misses in those blocks
- 2) Exponential strategy, where we assign an exponential function value to the series of instructions that cause a cache miss, in such a way that 0 means start of the series and 1 the peak of the series (the data miss itself)

EXAMPLE OF A MODEL FOR AN EXPONENTIAL STRATEGY

```python
class MissPredictionModel(nn.Module):
  def __init__(self, n_features=3, n_hidden=128, n_layers=2) -> None:
    super().__init__()
    self.n_hidden=n_hidden
    self.lstm = nn.LSTM(
        input_size=n_features,
        hidden_size=n_hidden,
        batch_first = True,
        num_layers=n_layers, # stack layers on top of each other
        dropout = 0.2 # using this is a bit tricky but pytorch sorts it out auto
        )
    self.regressor = nn.Linear(n_hidden,1)
    # self.activation = torch.nn.Sigmoid() # ovo je krajnji output layer!!!!

  def forward(self, x):
    self.lstm.flatten_parameters() # sorts out the GPU memory while doing distributed
    _,(hidden, _) = self.lstm(x)
    out_hidden = hidden[-1]
    out = self.regressor(out_hidden)# output of the last layer
    return out
```

MODEL

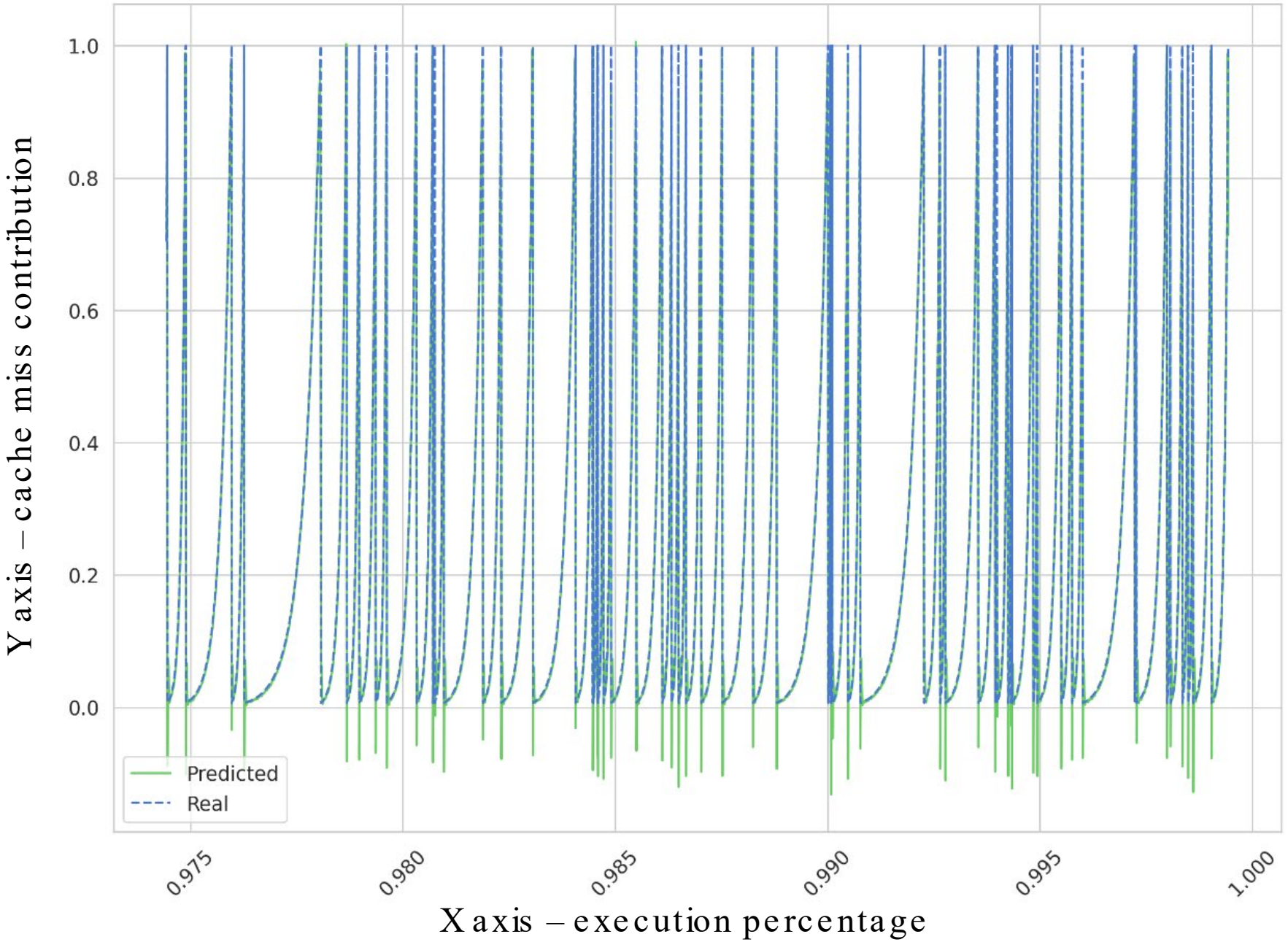
```python
from pytorch_lightning.callbacks import ModelCheckpoint, EarlyStopping
from pytorch_lightning.loggers import TensorBoardLogger
checkpoint_callback = ModelCheckpoint(
    dirpath = "checkpoints",
    filename = "best-checkpoint",
    save_top_k=1, # save only the best model
    verbose = True,
    monitor = 'val_loss',
    mode = "min"
)

logger = TensorBoardLogger("lightning-logs", name="miss_value")
early_stopping_callback = EarlyStopping("val_loss") # ako se proslih par epoha nis nije pormijeilo, zaustavi trening

trainer = pl.Trainer(
    logger=logger,
    callbacks= [early_stopping_callback, checkpoint_callback],
    max_epochs=N_EPOCHS,
    devices=1, accelerator="gpu",
    enable_progress_bar=True
)
```

TRAINING

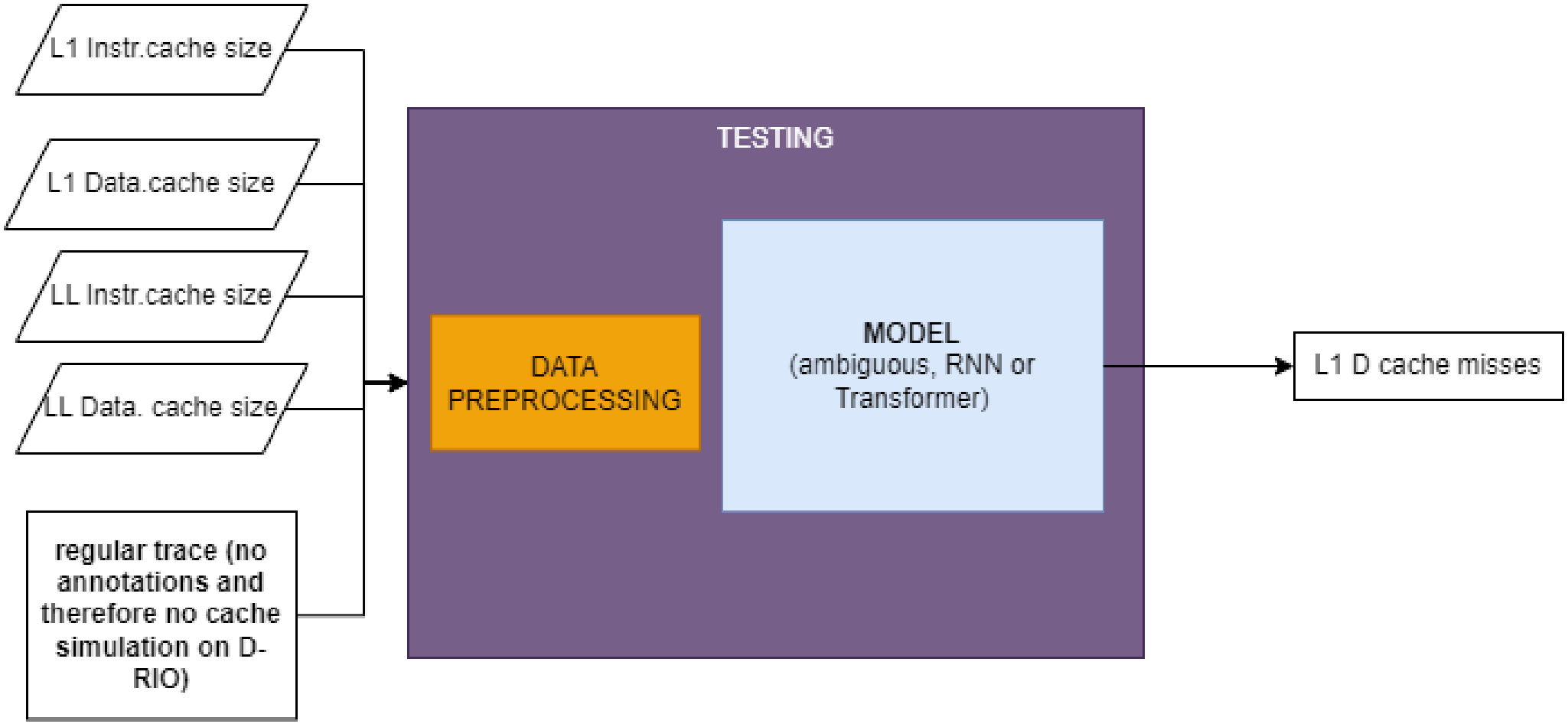
Data used (instructions annotated with exponential function values)

	execution_percentage	instruction_name	miss_contribution
0	0.000000	6.0	0.006738
1	0.000005	17.0	0.006939
2	0.000010	12.0	0.007042
3	0.000015	5.0	0.007146
4	0.000020	25.0	0.007252
...	...	...	...
199882	0.999410	12.0	0.776340
199883	0.999415	19.0	0.827064
199884	0.999420	2.0	0.881102
199885	0.999425	16.0	0.938670
199886	0.999430	2.0	1.000000

199887 rows × 3 columns


Y axis – cache miss contribution
1.0
0.8
0.6
0.4
0.2
0.0
Predicted
Real
0.975
0.980
0.985
0.990
0.995
1.000
X axis – execution percentage

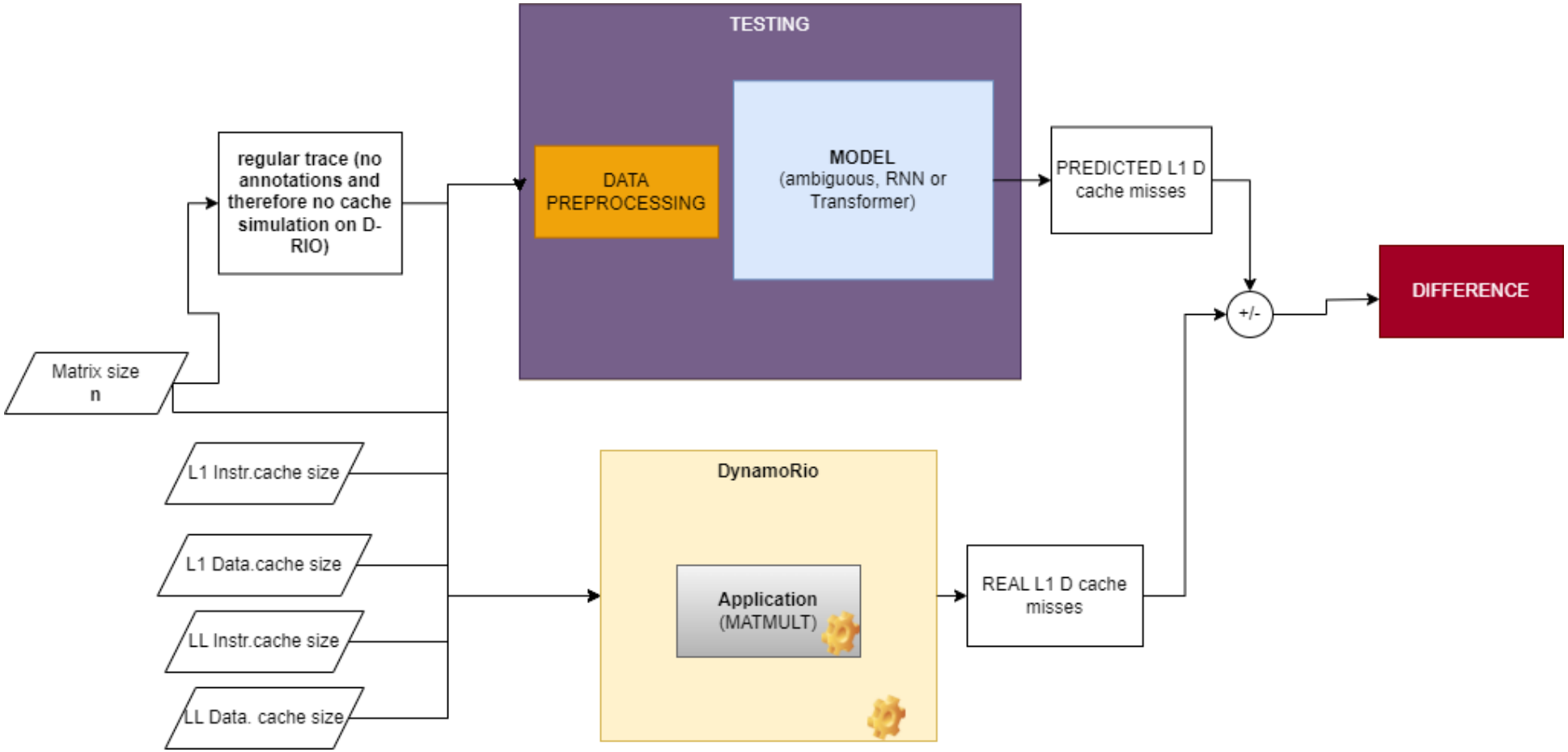
Final target of the model



REGULAR TRACE (NO ANNOTATIONS)

```
[184]ifetch      4 byte(s) @ 0x00007fd7791b10fa 48 83 f8 22          cmp    %rax $0x0000000000000022
[185]ifetch      2 byte(s) @ 0x00007fd7791b10fe 76 e9                jbe    $0x00007fd7791b10e9
[186]ifetch      4 byte(s) @ 0x00007fd7791b10e9 48 89 14 c1          mov    %rdx -> (%rcx,%rax,8)[8byte]
[187]write       8 byte(s) @ 0x00007fd7791cab70 by PC 0x00007fd7791b10e9
[188]ifetch      4 byte(s) @ 0x00007fd7791b10ed 48 8b 42 10          mov    0x10(%rdx)[8byte] -> %rax
[189]read        8 byte(s) @ 0x00007fd7791c9f50 by PC 0x00007fd7791b10ed
[190]ifetch      4 byte(s) @ 0x00007fd7791b10f1 48 83 c2 10          add    $0x0000000000000010 %rdx -> %rdx
[191]ifetch      3 byte(s) @ 0x00007fd7791b10f5 48 85 c0             test   %rax %rax
[192]ifetch      2 byte(s) @ 0x00007fd7791b10f8 74 4e                jz     $0x00007fd7791b1148
[193]ifetch      4 byte(s) @ 0x00007fd7791b10fa 48 83 f8 22          cmp    %rax $0x0000000000000022
[194]ifetch      2 byte(s) @ 0x00007fd7791b10fe 76 e9                jbe    $0x00007fd7791b10e9
[195]ifetch      4 byte(s) @ 0x00007fd7791b10e9 48 89 14 c1          mov    %rdx -> (%rcx,%rax,8)[8byte]
[196]write       8 byte(s) @ 0x00007fd7791cab78 by PC 0x00007fd7791b10e9
[197]ifetch      4 byte(s) @ 0x00007fd7791b10ed 48 8b 42 10          mov    0x10(%rdx)[8byte] -> %rax
[198]read        8 byte(s) @ 0x00007fd7791c9f60 by PC 0x00007fd7791b10ed
[199]ifetch      4 byte(s) @ 0x00007fd7791b10f1 48 83 c2 10          add    $0x0000000000000010 %rdx -> %rdx
[200]ifetch      3 byte(s) @ 0x00007fd7791b10f5 48 85 c0             test   %rax %rax
[201]ifetch      2 byte(s) @ 0x00007fd7791b10f8 74 4e                jz     $0x00007fd7791b1148
[202]ifetch      4 byte(s) @ 0x00007fd7791b10fa 48 83 f8 22          cmp    %rax $0x0000000000000022
[203]ifetch      2 byte(s) @ 0x00007fd7791b10fe 76 e9                jbe    $0x00007fd7791b10e9
[204]ifetch      3 byte(s) @ 0x00007fd7791b1100 48 89 fe             mov    %rdi -> %rsi
[205]ifetch      3 byte(s) @ 0x00007fd7791b1103 48 29 c6             sub    %rax %rsi -> %rsi
[206]ifetch      4 byte(s) @ 0x00007fd7791b1106 48 83 fe 0f          cmp    %rsi $0x000000000000000f
```

Validation stage



Main ideas for the future

- Analyze the difference (between real/predicted cache misses), which is important to Ericsson
- PAPER IDEAS
- Analyze potential usages of similar systems for predicting cache misses
- Compare the efficiency of using a RNN / transformer model vs just using basic regressions (tradeoff / usability, maybe regressions aren't as accurate but they may be faster)
- Introduce novel ideas for predicting cache misses (that may or may not pan out to usable stuff but may lead to usable stuff in the future?)

Prerequisites

- Learning about these approaches in general
- Learning about what was already done in the fiels
- Analyzing new ways this approach could be checked/verified for the future

Links to GitHub repos (for now)

- https://github.com/jelacicedin/drio-data-collection
- https://github.com/jelacicedin/timeseries-cache-miss-forecasting
- https://github.com/ptrbman/dynamorio-missing-instructions

SOTA

https://mlforsystems.org/assets/papers/neurips2018/cache_jha_2018.pdf

Cache Miss Rate Predictability via Neural Networks

Rishikesh Jha *
College of Information and Computer Sciences
University of Massachusetts Amherst
Amherst, MA 01003
rishikeshjha@cs.umass.edu

Arjun Karuvally *
College of Information and Computer Sciences
University of Massachusetts Amherst
Amherst, MA 01003
akaruvally@cs.umass.edu

Saket Tiwari *
College of Information and Computer Sciences
University of Massachusetts Amherst
Amherst, MA 01003
sakettiwari@cs.umass.edu

J. Eliot B. Moss
College of Information and Computer Sciences
University of Massachusetts Amherst
Amherst, MA 01003
moss@cs.umass.edu

Abstract

A program run, in the setting of computer architecture and compilers, can be characterized in part by its memory access patterns. We approach the problem of analyzing these patterns using machine learning. We characterize memory accesses using a sequence of *cache miss rates*, and present a new data set for this task. The data set draws from programs run on various Java virtual machines, and C and Fortran compilers. We work towards answering the scientific question: How predictable is a program's cache miss rate from interval to interval as it executes? We report the results of three distinct ANN models, which have been shown to be effective in sequence modeling. We show that programs can be differentiated in terms of the predictability of their cache miss rates.

SOTA

- https://doi.org/10.1145/3373376.3378498
- https://studentmdh.sharepoint.com/:b:/r/sites/PerFlex/Delade%20dokument/5.EDIN/2.RelatedWork/3373376.3378498.pdf?csf=1&web=1&e=pip7Uk

Classifying Memory Access Patterns for Prefetching

Grant Ayers*
Stanford University

Heiner Litz*
UC Santa Cruz

Christos Kozyrakis
Stanford University, Google

Parthasarathy Ranganathan
Google

Abstract

Prefetching is a well-studied technique for addressing the memory access stall time of contemporary microprocessors. However, despite a large body of related work, the memory access behavior of applications is not well understood, and it remains difficult to predict whether a particular application will benefit from a given prefetcher technique. In this work we propose a novel methodology to classify the memory access patterns of applications, enabling well-informed reasoning about the applicability of a certain prefetcher. Our approach leverages instruction dataflow information to un-