Tutorial: A Portable and Efficient Framework for Fine-grained Value Profilers

Xin You

Beihang University

Hands-on Tutorial @ CLUSTER24

北京航空航天大學
BEIHANG UNIVERSITY

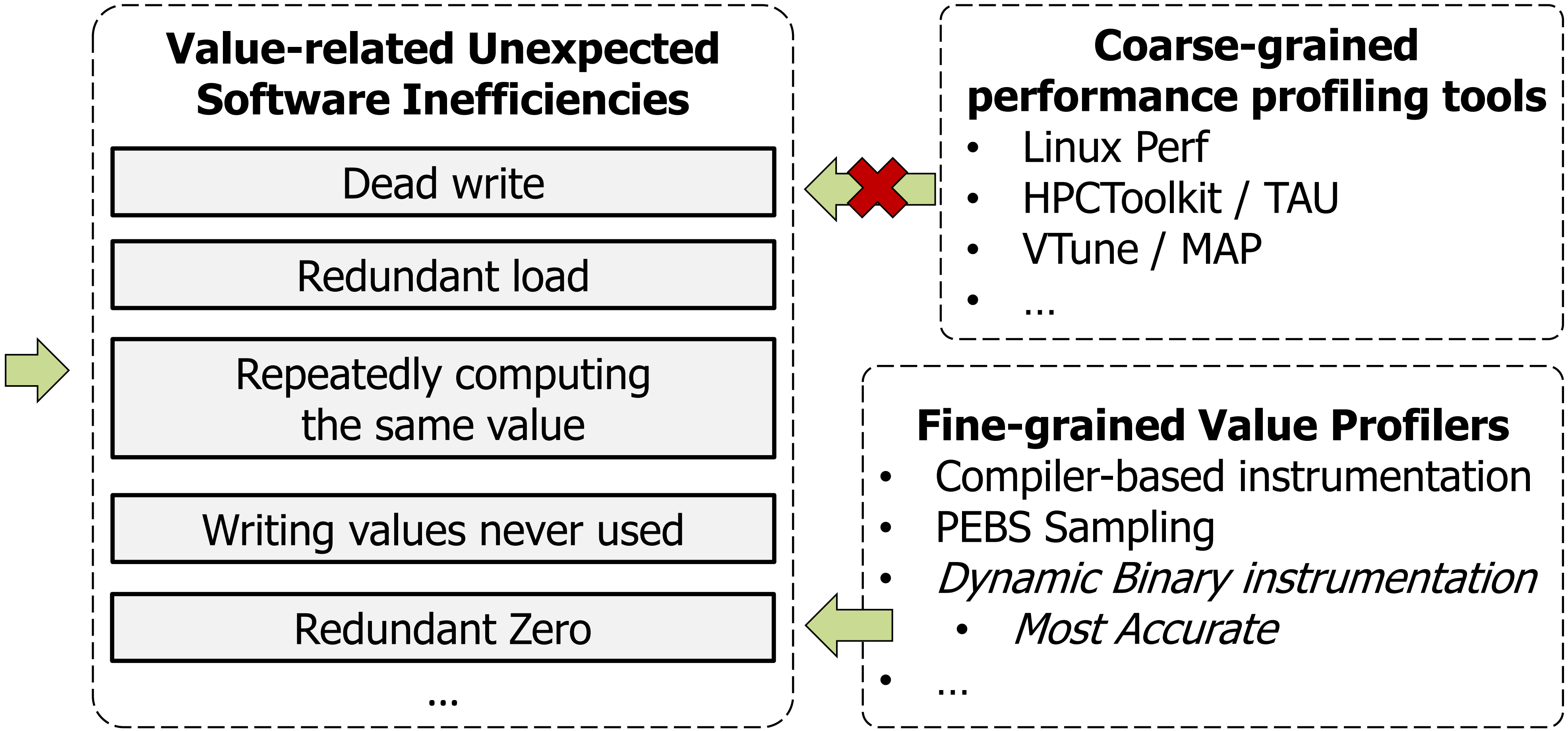
Outline

- Introduction
- Design Principles & Operand-centric Two-level Designs
- Methodology & Implementation
- Evaluation
- Hand-on Tutorial

Outline

- Introduction
- Design Principles & Operand-centric Two-level Designs
- Methodology & Implementation
- Evaluation
- Hand-on Tutorial

Introduction



Value-related Unexpected Software Inefficiencies

- Dead write
- Redundant load
- Repeatedly computing the same value
- Writing values never used
- Redundant Zero
- …

Coarse-grained performance profiling tools

- Linux Perf
- HPCToolkit / TAU
- VTune / MAP
- …

Fine-grained Value Profilers

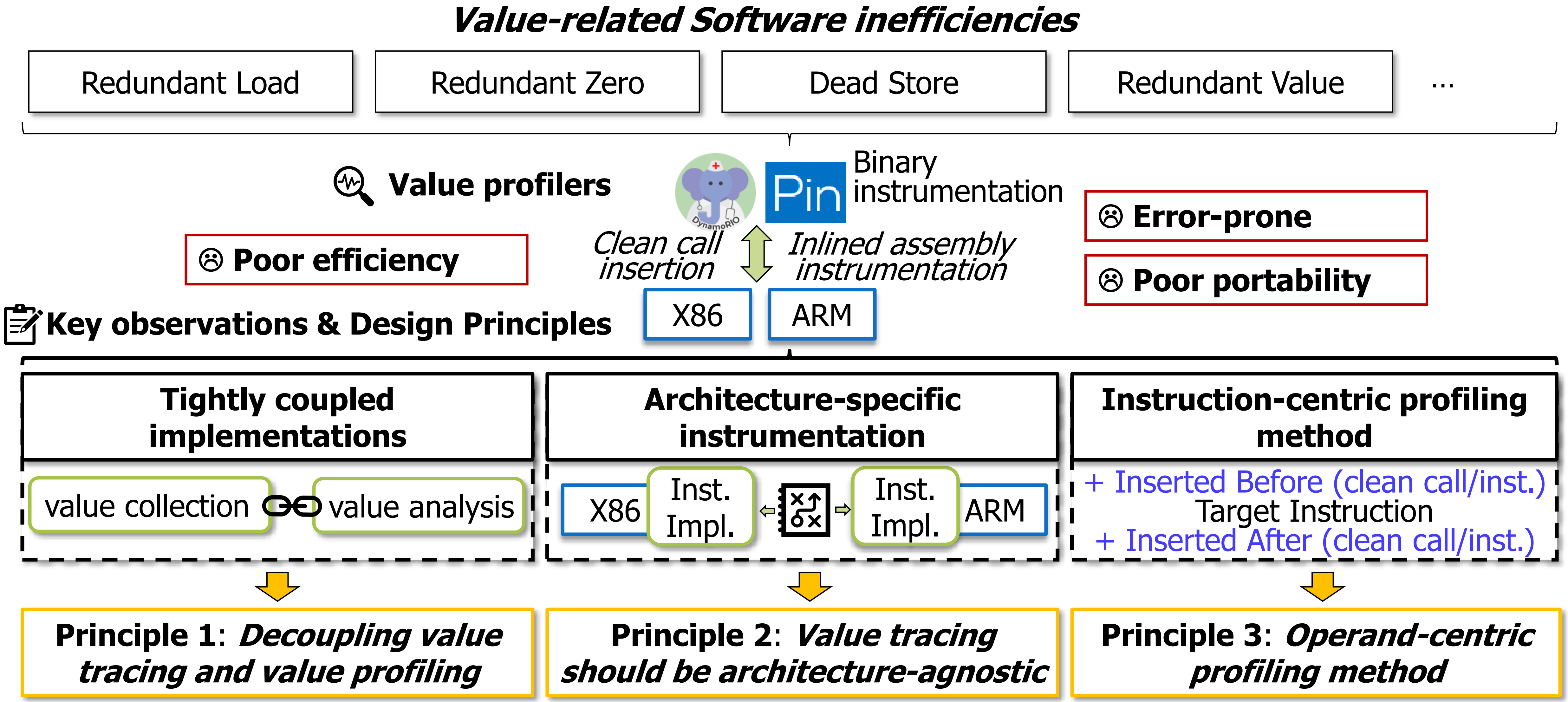
- Compiler-based instrumentation
- PEBS Sampling
- *Dynamic Binary instrumentation*
  - *Most Accurate*
- …

- Dynamic binary instrumentations is one of the most widely adopted techniques to develop fine-grained value profilers.

Outline

- Introduction
- Design Principles & Operand-centric Two-level Designs
- Methodology & Implementation
- Evaluation
- Hand-on Tutorial

Design Principles



Value-related Software inefficiencies

Redundant Load | Redundant Zero | Dead Store | Redundant Value | …

Value profilers

DynamoRIO

Pin

Binary instrumentation

☹ Poor efficiency

Clean call insertion ⇕ *Inlined assembly instrumentation*

X86 | ARM

☹ Error-prone

☹ Poor portability

Key observations & Design Principles

Tightly coupled implementations

value collection ⚭ value analysis

Architecture-specific instrumentation

X86 Inst. Impl. ⇦ ⇨ Inst. Impl. ARM

Instruction-centric profiling method

\+ Inserted Before (clean call/inst.)
Target Instruction
\+ Inserted After (clean call/inst.)

Principle 1: ***Decoupling value tracing and value profiling***

Principle 2: ***Value tracing should be architecture-agnostic***

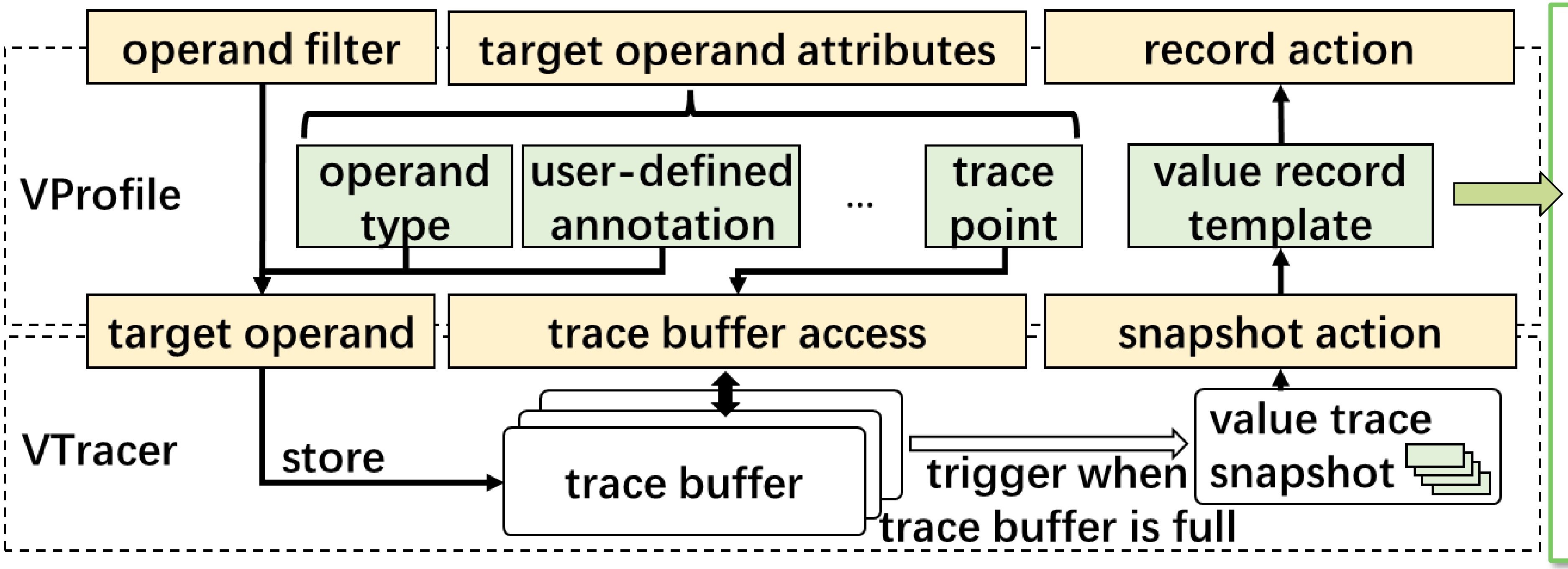
Principle 3: ***Operand-centric profiling method***

Operand-centric two-level design

Principle 1: ***Decoupling value tracing and value profiling***

Principle 2: ***Value tracing should be architecture-agnostic***

Principle 3: ***Operand-centric fine-grained value analysis***



Operand Attributes:

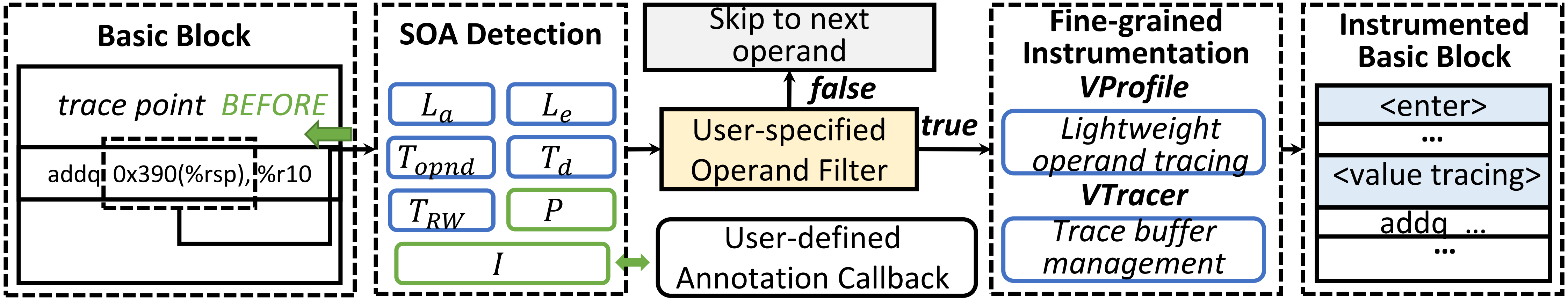
- T_{opnd}: operand type
- T_{RW}: READ/WRITE
- A: Address/Register
- V: runtime value
- C: calling context
- P: trace point
- T_d: data type
- L_a: access size
- L_e: element size
- I: user-defined annotation

- Based on **Principle 1~3**, we propose the *operand-centric two-level designs* and implement a prototype framework VClinic to achieve portability and efficiency for fine-grained value profilers.

Outline

- Introduction
- Design Principles & Operand-centric Two-level Designs
- Methodology & Implementation
- Evaluation
- Hand-on Tutorial

Methodology & Implementation



Basic Block

trace point *BEFORE*

addq 0x390(%rsp), %r10

SOA Detection

L_a L_e

T_{opnd} T_d

T_{RW} P

I

Skip to next operand

false

User-specified Operand Filter

true

User-defined Annotation Callback

Fine-grained Instrumentation

VProfile

Lightweight operand tracing

VTracer

Trace buffer management

Instrumented Basic Block

<enter>

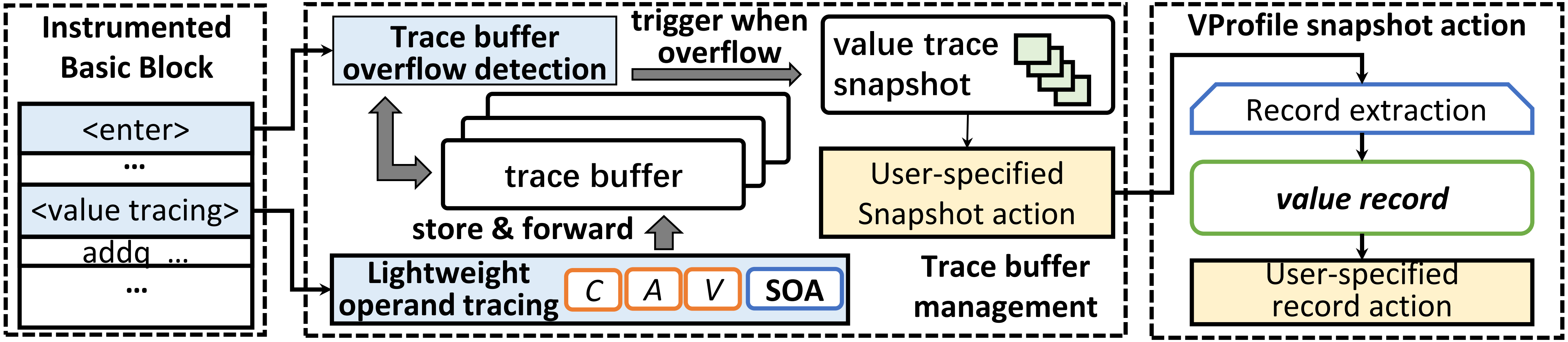
…

<value tracing>

addq …

…

(a) Instrumentation



Instrumented Basic Block

<enter>

…

<value tracing>

addq …

…

Trace buffer overflow detection

trigger when overflow

value trace snapshot

trace buffer

store & forward

Lightweight operand tracing *C* *A* *V* **SOA**

User-specified Snapshot action

Trace buffer management

VProfile snapshot action

Record extraction

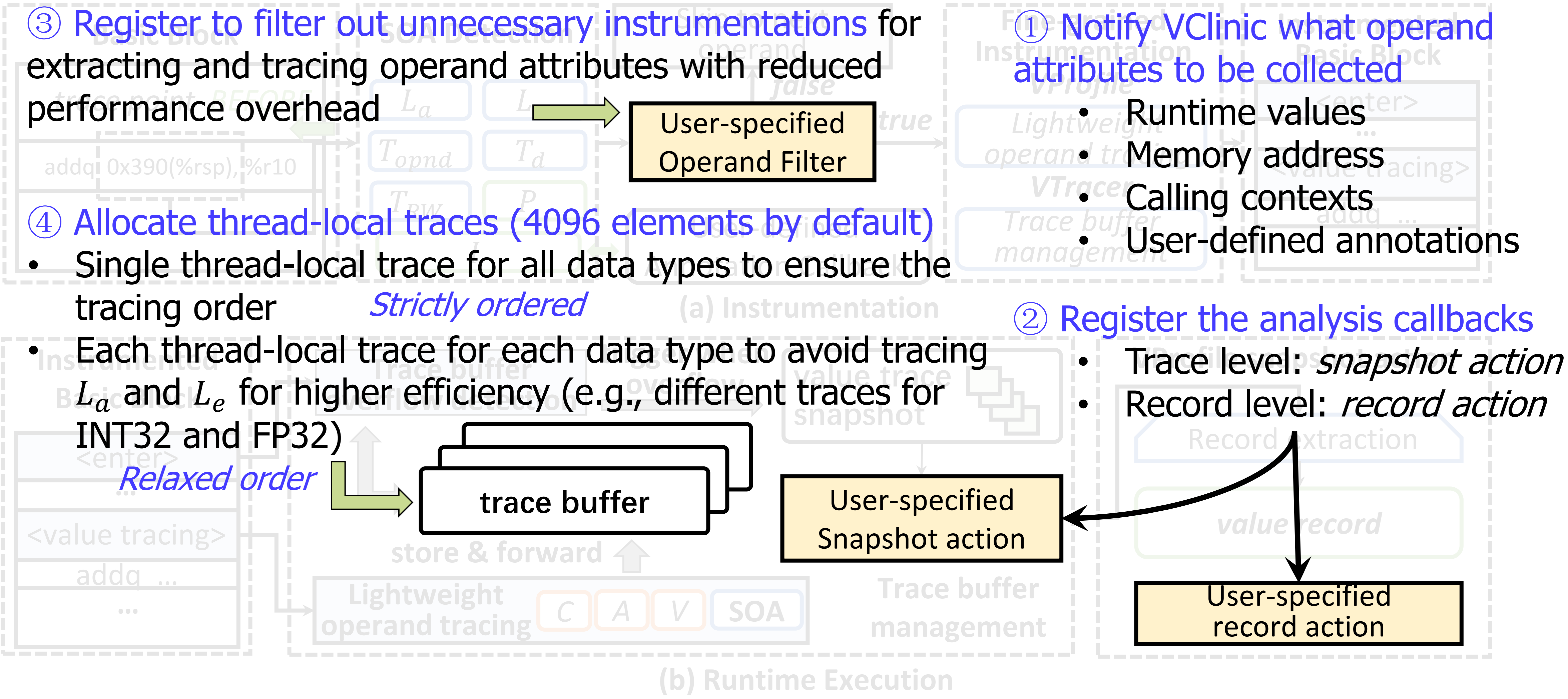
value record

User-specified record action

(b) Runtime Execution

Initialization → ***Instrumentation*** → ***Runtime execution*** → ***Finalization***

Methodology & Implementation



③ Register to filter out unnecessary instrumentations for extracting and tracing operand attributes with reduced performance overhead

④ Allocate thread-local traces (4096 elements by default)

- Single thread-local trace for all data types to ensure the tracing order *Strictly ordered*
- Each thread-local trace for each data type to avoid tracing L_a and L_e for higher efficiency (e.g., different traces for INT32 and FP32) *Relaxed order*

① Notify VClinic what operand attributes to be collected

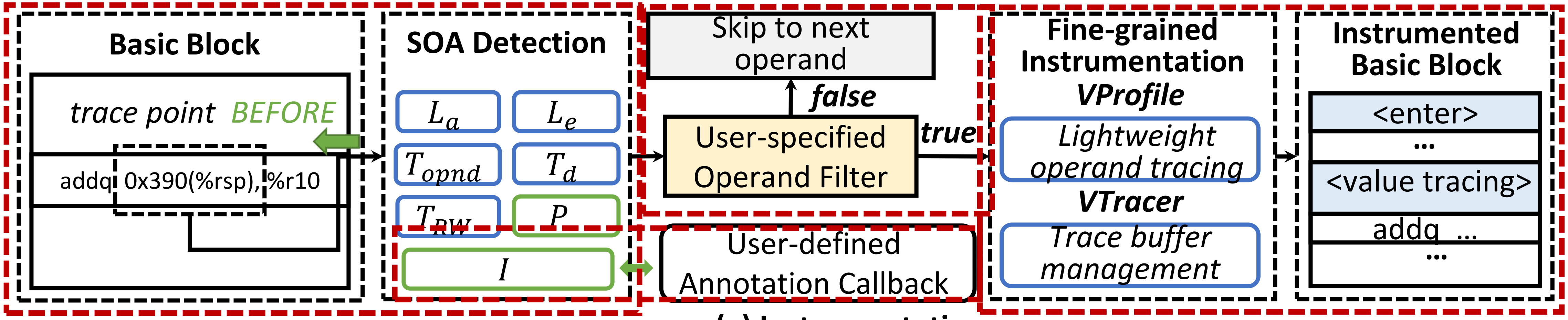
- Runtime values
- Memory address
- Calling contexts
- User-defined annotations

② Register the analysis callbacks

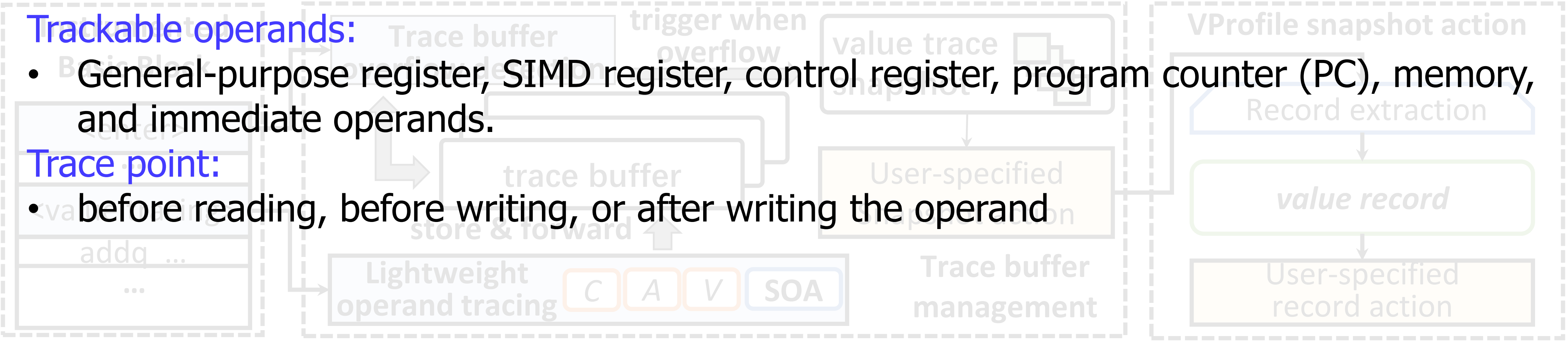
- Trace level: *snapshot action*
- Record level: *record action*

Initialization → ***Instrumentation*** → ***Runtime execution*** → ***Finalization***

Methodology & Implementation



(a) Instrumentation



Trackable operands:

- General-purpose register, SIMD register, control register, program counter (PC), memory, and immediate operands.

Trace point:

- before reading, before writing, or after writing the operand

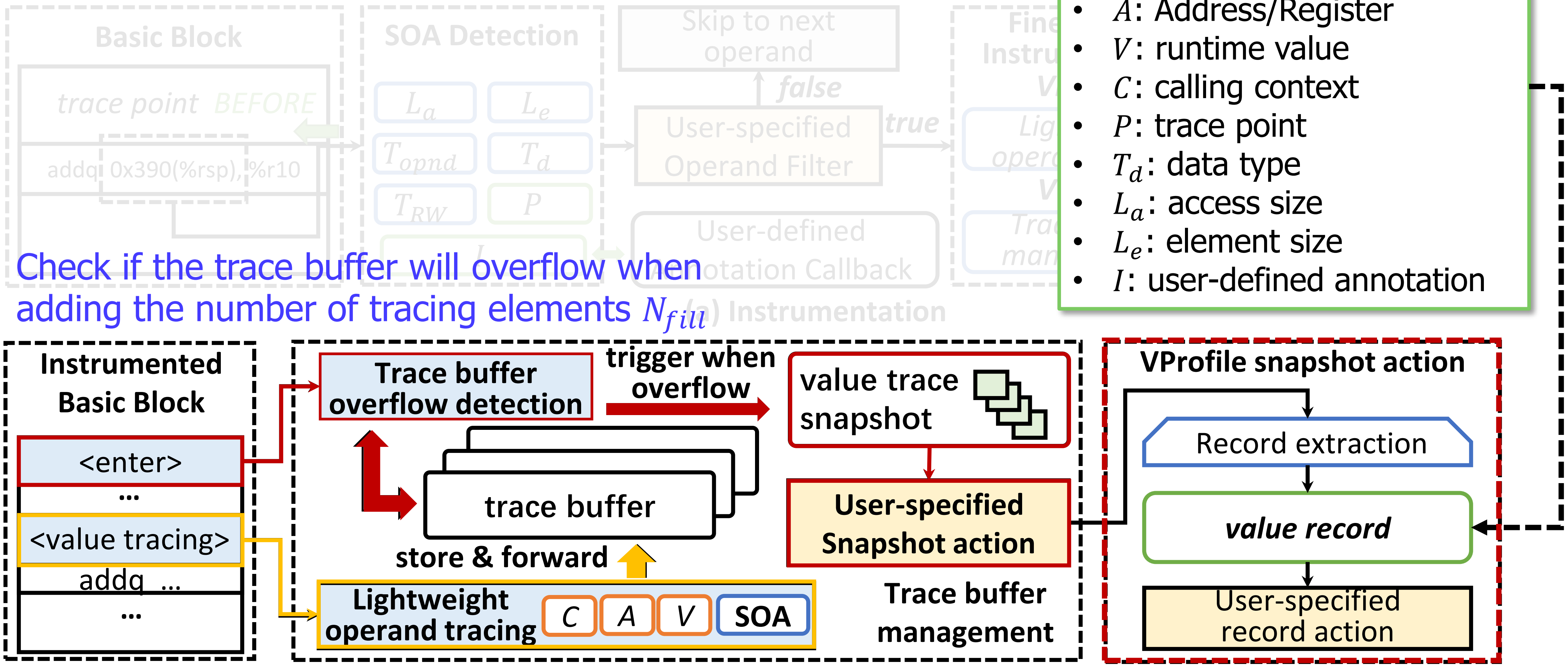
(b) Runtime Execution

Initialization → ***Instrumentation*** → ***Runtime execution*** → ***Finalization***

Methodology & Implementation

value record template:

- T_{opnd}: operand type
- T_{RW}: READ/WRITE
- A: Address/Register
- V: runtime value
- C: calling context
- P: trace point
- T_d: data type
- L_a: access size
- L_e: element size
- I: user-defined annotation



Check if the trace buffer will overflow when adding the number of tracing elements N_{fill}

Instrumented Basic Block

<enter>

…

<value tracing>

addq …

…

Trace buffer overflow detection

trigger when overflow

value trace snapshot

trace buffer

store & forward

Lightweight operand tracing

C *A* *V* **SOA**

User-specified Snapshot action

Trace buffer management

VProfile snapshot action

Record extraction

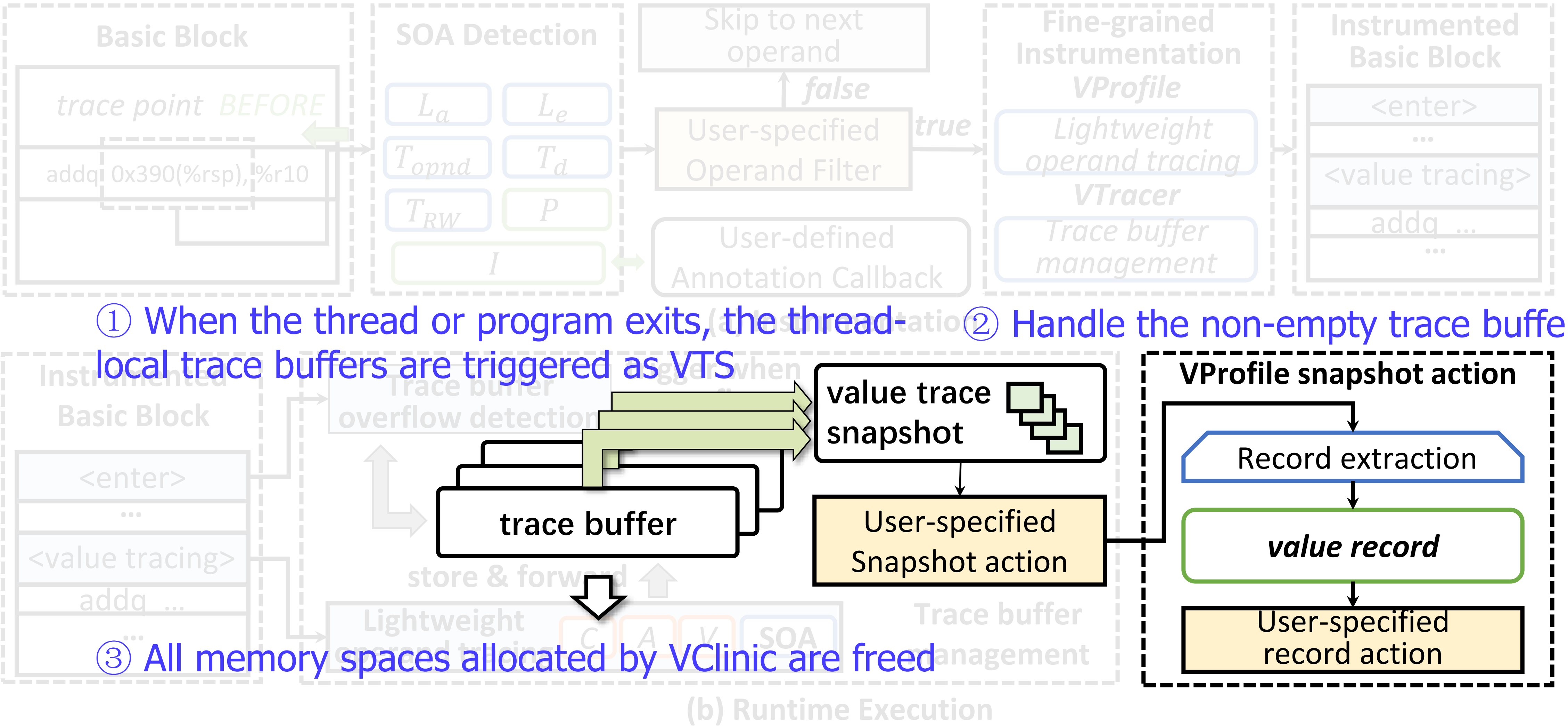
value record

User-specified record action

(b) Runtime Execution

Initialization → ***Instrumentation*** → ***Runtime execution*** → ***Finalization***

Methodology & Implementation



① When the thread or program exits, the thread-local trace buffers are triggered as VTS

② Handle the non-empty trace buffers

③ All memory spaces allocated by VClinic are freed

Initialization → ***Instrumentation*** → ***Runtime execution*** → ***Finalization***

Example – Redundant Value Profiler

```
bool REDSPY_FILTER_OPND ( opnd_t opnd , vprofile_src_t opmask ) {
        uint32_t mask1 =( ANY_DATA_TYPE | GPR_REGISTER | SIMD_REGISTER |
                WRITE | AFTER ) ;
        uint32_t mask2 =( ANY_DATA_TYPE | MEMORY | WRITE | BEFORE | AFTER ) ;
        return (( mask1 & opmask ) == opmask ) ||(( mask2 & opmask ) == opmask ) ;
}
DR_EXPORT void dr_client_main (...) {
        vprofile_init ( INSTR_FILTER ,0 ,0 ,0 , VPROFILE_COLLECT_CCT ) ;
        vtrace = vprofile_allocate_trace ( VPROFILE_TRACE_VAL_CCT_ADDR |
                VPROFILE_TRACE_BEFORE_WRITE |
                VPROFILE_TRACE_STRICTLY_ORDERED ) ;
        uint32_t opnd_mask =( ANY_DATA_TYPE | GPR_REGISTER | SIMD_REGISTER |
                MEMORY | WRITE | BEFORE | AFTER ) ;
        vprofile_register_trace_cb ( vtrace , REDSPY_FILTER_OPND ,
                opnd_mask , ANY , trace_update_cb ) ;
        ...
}
```

register-level

memory-level

Analysis implemented in registered record action

Detailed API Doc: https://VClinic.readthedocs.io/en/latest/index.html

Outline

- Introduction
- Design Principles & Operand-centric Two-level Designs
- Methodology & Implementation
- Evaluation
  - Experimental Setup
  - Productivity
  - Runtime and Memory Overhead & Scalability
  - Case Studies
- Hand-on Tutorial

Experimental Setup

- Implement representative value profilers with VClinic
  - Redundant load (*Dr.Load*), redundant zero (*Dr.Zero*), dead stores (*Dr.Dead*), value redundancies (*Dr.Red*)
  - Compare X86-specific value profilers (*Pin-based*): *LoadSpy*, *ZeroSpy*, *DeadSpy*, *RedSpy*
- NPB 3.4.2 with class C input
- Calling contexts obtained via DrCCTProf

Platform	X86	ARM
CPU	Xeon E5-2680v4@2.40GHz	Cavium ThunderX2@2.50GHz
Core	14	32
Memory	256 GB DDR4	128GB DDR4
Compiler	GCC 9.3.0 –g –O3 -fopenmp	GCC 9.4.0 –g –O3 -fopenmp
System	Ubuntu 20.04 Linux 5.8.0-55-generic	Ubuntu 20.04 Linux 5.4.0-74-generic

Productivity

Overall LoC: the overall line of codes
Value LoC: the line of codes to obtain value information
Code Eff.: the ratio of the overall line of codes and the line of codes for tool analysis

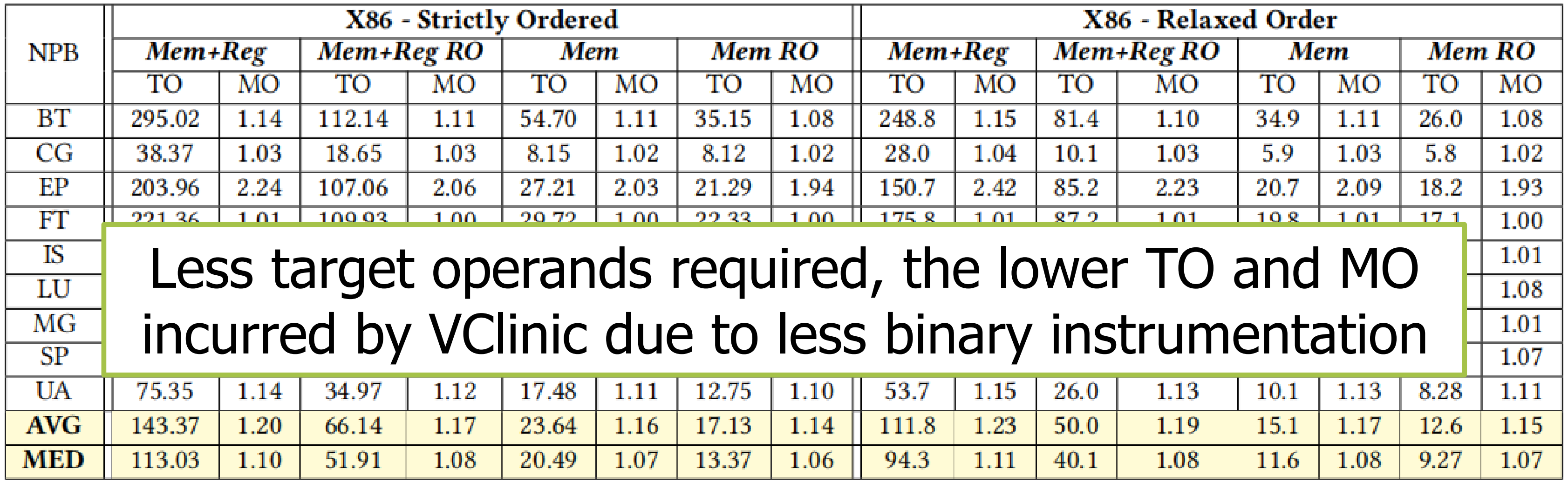
Target	***VClinic***			***Pin-based Framework***		
	Overall (LoC)	*Value (LoC)*	*Code Eff. (%)*	*Overall (Loc)*	*Value (Loc)*	*Code Eff. (%)*
Redundant Zero	1335	85	93.6	1859	634	65.9
Redundant Load	1033	87	91.6	1177	347	70.5
Dead Store	610	81	86.7	1782	297	83.3
Value Redundancy	1525	283	81.4	1902+683	736+119	66.9

- The overall LoC to implement these representative value profilers with VClinic is significantly less than the implementations with Pin.
- The Code Eff. of all evaluated value profilers implemented with VClinic are significantly higher than Pin.
- The representative tools implemented with VClinic are naturally applicable to both X86 and ARM platforms, which is not supported by Pin-based implementations.

Runtime and Memory Overhead

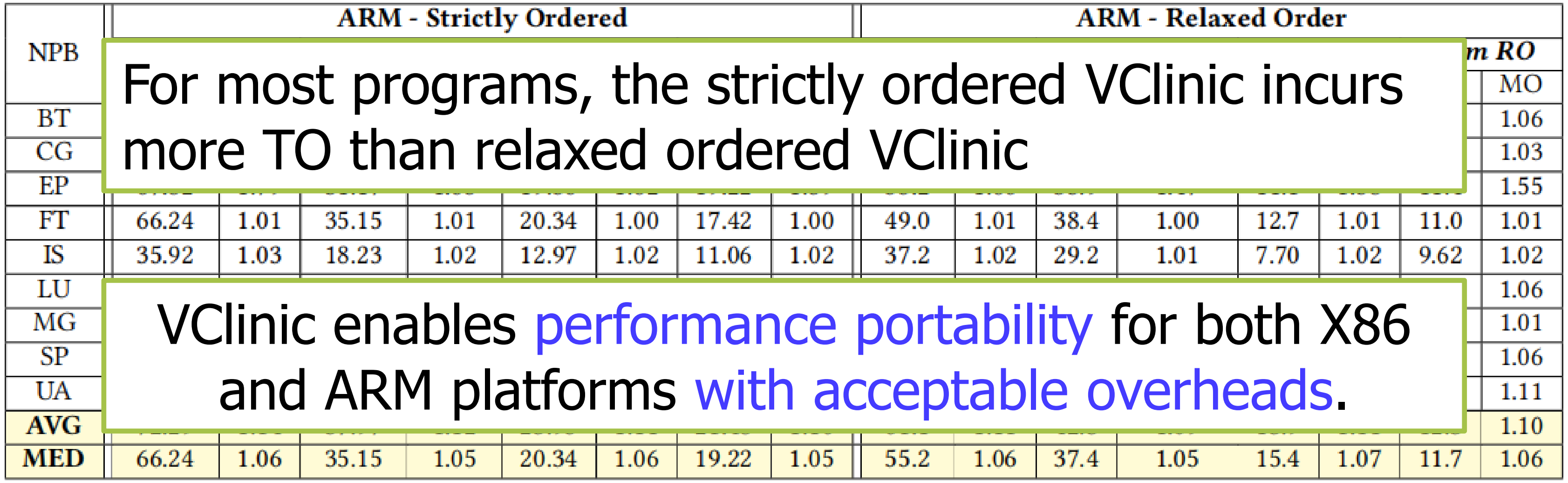
- **Runtime overhead (TO):** the execution time with VClinic divided by the native execution time.
- **Memory overhead (MO):** the peak memory consumption during execution with VClinic divided by the native peak memory consumption.

- **Mem+Reg:** profiling values of all memory and register operands;
- **Mem+Reg RO:** profiling values of all read-only memory and register operands;
- **Mem:** profiling values of all memory operands;
- **Mem RO:** profiling values of all read-only memory operands;



NPB	X86 - Strictly Ordered								X86 - Relaxed Order							
	Mem+Reg		*Mem+Reg RO*		*Mem*		*Mem RO*		*Mem+Reg*		*Mem+Reg RO*		*Mem*		*Mem RO*	
	TO	MO	TO	MO	TO	MO	TO	MO	TO	MO	TO	MO	TO	MO	TO	MO
BT	295.02	1.14	112.14	1.11	54.70	1.11	35.15	1.08	248.8	1.15	81.4	1.10	34.9	1.11	26.0	1.08
CG	38.37	1.03	18.65	1.03	8.15	1.02	8.12	1.02	28.0	1.04	10.1	1.03	5.9	1.03	5.8	1.02
EP	203.96	2.24	107.06	2.06	27.21	2.03	21.29	1.94	150.7	2.42	85.2	2.23	20.7	2.09	18.2	1.93
FT	221.36	1.01	109.93	1.00	29.72	1.00	22.33	1.00	175.8	1.01	87.2	1.01	19.8	1.01	17.1	1.00
IS	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1.01
LU	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1.08
MG	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1.01
SP	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1.07
UA	75.35	1.14	34.97	1.12	17.48	1.11	12.75	1.10	53.7	1.15	26.0	1.13	10.1	1.13	8.28	1.11
AVG	143.37	1.20	66.14	1.17	23.64	1.16	17.13	1.14	111.8	1.23	50.0	1.19	15.1	1.17	12.6	1.15
MED	113.03	1.10	51.91	1.08	20.49	1.07	13.37	1.06	94.3	1.11	40.1	1.08	11.6	1.08	9.27	1.07

Less target operands required, the lower TO and MO incurred by VClinic due to less binary instrumentation



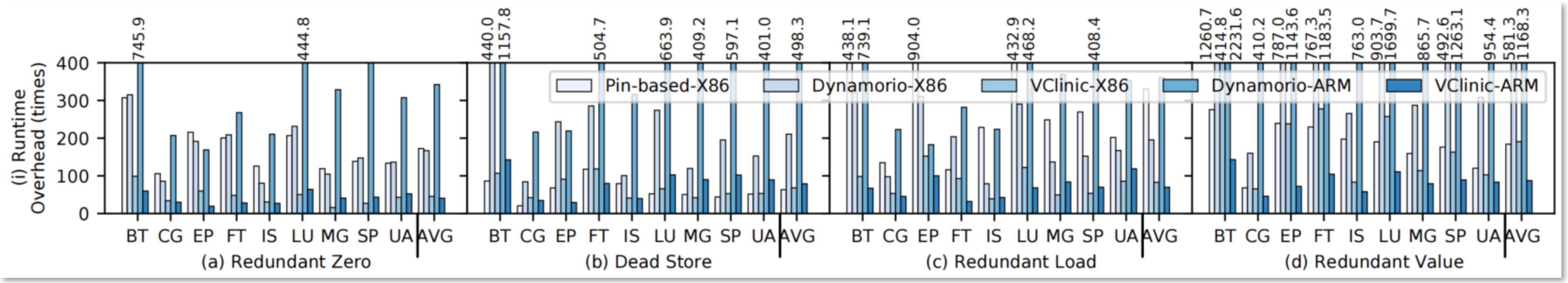
NPB	ARM - Strictly Ordered								ARM - Relaxed Order							
	[illegible]		[illegible]		[illegible]		[illegible]		[illegible]		[illegible]		[illegible]		*m RO*	
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	MO
BT	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1.06
CG	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1.03
EP	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1.55
FT	66.24	1.01	35.15	1.01	20.34	1.00	17.42	1.00	49.0	1.01	38.4	1.00	12.7	1.01	11.0	1.01
IS	35.92	1.03	18.23	1.02	12.97	1.02	11.06	1.02	37.2	1.02	29.2	1.01	7.70	1.02	9.62	1.02
LU	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1.06
MG	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1.01
SP	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1.06
UA	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1.11
AVG	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1.10
MED	66.24	1.06	35.15	1.05	20.34	1.06	19.22	1.05	55.2	1.06	37.4	1.05	15.4	1.07	11.7	1.06

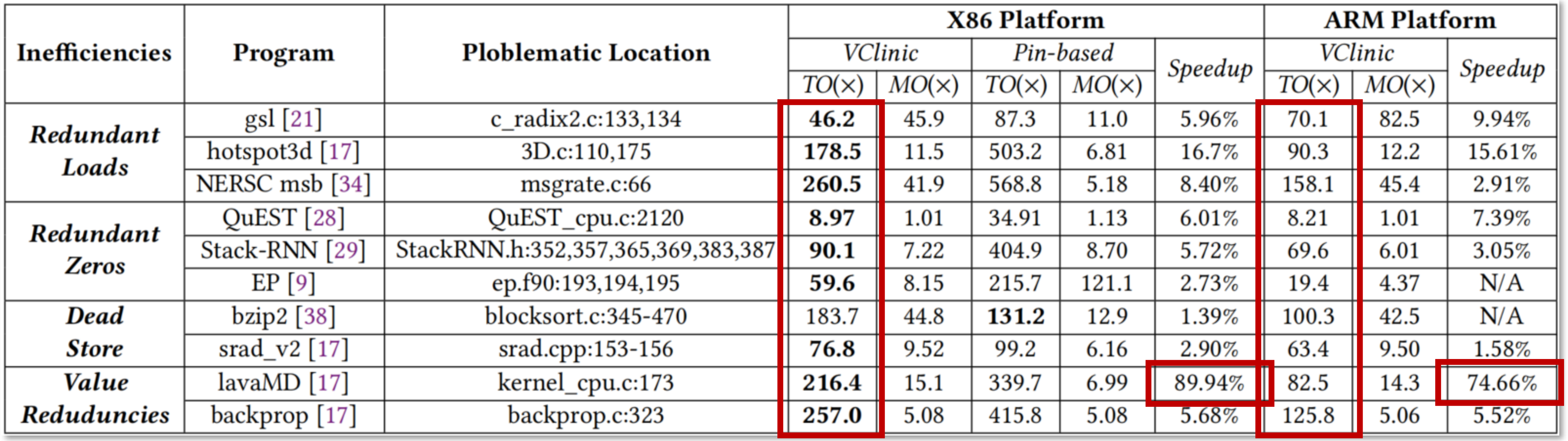
For most programs, the strictly ordered VClinic incurs more TO than relaxed ordered VClinic

VClinic enables performance portability for both X86 and ARM platforms with acceptable overheads.

Case Studies

- Representative value profilers can be efficiently constructed with *VClinic* on both X86 and ARM platforms with acceptable overhead.





Inefficiencies	**Program**	**Ploblematic Location**	**X86 Platform**					**ARM Platform**		
			VClinic		*Pin-based*		*Speedup*	*VClinic*		*Speedup*
			TO(×)	*MO*(×)	*TO*(×)	*MO*(×)		*TO*(×)	*MO*(×)	
Redundant Loads	gsl [21]	c_radix2.c:133,134	**46.2**	45.9	87.3	11.0	5.96%	70.1	82.5	9.94%
	hotspot3d [17]	3D.c:110,175	**178.5**	11.5	503.2	6.81	16.7%	90.3	12.2	15.61%
	NERSC msb [34]	msgrate.c:66	**260.5**	41.9	568.8	5.18	8.40%	158.1	45.4	2.91%
Redundant Zeros	QuEST [28]	QuEST_cpu.c:2120	**8.97**	1.01	34.91	1.13	6.01%	8.21	1.01	7.39%
	Stack-RNN [29]	StackRNN.h:352,357,365,369,383,387	**90.1**	7.22	404.9	8.70	5.72%	69.6	6.01	3.05%
	EP [9]	ep.f90:193,194,195	**59.6**	8.15	215.7	121.1	2.73%	19.4	4.37	N/A
Dead Store	bzip2 [38]	blocksort.c:345-470	183.7	44.8	**131.2**	12.9	1.39%	100.3	42.5	N/A
	srad_v2 [17]	srad.cpp:153-156	**76.8**	9.52	99.2	6.16	2.90%	63.4	9.50	1.58%
Value Redudancies	lavaMD [17]	kernel_cpu.c:173	**216.4**	15.1	339.7	6.99	89.94%	82.5	14.3	74.66%
	backprop [17]	backprop.c:323	**257.0**	5.08	415.8	5.08	5.68%	125.8	5.06	5.52%

For more details, please refer to our paper and API docs.

You X, Yang H, Lei K, et al. VClinic: A Portable and Efficient Framework for Fine-grained Value Profilers[C]//ASPLOS23: ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2023: 892-904.

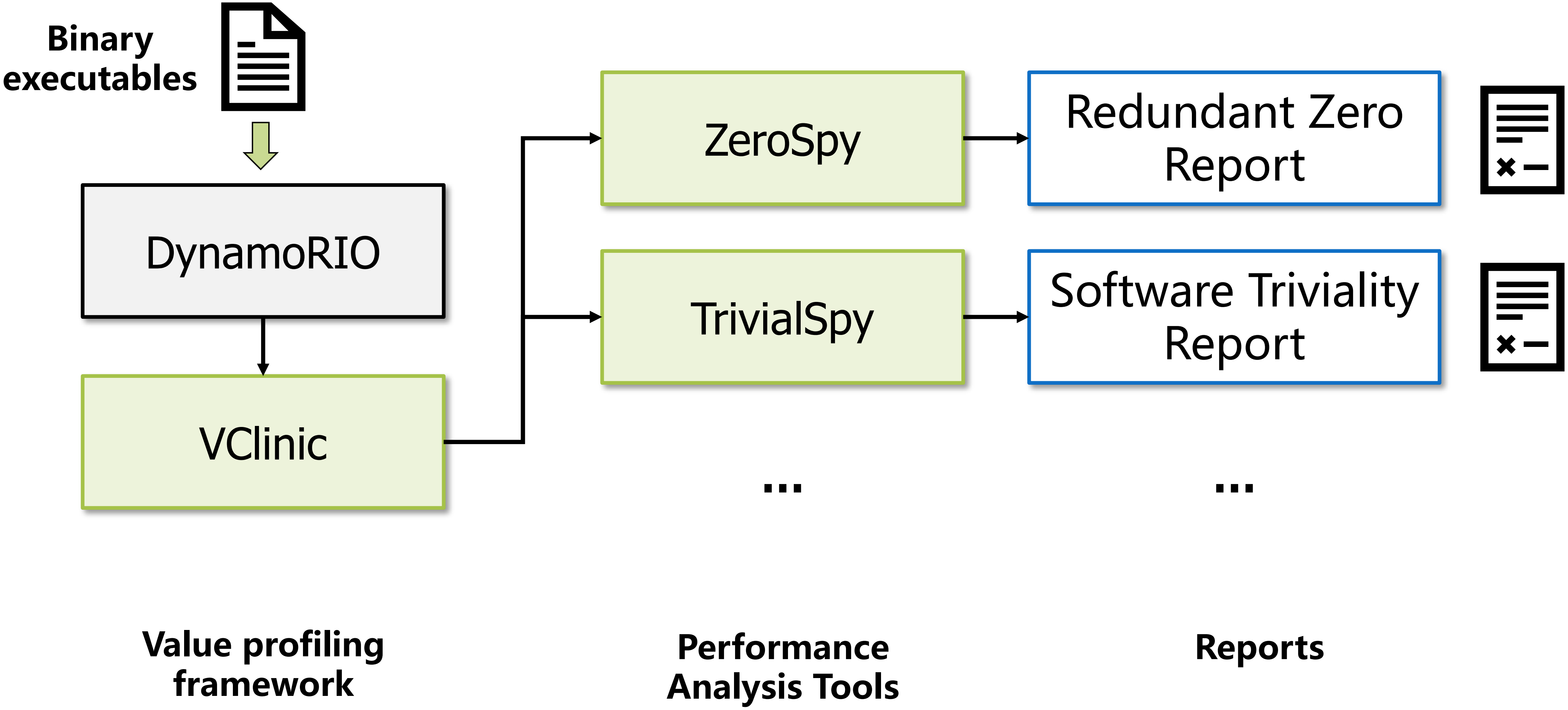
VClinic is open-source: https://github.com/VClinic/VClinic

```
bool REDSPY_FILTER_OPND ( opnd_t opnd , vprofile_src_t opmask ) {
        uint32_t mask1 =( ANY_DATA_TYPE | GPR_REGISTER | SIMD_REGISTER |
                WRITE | AFTER ) ;
        uint32_t mask2 =( ANY_DATA_TYPE | MEMORY | WRITE | BEFORE | AFTER ) ;
        return (( mask1 & opmask ) == opmask ) ||(( mask2 & opmask ) == opmask ) ;
}
DR_EXPORT void dr_client_main (...) {
        vprofile_init ( INSTR_FILTER ,0 ,0 ,0 , VPROFILE_COLLECT_CCT ) ;
        vtrace = vprofile_allocate_trace ( VPROFILE_TRACE_VAL_CCT_ADDR |
                VPROFILE_TRACE_BEFORE_WRITE |
                VPROFILE_TRACE_STRICTLY_ORDERED ) ;
        uint32_t opnd_mask =( ANY_DATA_TYPE | GPR_REGISTER | SIMD_REGISTER |
                MEMORY | WRITE | BEFORE | AFTER ) ;
        vprofile_register_trace_cb ( vtrace , REDSPY_FILTER_OPND ,
                opnd_mask , ANY , trace_update_cb ) ;
        ...
}
```

Outline

- Introduction
- Design Principles & Operand-centric Two-level Designs
- Methodology & Implementation
- Evaluation
- Hand-on Tutorial
  - Overview of VClinic value profiling toolkit
  - Installation & Development Guidance
  - Developing with VClinic – Zero Byte Statistics (with CCT)
  - Developing with VClinic – Redundant Load Profiler

Overview of VClinic value profiling toolkit



Installation: General Guidance

- Install from Source code
    - Dependencies: git, gcc, g++, make, cmake>=3.20
    - Source:
      ```
      git clone --recursive https://github.com/VClinic/VClinic.git
      ```
    - Compile and install:
      ```
      cd VClinic && ./build.sh
      ```
    - Configure:
      ```
      export DRRUN=`pwd`/build/bin64/drrun
      ```

- After installation, one can use the built-in value profilers in VClinic to analyze the target program

```
$DRRUN –t <CLINET TOOL> -- <EXE> <ARGS>
```

- API documents: **https://VClinic.readthedocs.io/en/latest/api.html**

Installation: For HPC Cluster at CNIC

- Install from Source code
  - The dependencies are already automatically loaded via environment module
  - Change directory to provided work directory: `cd /path/to/workdir`
  - Source: `cp -r /public/home/buaa_hipo/shared_folder/VClinic ./`
  - Compile and install: `cd VClinic && ./build.sh`
  - Configure: `` export DRRUN=`pwd`/build/bin64/drrun ``

- For executing program with VClinic client tool, we provide a template slurm job script:
  - **/public/home/buaa_hipo/shared_folder/slurm-template.sh**
  - By executing the script with **bash**, the script can submit a slurm job with **sbatch** command
  - Details in the following hand-on tutorial cases

~5 min

Slides is available at tutorial home page:
https://buaa-hipo.github.io/vprofiler-tutorial-cluster24/

VClinic Development Guidance – Overview

- Developing a new client, we need 5 steps:
- 1. Create a new client source code folder in src/clients/

```
cd VClinic && mkdir -p src/clients/YOUR_CLINET_TOOL/
```

- 2. Create the CMakeLists.txt for cmake configuration;
- 3. Implement your value profiler;
- 4. Re-compile VClinic to generate the newly implemented tool;
- 5. Use the developed tool for profiling!

VClinic Development Guidance – Code Skelton

Header files

```
#include "dr_api.h"
#include "vprofile.h"
vtrace_t* vtrace;
```

Estimate the filled in slots for detecting potential trace buffer overflow

```
bool
VPROFILE_FILTER_OPND(opnd_t opnd, vprofile_src_t
opmask) {
    uint32_t user_mask = (...);
    return ((user_mask & opmask) == opmask);
}
```

The update callback for user-specified record action

```
void update(val_info_t *info) {...}

static void
ClientInit(int argc, const char *argv[]) {...}

static void
ClientExit(void)
{
    vprofile_unregister_trace(vtrace);
    vprofile_exit();
}
```

The finalization callback when client exits (i.e., target program terminates)

```
DR_EXPORT void
dr_client_main(client_id_t id, int argc, const
char *argv[])
{   ...
    vprofile_init(VPROFILE_FILTER_ALL_INSTR,
NULL, NULL, NULL, VPROFILE_DEFAULT);
    vtrace = vprofile_allocate_trace(...);

    uint32_t opnd_mask = ...;
    vprofile_register_trace_template_cb(vtrace,
        VPROFILE_FILTER_OPND,
        opnd_mask,
        update);

    dr_register_exit_event(ClientExit);
}
```

Client main function (i.e., tool entry)

Reference code skelton location:
src/clients/vprofile_mem_and_reg/

VClinic Development Guidance – Trace allocation

```
DR_EXPORT void
dr_client_main(client_id_t id, int argc, const
char *argv[])
{   ...
    vprofile_init(VPROFILE_FILTER_ALL_INSTR,
NULL, NULL, NULL, VPROFILE_DEFAULT);
    vtrace = vprofile_allocate_trace(...);

    uint32_t opnd_mask = ...;
    vprofile_register_trace_templ
        VPROFILE_FILTER_OPND,
        opnd_mask,
        update);

    dr_register_exit_event(Client
}
```

Allocate trace with configurations of interested operand attributes
(VPROFILE_TRACE_DEFAULT is an alias of VPROFILE_TRACE_VALUE)

Configurable enumerate value	Explanation
VPROFILE_TRACE_VALUE	Trace target operand values
VPROFILE_TRACE_ADDR	Trace target address of memory operands
VPROFILE_TRACE_CCT	Trace calling context
VPROFILE_TRACE_INFO	Trace user-defined annotation
VPROFILE_TRACE_STRICTLY_ORDER	Tracing is strictly ordered
VPROFILE_TRACE_REG_IN_MEMREF	Trace register in memory operands
VPROFILE_TRACE_BEFORE_WRITE	Trace destinated memory operands before write operations

VClinic Development Guidance – Operand Mask

```
DR_EXPORT void
dr_client_main(client_id_t id, int argc, const
char *argv[])
{   ...
    vprofile_init(VPROFILE_FILTER_ALL_INSTR,
NULL, NULL, NULL, VPROFILE_DEFAULT);
    vtrace = vprofile_allocate_trace(...);

    uint32_t opnd_mask = ...;
    vprofile_register_trace_template_cb(vtrace,
        VPROFILE_FILTER_OPND,
        opnd_mask,
        update);

    dr_register_exit_event(ClientExit);
}
```

Mask for interested operand types and only operands within the mask will be further processed

(all configurations can be combined with OR)

Configurable enumerate value (partial)	
REGISTER	READ
PC	WRITE
MEMORY	BEFORE
IMMEDIATE	AFTER
IS_INTEGER	IS_FLOATING

VClinic Development Guidance – Operand Filter

```
#include "dr_api.h"
#include "vprofile.h"
vtrace_t* vtrace;
bool
VPROFILE_FILTER_OPND(opnd_t opnd, vprofile_src_t
opmask) {
    uint32_t user_mask = (...);
    return ((user_mask & opmask) == opmask);
}

void update(val_info_t *info) {...}

static void
ClientInit(int argc, const char *argv[]) {...}

static void
ClientExit(void)
{
    vprofile_unregister_trace(vtrace);
    vprofile_exit();
}
```

Estimate the filled in slots for detecting potential trace buffer overflow

Only the operands with *true* return value will be traced for further record actions (i.e., *update* callback)

(*opmask* is the extracted operand mask of candidate operand *opnd*)

Configurable enumerate value (partial)	
REGISTER	READ
PC	WRITE
MEMORY	BEFORE
IMMEDIATE	AFTER
IS_INTEGER	IS_FLOATING

VClinic Development Guidance – Update callback

```
#include "dr_api.h"
#include "vprofile.h"
vtrace_t* vtrace;
bool
VPROFILE_FILTER_OPND(opnd_t opnd, vprofile_src_t
opmask) {
    uint32_t user_mask = (...);
    return ((user_mask & opmask) == opmask);
}
```

The update callback for user-specified record action

```
void update(val_info_t *info) {...}

static void
ClientInit(int argc, const char *argv[]) {...}

static void
ClientExit(void)
{
    vprofile_unregister_trace(vtrace);
    vprofile_exit();
}
```

User-defined record action can be implemented in *update* callback with the extracted record in format of *val_info_t* (the untraced operand attributes are undefined)

This callback should be thread-safe: it may be called in parallel for multithreaded programs

val_info_t		
addr	Uint64_t	Address/Register identifier
type	Uint32_t	Operand type (`vprofile_src_t`)
ctxt_hndl	Int32_t	Calling context handler
val	Void*	Runtime value of the operand
info	Void*	User-defined annotation
Size	Uint8_t	Total length in byte
Esize	Uint8_t	Element length in byte
Is_float	bool	Whether the operand is a floating point value

Developing with VClinic – Zero Byte Statistics

- Now try to implement a zero byte statistic profiler ***zerobyte*** with VClinic!
  - A brief simplification of ZeroSpy tool.
- Target:
  - How many **zero bytes** are loaded from **memory**
    - Hint: we are interested in memory load operands
    - Reference code skelton: src/clients/vprofile_memory_read
    - Analysis should be implemented in ***update*** callback
  - Print the zero byte statistics at the end of target program execution.
    - Report **the total number of zero bytes** and **loaded bytes** from memory
    - Report generation should be implemented in ***ClientExit*** callback

API Reference:
https://VClinic.readthedocs.io/en/latest/api.html

~10 min

Developing with VClinic – Zero Byte Statistics

- Prepare source code directory:

```
cd VClinic && mkdir -p src/clients/zerobyte
```

- Modify CMake configuration according to the existing reference implementations

```
cp src/clients/zerospy/CMakeLists.txt src/clients/zerobyte/
sed -i “s/zerospy/zerobyte” CMakeLists.txt
```

- Implement tool with VClinic:

```
vim src/clients/zerobyte/zerobyte.cpp
```

- 1. Add corresponding include header files and variable definitions:

```
#include <unordered_map>
#include "vprofile.h"
using namespace std;

vtrace_t* vtrace;
uint64_t grandTotBytesLoad = 0;
uint64_t grandTotBytesRedLoad = 0;
```

Developing with VClinic – Zero Byte Statistics

- 2. implement handler to process each value record

```
void trace_update_cb(val_info_t *info) {
    uint8_t *val = (uint8_t*)info->val;
    int size = info->size;
    uint64_t red=0;
    for(int i=0; i<size; ++i) {
        red += (val[i]==0)?1:0;
    }
    __sync_fetch_and_add(&grandTotBytesRedLoad,red);
    __sync_fetch_and_add(&grandTotBytesLoad,size);
}
```

We implement analysis and recording with thread-safe atomic operations

Developing with VClinic – Zero Byte Statistics

- 3. implement the handler function when tools are loaded and unloaded.

```
static void
ClientInit(int argc, const char *argv[]) {}

static void
ClientExit(void)
{
    dr_fprintf(STDOUT, "\n#Redundant Read:");
    dr_fprintf(STDOUT, "\nTotalBytesLoad: %lu \n",grandTotBytesLoad);
    dr_fprintf(STDOUT, "\nRedundantBytesLoad: %lu %.2f\n",grandTotBytesRedLoad,
grandTotBytesRedLoad * 100.0/grandTotBytesLoad);
    vprofile_unregister_trace(vtrace);
    vprofile_exit();
}
```

Implementation of report generation

Developing with VClinic – Zero Byte Statistics

- 4. implement the operand and instruction filter

We interested in values of MEMORY READ operands with ANY_DATA_TYPE BEFORE memory read operation

```
// We only interest in memory loads
bool
VPROFILE_FILTER_OPND(opnd_t opnd, vprofile_src_t opmask) {
    uint32_t user_mask = ANY_DATA_TYPE | MEMORY | READ | BEFORE;
    return ((user_mask & opmask) == opmask);
}

bool
filter_read_mem_access_instr(instr_t *instr)
{
    return instr_reads_memory(instr) && !instr_is_prefetch(instr);
}

#define FILTER_READ_MEM_ACCESS_INSTR filter_read_mem_access_instr
```

Developing with VClinic – Zero Byte Statistics

- 5. implementation of the main analysis function

We interested in ***values*** of MEMORY READ operands with ANY_DATA_TYPE BEFORE memory read operation

```
#ifdef __cplusplus
extern "C" {
#endif
DR_EXPORT void dr_client_main(client_id_t id, int argc, const char *argv[]) {
    dr_set_client_name("DynamoRIO Client ‘zerobytes’ ", "http://dynamorio.org/issues");
    ClientInit(argc, argv);
    dr_register_exit_event(ClientExit);
    vprofile_init(FILTER_READ_MEM_ACCESS_INSTR, NULL, NULL, NULL, VPROFILE_DEFAULT);
    vtrace = vprofile_allocate_trace(VPROFILE_TRACE_VALUE);
    uint32_t opnd_mask = ANY_DATA_TYPE | MEMORY | READ | BEFORE;
    vprofile_register_trace_cb(vtrace, VPROFILE_FILTER_OPND, opnd_mask, ANY, trace_update_cb);
}
#ifdef __cplusplus
}
#endif
```

Developing with VClinic – Zero Byte Statistics

- **Total LoC: ~72 lines without any detailed implementations of instrumentation**
- Re-compile VClinic to generate the newly implemented zerobyte tool

```
./build.sh
```

- Compiled zerobyte tool can be used via command line:

```
$DRRUN -t zerobyte -- <EXE> <ARGS>
```

- On CNIC cluster, we need to submit a slurm job for program execution (use provided template):

```
NTASKS=1
JOBNAME=“JOBNAME"
DRRUN="`pwd`/VClinic/build/bin64/drrun"
TOOL=“zerobyte"
# TARGET command for profiling: CMD="<EXE> <ARGS>"
CMD=“<EXE> <ARGS>"
mkdir -p log
echo "START $JOBNAME WITH NTASK=$NTASKS "
nowdate=$(date +%Y_%m_%d_%H_%M_%S)
echo $nowdate
sbatch << END
```

```
#!/bin/bash
#SBATCH -J $JOBNAME
#SBATCH -o log/$JOBNAME-$NTASKS-%j-$nowdate.log
#SBATCH -e log/$JOBNAME-$NTASKS-%j-$nowdate.err
#SBATCH -p test
#SBATCH --cpus-per-task=16
#SBATCH --ntasks-per-node=1
#SBATCH -n $NTASKS
# Your SCRIPT commands
time $DRRUN -t $TOOL -- $CMD
END
```

Developing with VClinic – Zero Byte Statistics

- For instance, use zerobyte tool to detect the zero bytes loaded in backprop program:

```
cp -r /public/home/buaa_hipo/shared_folder/backprop ./
pushd backprop && make && popd
```

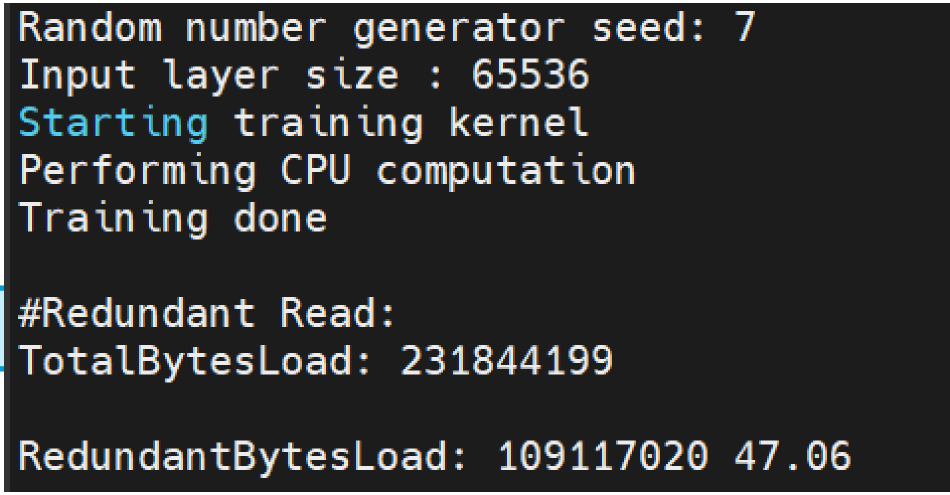
- On CNIC cluster, we need to submit a slurm job for program execution (use provided template):

```
cp /public/home/buaa_hipo/shared_folder/slurm-template.sh ./slurm-vclinic.sh
```

  - JOBNAME="vclinic-zerobyte"
  - TOOL="zerobyte"
  - CMD="`pwd`/backprop/backprop 65536"
- Execute backprop program with developed ZeroByte tool:

```
bash ./slurm-vclinic.sh
```

- Results in log/xxx.out

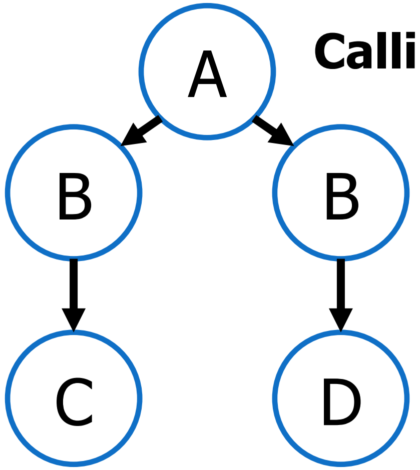


```
Random number generator seed: 7
Input layer size : 65536
Starting training kernel
Performing CPU computation
Training done

#Redundant Read:
TotalBytesLoad: 231844199

RedundantBytesLoad: 109117020 47.06
```

Developing with VClinic – Calling Context Attribution

- Background: What is Calling Context?
  - Each node is a calling context
- VClinic can collect CCT with DrCCTLib and attribute calling context to collected operands.

```
void A() {
    B(1);
    B(-1);
}
void B(int x) {
    if (x>0) { C(); }
    else     { D(); }
}
```



Calling Context Tree

- **Calling context is useful for actionable guidance** of detected performance issues

Enable CCT collection by adding VPROFILE_COLLECT_CCT for *vprofile_init* and adding VPROFILE_TRACE_CCT for trace allocation.

The collected calling context handler is located in *ctxt_hndl* attribute in record template, which can be further processed with DrCCTLib. For instance, the calling context can be printed with debug information via DrCCTLib API *drcctlib_print_backtrace*

Documents of DrCCTLib: https://drcctprof.readthedocs.io/en/latest/

Developing with VClinic – Zero Byte Statistics with CCT

- **Advanced Development:** Based on the previous implementation of ZeroByte tool, try to implement a ZeroByte_CCT tool to count loaded the zero bytes from memory within each calling context.
- **Report the Top 10 with corresponding metrics and calling context attributions**
- **Hints:**
  - Enable CCT collection with VPROFILE_COLLECT_CCT for vprofile initialization (*vprofile_init*)
  - Enable CCT attribution with VPROFILE_TRACE_CCT for trace allocation.
  - The collected calling context handler is located in *ctxt_hndl* attribute in *val_info_t*
  - Metrics can be merged by the value of *ctxt_hndl* (i.e., calling context handler)
  - The debug information of calling context can be printed by *drcctlib_print_backtrace* API call of DrCCTLib

~10 min

https://VClinic.readthedocs.io/en/latest/api.html

Developing with VClinic – Zero Byte Statistics with CCT

- Prepare source code directory:

```
cd VClinic && mkdir -p src/clients/zerobyte_cct
```

- Modify CMake configuration according to the existing reference implementations

```
cp src/clients/zerospy/CMakeLists.txt src/clients/zerobyte_cct/
sed -i “s/zerospy/zerobyte_cct” CMakeLists.txt
```

- Implement tool with VClinic:

```
vim src/clients/zerobyte_cct/zerobyte_cct.cpp
```

- 1. Add corresponding include header files and variable definitions:

```
#include <unordered_map>
#include "vprofile.h“
#include "drcctlib.h"
#include <list>
using namespace std;

vtrace_t* vtrace;
uint64_t grandTotBytesLoad = 0;
uint64_t grandTotBytesRedLoad = 0;
```

Developing with VClinic – Zero Byte Statistics with CCT

- 2. implement handler to process each value record and **merge statistics by calling context**

```
std::unordered_map<uint64_t, std::pair<uint64_t, uint64_t> > redLoadCCT;
static void* gLock;
void trace_update_cb(val_info_t *info) {
    ...
    dr_mutex_lock(gLock);
    auto it = redLoadCCT.find(info->ctxt_hndl);
    if (it==redLoadCCT.end()) {
        redLoadCCT[info->ctxt_hndl] = std::make_pair(red, size);
    } else {
        it->second.first += red;
        it->second.second+= size;
    }
    dr_mutex_unlock(gLock);
}
```

Developing with VClinic – Zero Byte Statistics with CCT

- 3. implement the handler function when tools are loaded and unloaded.

```
static void
ClientInit(int argc, const char *argv[]) { gLock = dr_mutex_create(); }

static void
ClientExit(void)
{
    dr_fprintf(STDOUT, "\n#Redundant Read:");
    dr_fprintf(STDOUT, "\nTotalBytesLoad: %lu \n",grandTotBytesLoad);
    dr_fprintf(STDOUT, "\nRedundantBytesLoad: %lu %.2f\n",grandTotBytesRedLoad,
grandTotBytesRedLoad * 100.0/grandTotBytesLoad);
    <...Add implementation of sorting and report generation>
    vprofile_unregister_trace(vtrace);
    vprofile_exit();
}
```

Full implementation refers to:

/public/home/buaa_hipo/shared_folder/Tutorial-VClinic/zerobyte_cct

Developing with VClinic – Zero Byte Statistics with CCT

- 4. implement the operand and instruction filter **(no changes)**

We interested in values of MEMORY READ operands with ANY_DATA_TYPE BEFORE memory read operation

```
// We only interest in memory loads
bool
VPROFILE_FILTER_OPND(opnd_t opnd, vprofile_src_t opmask) {
    uint32_t user_mask = ANY_DATA_TYPE | MEMORY | READ | BEFORE;
    return ((user_mask & opmask) == opmask);
}

bool
filter_read_mem_access_instr(instr_t *instr)
{
    return instr_reads_memory(instr) && !instr_is_prefetch(instr);
}

#define FILTER_READ_MEM_ACCESS_INSTR filter_read_mem_access_instr
```

Developing with VClinic – Zero Byte Statistics with CCT

- 5. implementation of the main function **with CCT collection and attribution enabled**

We interested in ***values*** of MEMORY READ operands with ANY_DATA_TYPE BEFORE memory read operation

```
#ifdef __cplusplus
extern "C" {
#endif
DR_EXPORT void dr_client_main(client_id_t id, int argc, const char *argv[]) {
    dr_set_client_name("DynamoRIO Client ‘zerobytes’ ", "http://dynamorio.org/issues");
    ClientInit(argc, argv);
    dr_register_exit_event(ClientExit);
    vprofile_init(FILTER_READ_MEM_ACCESS_INSTR, NULL, NULL, NULL, VPROFILE_COLLECT_CCT);
    vtrace = vprofile_allocate_trace(VPROFILE_TRACE_VAL_CCT);
    uint32_t opnd_mask = ANY_DATA_TYPE | MEMORY | READ | BEFORE;
    vprofile_register_trace_cb(vtrace, VPROFILE_FILTER_OPND, opnd_mask, ANY, trace_update_cb);
}
#ifdef __cplusplus
}
#endif
```

Developing with VClinic – Zero Byte Statistics with CCT

- **Total LoC: 2 lines modification to enable CCT collection and attribution**
- Re-compile VClinic to generate the newly implemented zerobyte_cct tool

```
./build.sh
```

- For instance, use zerobyte_cct tool to detect the zero bytes loaded in backprop program:
- On CNIC cluster, we submit a slurm job for program execution (modify provided template):

```
cp /public/home/buaa_hipo/shared_folder/slurm-template.sh ./slurm-vclinic-cct.sh
```

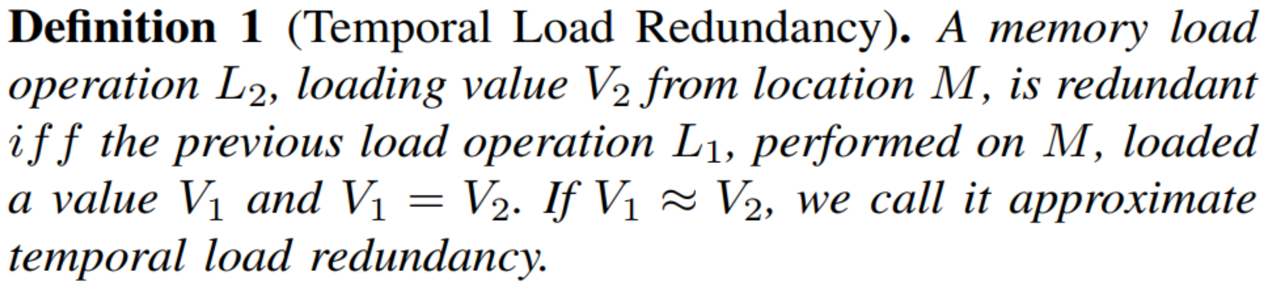
  - **JOBNAME="vclinic-zerobyte-cct"**
  - **TOOL="zerobyte_cct"**
  - **CMD="`pwd`/rodinia_3.1/openmp/backprop/backprop 65536"**
- Execute backprop program with developed ZeroByte with CCT tool:

```
bash ./slurm-vclinic-cct.sh
```

- Results in log/xxx.out

Developing with VClinic – Redundant Load Profiler

- **Let's detect a real-world inefficiencies: *Redundant Load* [1]**
  - **Redundant Load refers to two subsequent memory load operation from the same memory address load the same value**



Definition 1 (Temporal Load Redundancy). *A memory load operation L_2, loading value V_2 from location M, is redundant iff the previous load operation L_1, performed on M, loaded a value V_1 and $V_1 = V_2$. If $V_1 \approx V_2$, we call it approximate temporal load redundancy.*

- We can try to implement the (temporal) load redundancy in VClinic, namely *drload*

- Reference implementation: **src/clients/loadspy**

- When developed, you can try to analyze the redundant loads in hotspot3d in rodinia 3.1

---

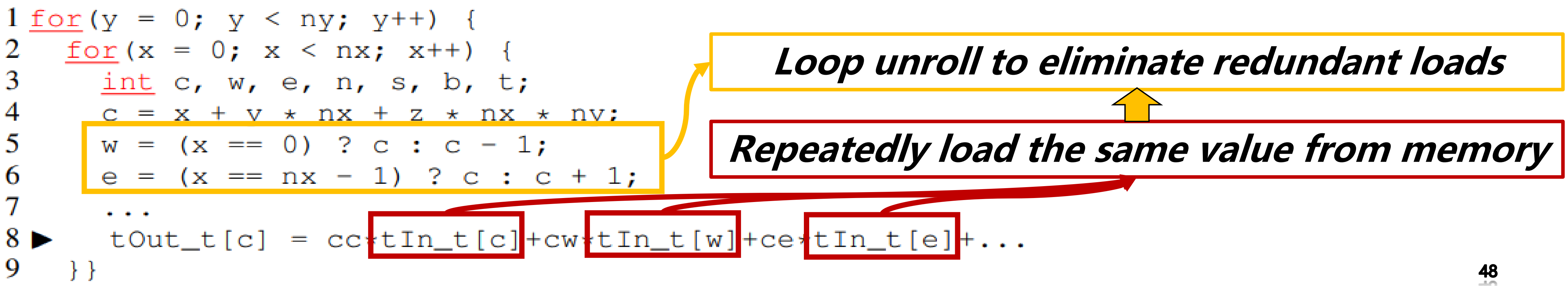
[1] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. 2019. Redundant loads: a software inefficiency indicator. In Proceedings of the 41st International Conference on Software Engineering (ICSE '19). IEEE Press, 982–993. https://doi.org/10.1109/ICSE.2019.00103

Developing with VClinic – Redundant Load Profiler

- Based on VClinic, we implement **Dr.Load** to detect redundant loads in hotspot3d program in rodinia 3.1 benchmark.
- Report **30.08%** **floating point redundant loads**, the problematic codes locate at ***3D.c:175***
- After optimization, gain **16.7%** speedup on X86, **15.61%** speedup on ARM.

```
======= (5.603124) % ======
#0   0x00000000004013a7 "movss  xmm3, dword ptr
[r15+rsi*4]" in computeTempOMP._omp_fn.0 at [3D.c:175]
#1   0x00007f781eac386b "call   r12" in <MISSING> at [3D.c:0]
#2   0x00007f781ea61603 "call   qword ptr [rax+0x00000640]" in
start_thread at [3D.c:0]
…
```

```
--------------------Redundant load with--------------------------
#0   0x000000000040136a "movss  xmm3, dword ptr
[r13+rax*4+0x00]" in computeTempOMP._omp_fn.0 at
[3D.c:175]
#1   0x00007f781eac386b "call   r12" in <MISSING> at [3D.c:0]
#2   0x00007f781ea61603 "call   qword ptr [rax+0x00000640]" in
start_thread at [3D.c:0]
…
```



```
for(y = 0; y < ny; y++) {
  for(x = 0; x < nx; x++) {
    int c, w, e, n, s, b, t;
    c = x + y * nx + z * nx * ny;
    w = (x == 0) ? c : c - 1;
    e = (x == nx - 1) ? c : c + 1;
    ...
    tOut_t[c] = cc*tIn_t[c]+cw*tIn_t[w]+ce*tIn_t[e]+...
}}
```

Loop unroll to eliminate redundant loads

Repeatedly load the same value from memory

Thanks! Q&A

Contact: youxin2015@buaa.edu.cn