Towards automatic characterization of microarchitectural behaviour for performance modelling of computing kernels: a comprehensive analysis of Cortex A72 and Intel architectures

PhD defense

Prepared under the supervision of Fabrice RASTELLO — CORSE team

Théophile BASTIAN

9th December 2024

Université Grenoble Alpes

Introduction (en français)

Le supercalculateur Fugaku

Le bâtiment du supercalculateur Fugaku

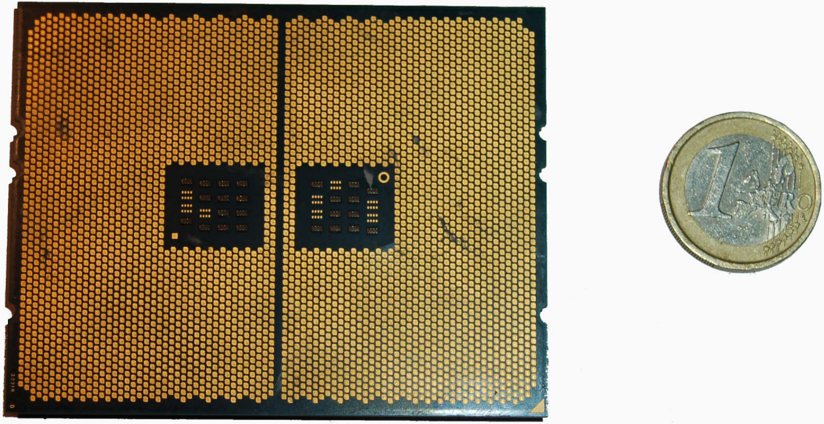
Une “baie” de Fugaku

Un serveur

© Supermicro

Un serveur

© Supermicro



Un processeur

À quoi ces supercalculateurs servent ?

- Calcul scientifique
  - Simulations de fluides (océans, aérodynamique, …)
  - Modélisations en chimie, biologie, …
  - Études du climat
- Prévisions météo

  → Météo-France : 29e plus puissant supercalculateur en 2020
- Développement de modèles IA
- …

Fugaku : 158,976 CPUs

Coût

- Un processeur : $\sim$ 100–1 000 €
- *Fugaku* : 1 milliard $

Consommation

- *Fugaku* : 30–40 MW
- $\sim$ 5 % d'un réacteur nucléaire

$\rightarrow$ gagner quelques % de performance, c'est très rentable !

Comment optimise-t-on ?

- Méthodes “classiques” déjà appliquées (algorithmique, parallélisation, …)
- Sections critiques : petit morceau de programme répété massivement
- Optimiser pour un processeur spécifique connu

Chercher où et pourquoi le processeur perd du temps.

Trois goulots d’étranglement étudiés

Trois goulots d'étranglement étudiés

Backend

- Les ouvriers de l'atelier
- Ouvriers surchargés : impossible d'aller plus vite
- Possiblement un seul métier

Trois goulots d'étranglement étudiés

Backend

- Les ouvriers de l'atelier
- Ouvriers surchargés : impossible d'aller plus vite
- Possiblement un seul métier

Frontend

- Manager
- Surchargé $\implies$ ouvriers sous-utilisés

Trois goulots d'étranglement étudiés

Backend

- Les ouvriers de l'atelier
- Ouvriers surchargés : impossible d'aller plus vite
- Possiblement un seul métier

Frontend

- Manager
- Surchargé $\Longrightarrow$ ouvriers sous-utilisés

Dépendances

- Tâches blocantes
- Tout l'atelier attend qu'un ouvrier ait fini

Analyseurs de code

- Analyser la situation :
  - Quel goulot d'étranglement ?
  - Où ?
  - Pourquoi ?

Analyseurs de code

- Analyser la situation :
  - Quel goulot d'étranglement ?
  - Où ?
  - Pourquoi ?
- CPU : “boite noire”
- $\sim$ 1 milliard instructions / seconde

Analyseurs de code

- Analyser la situation :
  - Quel goulot d'étranglement ?
  - Où ?
  - Pourquoi ?
- CPU : "boite noire"
- $\sim$ 1 milliard instructions / seconde

$\rightarrow$ On modélise pour analyser ! **"Analyseurs de code"**

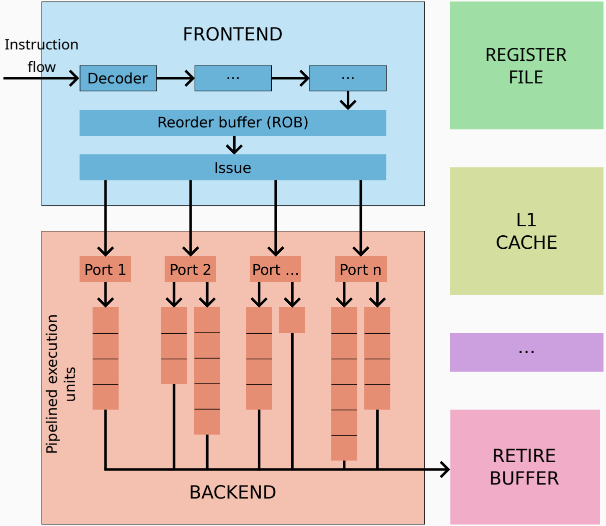
Summary

- Performance prediction for computational microkernels
- Approach based on bottlenecks
- Requires microarchitectural models

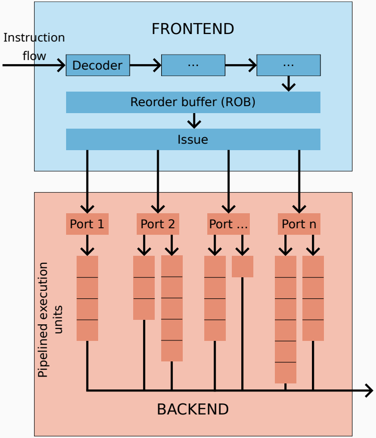
Works centered on developing parts of these models

Foundations

Bird’s eye view of a CPU



Possible bottlenecks



- Frontend: μOPs not issued fast enough
- Backend: saturated execution units
- Dependencies: computation is stalled waiting for previous results

What do we analyze?

Pieces of code referred as "microkernels":

- body of an (assumed) infinite loop;
- in steady-state;
- L1-resident (memory model is out of scope);
- straight-line code (branches assumed not taken).

```
loop:
  movsd (%rcx, %rax), %xmm0
  mulsd %xmm1, %xmm0
  addsd (%rdx, %rax), %xmm0
  movsd %xmm0, (%rdx, %rax)
  addq $8, %rax
  cmpq $0x2260, %rax
  jne loop
```

Reasonable hypotheses for the category of codes worth optimizing this way!

Code analyzers

- Predict performance of a microkernel
- Features microarchitectural models
- Most often static analyzers
- Predict at least the *reverse-throughput* of a kernel (cycles per iteration)
- May derive further useful metrics, e.g. bottlenecks, by inspecting their model at will

Existing code analyzers

Behavioural

- `IACA`: Intel, proprietary. Intel CPUs only.
- `llvm-mca`: `llvm` project, FOSS.
- `uiCA`, `uops.info`: academia. Intel CPUs only.

ML-based

- `Ithemal`: academia.

Behavioural tools are (to some extent) based on manually-made models!

Existing code analyzers

Behavioural

- `IACA`: Intel, proprietary. Intel CPUs only.
- `llvm-mca`: `llvm` project, FOSS.
- `uiCA`, `uops.info`: academia. Intel CPUs only.

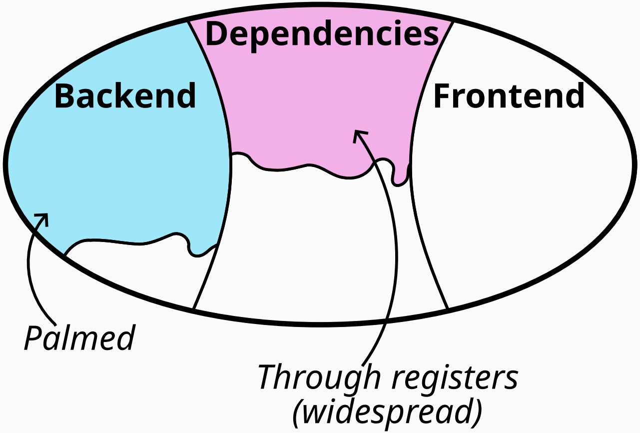
ML-based

- `Ithemal`: academia.

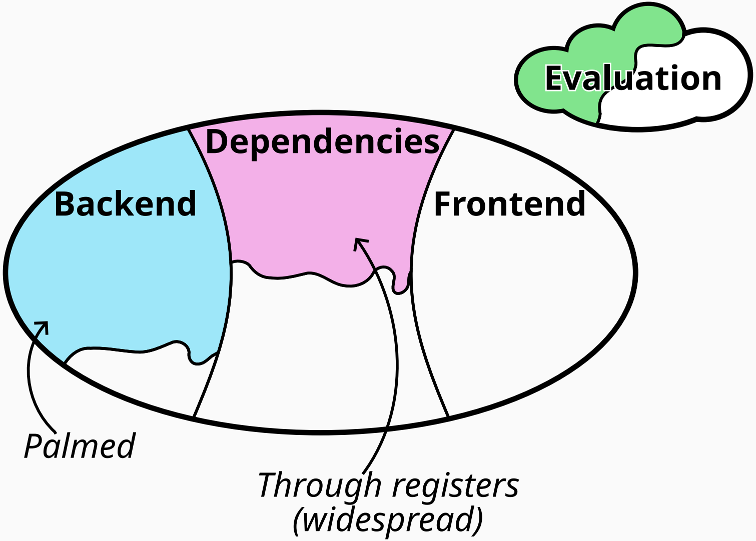
Behavioural tools are (to some extent) based on manually-made models!

Ambition: automated model generation.

When I started my PhD...



When I started my PhD...



`CesASMe`: evaluate and compare state-of-the-art code analyzers

Matrix multiplication:

```
loop:
    movsd (%rcx, %rax), %xmm0
    mulsd %xmm1, %xmm0
    addsd (%rdx, %rax), %xmm0
    movsd %xmm0, (%rdx, %rax)
    addq $8, %rax
    cmpq $0x2260, %rax
    jne loop
```

`llvm-mca`: 1.5 cycles/iter
`IACA`: 2.0 cycles/iter
`Ithemal`: 2.0 cycles/iter
`uiCA`: 3.0 cycles/iter

Which tool is correct?

Matrix multiplication:

```
loop:
    movsd (%rcx, %rax), %xmm0
    mulsd %xmm1, %xmm0
    addsd (%rdx, %rax), %xmm0
    movsd %xmm0, (%rdx, %rax)
    addq $8, %rax
    cmpq $0x2260, %rax
    jne loop
```

`llvm-mca`:	1.5 cycles/iter
`IACA`:	2.0 cycles/iter
`Ithemal`:	2.0 cycles/iter
`uiCA`:	3.0 cycles/iter

Which tool is correct?

We lack:

Context

Benchmarks

Generating benchmarks

We need benchmarks…

- representative
- infinite, L1-resident loops
- without control flow
- stressing diverse resources

- plenty of them

Generating benchmarks

We need benchmarks...

- representative — Polybench
- infinite, L1-resident loops
- without control flow
- stressing diverse resources

- plenty of them

Generating benchmarks

We need benchmarks…

- representative — Polybench
- infinite, L1-resident loops — “microkernelification” + verify
- without control flow
- stressing diverse resources

- plenty of them

Generating benchmarks

We need benchmarks…

• representative	Polybench
• infinite, L1-resident loops	"microkernelification" + verify
• without control flow	Polybench
• stressing diverse resources	
• plenty of them	

Generating benchmarks

We need benchmarks…

• representative	Polybench
• infinite, L1-resident loops	"microkernelification" + verify
• without control flow	Polybench
• stressing diverse resources	Polyhedral transformations
	+ unrolling + compiler options
• plenty of them	

Generating benchmarks

We need benchmarks...

• representative	Polybench
• infinite, L1-resident loops	"microkernelification" + verify
• without control flow	Polybench
• stressing diverse resources	Polyhedral transformations + unrolling + compiler options
• plenty of them	Even more of all those ↗

⇝ **yields 3500 benchmarks**

In-context baseline: lifting predictions

Consider instead $\mathcal{K}$ = full kernel, with its context

$\rightsquigarrow$ multiple basic blocks

In-context baseline: lifting predictions

Consider instead $\mathcal{K}$ = full kernel, with its context
$\rightsquigarrow$ multiple basic blocks

- Measure total kernel time **in context**
- Instrument full kernel $\mathcal{K}$: for each basic block, $\textsf{occur}(\text{bb})$
- For each tool
  - for each bb, $\textsf{prediction}(\text{bb})$
  - *lift* predictions:

$$\textsf{prediction}(\mathcal{K}) = \sum_{\text{bb} \in \mathcal{K}} \textsf{occur}(\text{bb}) \times \textsf{prediction}(\text{bb})$$

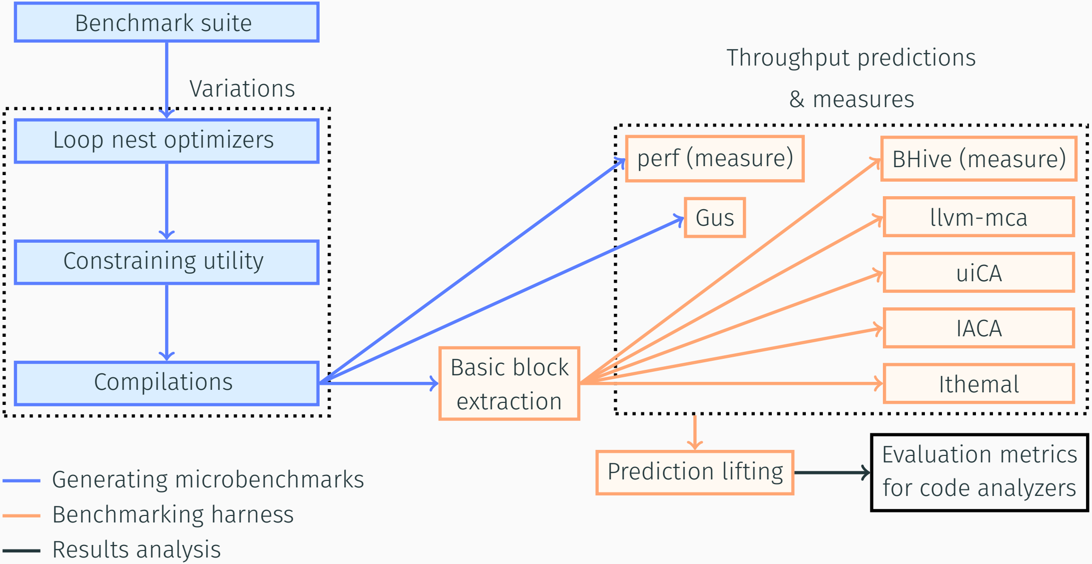
In-context baseline: lifting predictions

Consider instead $\mathcal{K}$ = full kernel, with its context
$\rightsquigarrow$ multiple basic blocks

- Measure total kernel time **in context**
- Instrument full kernel $\mathcal{K}$: for each basic block, $\mathsf{occur}(\mathrm{bb})$
- For each tool
  - for each bb, $\mathsf{prediction}(\mathrm{bb})$
  - *lift* predictions:

$$\mathsf{prediction}(\mathcal{K}) = \sum_{\mathrm{bb} \in \mathcal{K}} \mathsf{occur}(\mathrm{bb}) \times \mathsf{prediction}(\mathrm{bb})$$

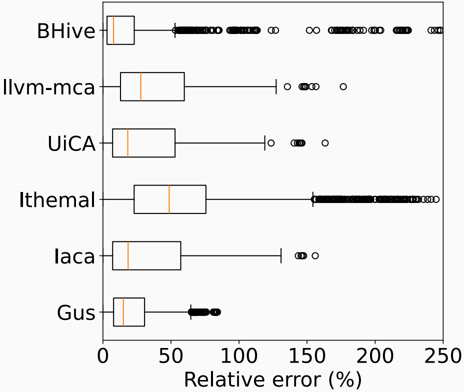
Now we have a baseline.



Benchmark suite
Variations
Loop nest optimizers
Constraining utility
Compilations
Basic block extraction
Throughput predictions & measures
perf (measure)
Gus
BHive (measure)
llvm-mca
uiCA
IACA
Ithemal
Prediction lifting
Evaluation metrics for code analyzers
Generating microbenchmarks
Benchmarking harness
Results analysis

First results (Intel Skylake on Grid5000)

$$\text{err} = \frac{|\text{predict} - \text{measure}|}{\text{measure}}$$

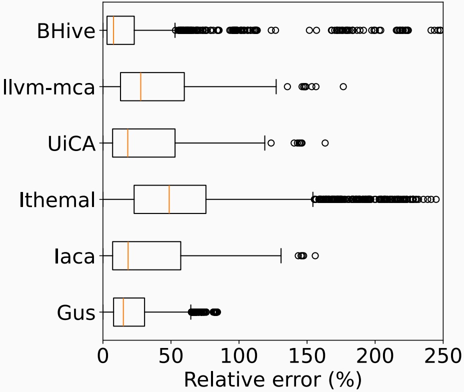


Outliers > 250 % trimmed

Associated table in supplementary material

First results (Intel Skylake on Grid5000)

$$\text{err} = \frac{|\text{predict} - \text{measure}|}{\text{measure}}$$



Outliers > 250 % trimmed

Associated table in supplementary material

Severely worse than previous evaluations!

Harness broken? Harder benchmarks? Previously undetected weaknesses?

Searching for areas of improvement

- Tools often wrong on the *same* rows
  - `llvm-mca`, `IACA` and `uiCA` share 80 % of their worst 30 %
- Often `-O1` rows

Crucial difference:

Bad

```
for(c3)
    tmp[c1] += A[c1][c3] * x[c3];
```

Good

```
for(c3)
    A[c1][c3] += u1[c1] * v1[c3]
               + u2[c1] * v2[c3];
```

Searching for areas of improvement

- Tools often wrong on the *same* rows
  - `llvm-mca`, `IACA` and `uiCA` share 80 % of their worst 30 %
- Often `-O1` rows

Crucial difference:

Bad: reduction

```
for(c3)
    tmp[c1] += A[c1][c3] * x[c3];
```

Good: map

```
for(c3)
    A[c1][c3] += u1[c1] * v1[c3]
               + u2[c1] * v2[c3];
```

Searching for areas of improvement

- Tools often wrong on the *same* rows
  - `llvm-mca`, `IACA` and `uiCA` share 80 % of their worst 30 %
- Often `-O1` rows

Crucial difference:

Bad: reduction

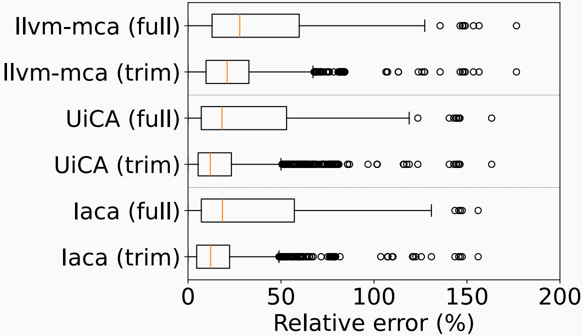
```
for(c3)
    tmp[c1] += A[c1][c3] * x[c3];
```

Good: map

```
for(c3)
    A[c1][c3] += u1[c1] * v1[c3]
               + u2[c1] * v2[c3];
```

Dependencies through memory!

Pruning memory-carried dependencies (Intel Skylake on Grid5000)



Outliers > 200 %

trimmed

Closer to expected results

`staticdeps`: static extraction of memory-carried dependencies

Dependencies, through registers

```
0:  mov  (%rax), %rcx
    ...
3:  add  %rcx, %rdx
```

- Track register writes
- Output dependency upon read

0 → 3 through `%rcx`

Dependencies, loop-carried

```
loop:
0:  add  %rcx, %rdx
    ...
3:  mov  (%rax), %rcx
6:  jmp loop
```

$\longrightarrow$

```
0:  add  %rcx, %rdx
    ...
3:  mov  (%rax), %rcx
0:  add  %rcx, %rdx
    ...
3:  mov  (%rax), %rcx
```

3 → 0 through `%rcx`, loop-carried

Dependencies, through memory

```
mov %r10, 4(%rax)
add $4, %rax
add (%rax), %rbx
```

- Through memory: indirections, arithmetics, …
- Requires comparison of arbitrary symbolic expressions
- Use randomness as a kind of hash table instead
- Loop-carried: luckily, ROB is finite and small

Dependencies, through memory

```
mov %r10, 4(%rax)
add $4, %rax
add (%rax), %rbx
```

- Through memory: indirections, arithmetics, ...
- Requires comparison of arbitrary symbolic expressions
- Use randomness as a kind of hash table instead
- Loop-carried: luckily, ROB is finite and small

Hypothesis: pointers from context do not alias.
Compilers prefer passing a single pointer.

The `staticdeps` algorithm

- Unroll kernel until $|\mathcal{K}| \geq |\text{ROB}| + |\mathcal{K}_0|$
- Simulate execution
- Unknown value (reg./mem.)? Sample uniformly in $0 \ldots 2^{64} - 1$ ("fresh")
- Compute arithmetics normally (overflow is fine)
- Float or unknown operands $\rightsquigarrow \bot$
- Upon write, remember from which instruction
- Upon read, if writer known, output dependency

An example: memoized Fibonacci sequence

```c
int fibo(int* F, int n) {
    for(int i=2; i <= n; ++i) {
        F[i] = F[i-1] + F[i-2];
    }
    return F[n];
}
```

→

```
0:    mov     (%rax),%edx
1:    add     0x4(%rax),%edx
2:    mov     %edx,0x8(%rax)
3:    add     $0x4,%rax
4:    cmp     %rcx,%rax
5:    jne     0
```

After instr	Registers %rax	%edx	Memory 100	104	108	112	116	Dep
Start	?	?	?	?	?	?	?	

Mem. read   Mem. write

```
0: mov   (%rax),%edx
1: add   0x4(%rax),%edx
2: mov   %edx,0x8(%rax)
3: add   $0x4,%rax
4: cmp   %rcx,%rax
5: jne   0
```

After instr	Registers %rax	%edx	Memory 100	104	108	112	116	Dep
Start	?	?	?	?	?	?	?	
0	100	200	200	?	?	?	?	

Mem. read   Mem. write

```
0: mov    (%rax),%edx
1: add    0x4(%rax),%edx
2: mov    %edx,0x8(%rax)
3: add    $0x4,%rax
4: cmp    %rcx,%rax
5: jne    0
```

After instr	Registers %rax	%edx	Memory 100	104	108	112	116	Dep
Start	?	?	?	?	?	?	?	
0	100	200	200	?	?	?	?	
1	100	376	200	176	?	?	?	

Mem. read   Mem. write

```
0: mov    (%rax),%edx
1: add    0x4(%rax),%edx
2: mov    %edx,0x8(%rax)
3: add    $0x4,%rax
4: cmp    %rcx,%rax
5: jne    0
```

After instr	Registers %rax	%edx	Memory 100	104	108	112	116	Dep
Start	?	?	?	?	?	?	?	
0	100	200	200	?	?	?	?	
1	100	376	200	176	?	?	?	
2	100	376	200	176	376	?	?	

Mem. read   Mem. write

```
0: mov   (%rax),%edx
1: add   0x4(%rax),%edx
2: mov   %edx,0x8(%rax)
3: add   $0x4,%rax
4: cmp   %rcx,%rax
5: jne   0
```

Mem. read | Mem. write

```
0: mov    (%rax),%edx
1: add    0x4(%rax),%edx
2: mov    %edx,0x8(%rax)
3: add    $0x4,%rax
4: cmp    %rcx,%rax
5: jne    0
```

After	Registers		Memory					Dep
instr	%rax	%edx	100	104	108	112	116	
Start	?	?	?	?	?	?	?	
0	100	200	200	?	?	?	?	
1	100	376	200	176	?	?	?	
2	100	376	200	176	376	?	?	
3	104	376	200	176	376	?	?	
4	104	376	200	176	376	?	?	

Mem. read | Mem. write

```
0: mov   (%rax),%edx
1: add   0x4(%rax),%edx
2: mov   %edx,0x8(%rax)
3: add   $0x4,%rax
4: cmp   %rcx,%rax
5: jne   0
```

After	Registers		Memory					Dep
instr	%rax	%edx	100	104	108	112	116	
Start	?	?	?	?	?	?	?	
0	100	200	200	?	?	?	?	
1	100	376	200	176	?	?	?	
2	100	376	200	176	376	?	?	
3	104	376	200	176	376	?	?	
4	104	376	200	176	376	?	?	
0	104	176	200	176	376	?	?	

Mem. read | Mem. write

```
0: mov  (%rax),%edx
1: add  0x4(%rax),%edx
2: mov  %edx,0x8(%rax)
3: add  $0x4,%rax
4: cmp  %rcx,%rax
5: jne  0
```

After instr	Registers %rax	%edx	Memory 100	104	108	112	116	Dep
Start	?	?	?	?	?	?	?	
0	100	200	200	?	?	?	?	
1	100	376	200	176	?	?	?	
2	100	376	200	176	376	?	?	
3	104	376	200	176	376	?	?	
4	104	376	200	176	376	?	?	
0	104	176	200	176	376	?	?	
1	104	552	200	176	376	?	?	-1,2 →

Mem. read   Mem. write

```
0: mov    (%rax),%edx
1: add    0x4(%rax),%edx
2: mov    %edx,0x8(%rax)
3: add    $0x4,%rax
4: cmp    %rcx,%rax
5: jne    0
```

After instr	Registers %rax	%edx	Memory 100	104	108	112	116	Dep
Start	?	?	?	?	?	?	?	
0	100	200	200	?	?	?	?	
1	100	376	200	176	?	?	?	
2	100	376	200	176	376	?	?	
3	104	376	200	176	376	?	?	
4	104	376	200	176	376	?	?	
0	104	176	200	176	376	?	?	
1	104	552	200	176	376	?	?	-1,2 →
2	104	552	200	176	376	552	?	

Mem. read   Mem. write

```
0: mov    (%rax),%edx
1: add    0x4(%rax),%edx
2: mov    %edx,0x8(%rax)
3: add    $0x4,%rax
4: cmp    %rcx,%rax
5: jne    0
```

After instr	Registers %rax	%edx	Memory 100	104	108	112	116	Dep
Start	?	?	?	?	?	?	?	
0	100	200	200	?	?	?	?	
1	100	376	200	176	?	?	?	
2	100	376	200	176	376	?	?	
3	104	376	200	176	376	?	?	
4	104	376	200	176	376	?	?	
0	104	176	200	176	376	?	?	
1	104	552	200	176	376	?	?	-1,2 →
2	104	552	200	176	376	552	?	
0	108	376	200	176	376	552	?	-2,2 →

Mem. read | Mem. write

```
0: mov   (%rax),%edx
1: add   0x4(%rax),%edx
2: mov   %edx,0x8(%rax)
3: add   $0x4,%rax
4: cmp   %rcx,%rax
5: jne   0
```

After	Registers		Memory					Dep
instr	%rax	%edx	100	104	108	112	116	
Start	?	?	?	?	?	?	?	
0	100	200	200	?	?	?	?	
1	100	376	200	176	?	?	?	
2	100	376	200	176	376	?	?	
3	104	376	200	176	376	?	?	
4	104	376	200	176	376	?	?	
0	104	176	200	176	376	?	?	
1	104	552	200	176	376	?	?	-1,2 →
2	104	552	200	176	376	552	?	
0	108	376	200	176	376	552	?	-2,2 →
1	108	928	200	176	376	552	?	-1,2 →

Mem. read	Mem. write

```
0: mov   (%rax),%edx
1: add   0x4(%rax),%edx
2: mov   %edx,0x8(%rax)
3: add   $0x4,%rax
4: cmp   %rcx,%rax
5: jne   0
```

After	Registers		Memory					Dep
instr	%rax	%edx	100	104	108	112	116	
Start	?	?	?	?	?	?	?	
0	100	200	200	?	?	?	?	
1	100	376	200	176	?	?	?	
2	100	376	200	176	376	?	?	
3	104	376	200	176	376	?	?	
4	104	376	200	176	376	?	?	
0	104	176	200	176	376	?	?	
1	104	552	200	176	376	?	?	-1,2 →
2	104	552	200	176	376	552	?	
0	108	376	200	176	376	552	?	-2,2 →
1	108	928	200	176	376	552	?	-1,2 →
2	108	928	200	176	376	552	928	

Practical implementation

- Python code
- Reads asm / elf / symbol in elf
- Disassembly: `capstone`
- Semantics: VEX (aka Valgrind)

$\rightsquigarrow$ fast; supports many architectures

Limitations

- Randomness may generate false positives
    - Very unlikely: 2^{64} vs. $\sim 10^4$
    - If needed, amplify (run twice)
- No false negatives caused by randomness, however
- Unaware of context: *assumes no pointers alias*
    - Intrinsic limitation of block-based code analyzers
    - Future works: information from
        - the compiler?
        - a light instrumentation pass?

Evaluation: coverage

- Baseline: instrumentation (extract deps at runtime)
- Filter *long-distance dependencies* ($> |\text{ROB}|$)
- On all `CesASMe` benchmarks

$$\text{cov}_u = \frac{|\text{found}|}{|\text{found}| + |\text{missed}|}$$

$$\text{cov}_w = \frac{\sum_{d \in \text{found}} \rho_d}{\sum_{d \in \text{found} \cup \text{missed}} \rho_d}$$

cov_u (%)	cov_w (%)
94.4	98.3

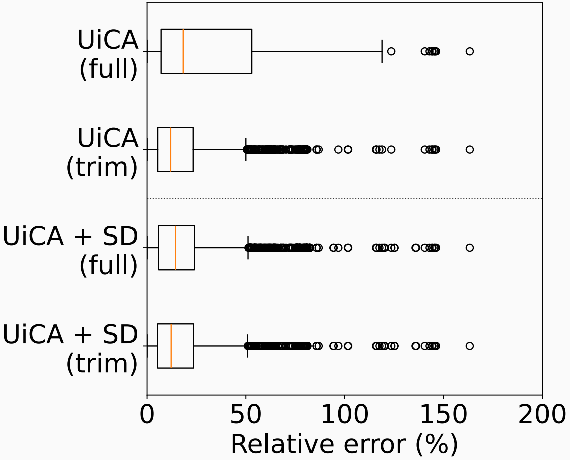
Evaluation: *points-to* analysis

- Quantify whether $\exists p, q \in$ context pointing to the same memory region ("points-to")
- Proxy: if $i_0 \rightarrow i_1$, then $q \in i_1$ aliases $p \in i_0$
- If $p = q$, we should catch it
- If not: either *long-distance* with $p = q$, or $p \neq q$!

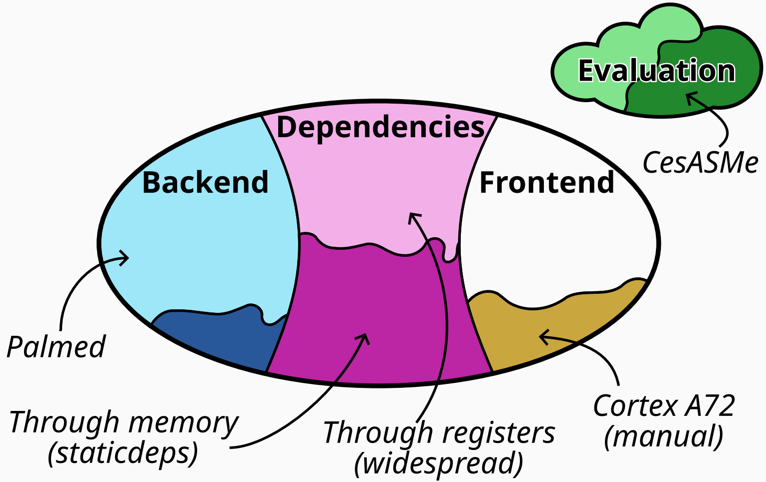
$\rightsquigarrow$ Keep long-distance dependencies; evaluate coverage on this

cov_u (%)	cov_w (%)
95.0	93.7

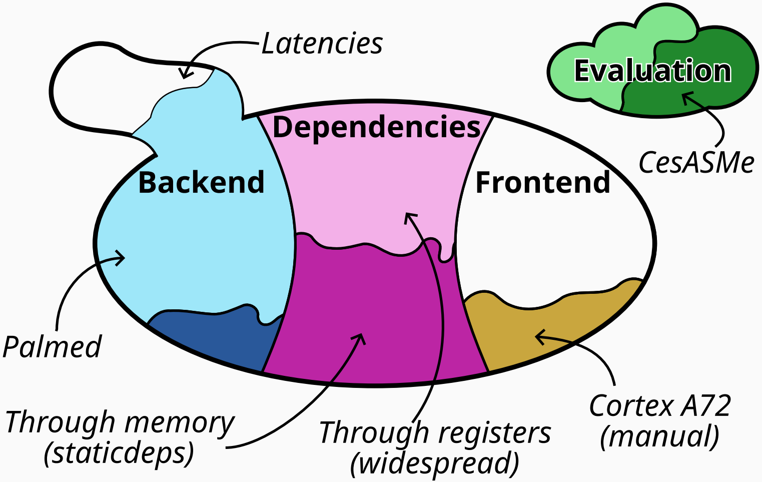
Evaluation: use in `uiCA`

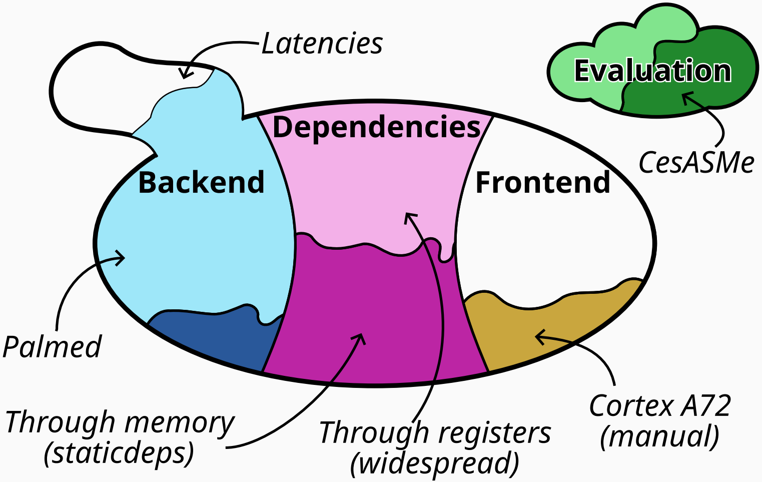


Wrapping up: the `A72` `combined` model



Evaluation
CesASMe
Dependencies
Backend
Frontend
Palmed
Through memory
(staticdeps)
Through registers
(widespread)
Cortex A72
(manual)


Latencies
Evaluation
Dependencies
Backend
Frontend
CesASMe
Palmed
Through memory
(staticdeps)
Through registers
(widespread)
Cortex A72
(manual)



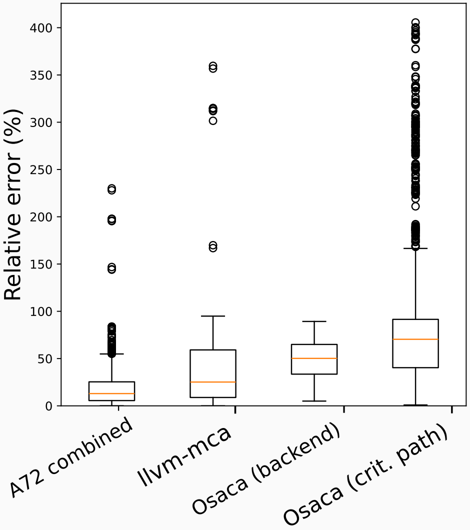
⇝ Let’s make a model for the Cortex A72!

From dependencies to cycles

- `staticdeps`: set to also report register dependencies.
- Unroll $\mathcal{K}$ to fill the ROB
- Build dependencies graph: edges are dependencies, weighted by source instruction latency (given by `Palmed`).
- Compute longest path, divide by repetitions of $\mathcal{K}$

$\rightsquigarrow$ **lower bound on execution time**

Combine frontend, backend, critical by taking the max


Relative error (%)
0
50
100
150
200
250
300
350
400
A72 combined
llvm-mca
Osaca (backend)
Osaca (crit. path)

Conclusion

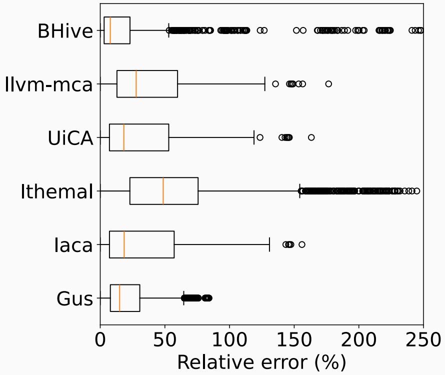
- `CesASMe`: a framework to faithfully compare code analyzers;
    - used to compare SotA analyzers
    - reveals dependencies through memory as clear weakness
- `staticdeps`: a static analyzer to extract dependencies, incl. through memory
- A manual frontend model for the Cortex A72 ARM processor
    - parametric model for future works on the frontend
    - partially automated
- A loosely combined model including those, outperforming (manual) SotA.

Questions?

Results (detailed versions)

CesASMe — Detailed first results

Bencher	Failures (%)	MAPE (%)	Median (%)	K_τ	Time (CPU·h)
BHive	37.20	27.95	23.01	0.81	1.37
llvm-mca	0.00	36.71	59.80	0.57	0.96
UiCA	0.00	29.59	52.99	0.58	2.12
Ithemal	0.00	57.04	75.69	0.39	0.38
Iaca	0.00	30.23	57.18	0.59	1.31
Gus	0.00	20.37	30.59	0.82	188.04



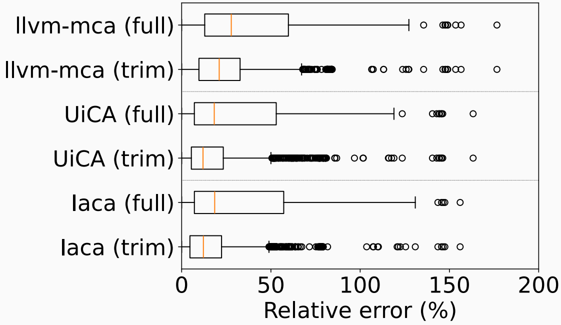
Outliers > 250 % trimmed

Severely worse than previous evaluations!

Harness broken? Harder benchmarks? Previously undetected weaknesses?

CesASMe — pruning memory-carried dependencies (detailed)

Bencher	Dataset	MAPE (%)	Median (%)	K_τ
llvm-mca	Full	36.71	59.80	0.57
	Trim	27.06	21.04	0.79
UiCA	Full	29.59	52.99	0.58
	Trim	18.42	11.96	0.80
Iaca	Full	30.23	57.18	0.59
	Trim	17.55	12.17	0.82

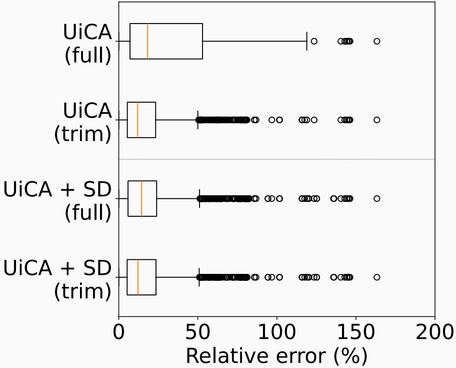


Outliers > 200 % trimmed

Closer to expected results

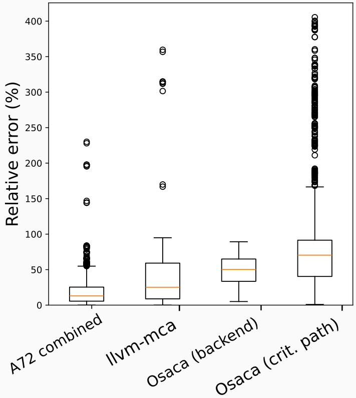
staticdeps: use in uiCA (detailed)

Dataset	Bencher	MAPE (%)	Median (%)	K_τ
Full	uiCA	29.59	18.26	0.58
	+ staticdeps	19.15	14.44	0.81
Trim	uiCA	18.42	11.96	0.80
	+ staticdeps	18.77	12.18	0.80



`A72 combined`: results (detailed)

Bencher	Fail (%)	MAPE (%)	Median (%)	Q1 (%)	Q3 (%)	K_τ
A72 combined	0.51	19.26	12.98	5.57	25.38	0.75
llvm-mca	0.06	32.60	25.17	8.84	59.16	0.69
Osaca (backend)	0.17	49.33	50.19	33.53	64.94	0.67
Osaca (crit. path)	0.17	84.02	70.39	40.37	91.47	0.24



Misc supplementary material

CesASMe and control flow

Straight-line code: hypothesis of code analysers, but also…

```
for(i) {
    if(A[i] % 2 == 0)
        A[i] *= 2;
    A[i] += B[i];
}
```

- If not taken: map
- If taken: dependency in `A[i]`!
- Performance varies depending on branch
- Performance strongly depends on input data

staticdeps: lack of context

Context-dependent stride

```
for(int i=0; i < n-k; ++i)
    A[i] += A[i+k];
```

↓

```
loop:
  mov   (%rax,%rdx,4),%ecx
  add   %ecx,(%rax)
  add   $0x4,%rax
  cmp   %rsi,%rax
  jne   loop
```

No dep found!

Graphs algorithms

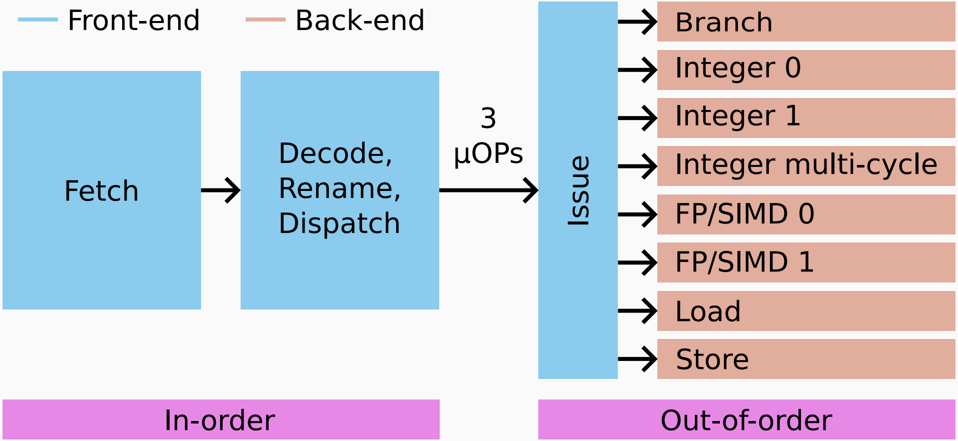
- Graphs: commonly represented as e.g.

```
struct Node {
  // ...
  vector<Node*> siblings;
};
```

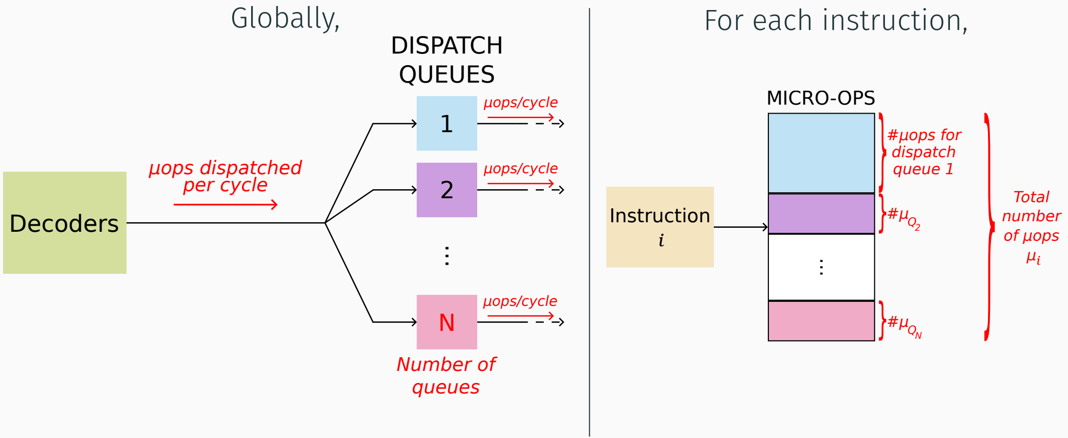
- Values of `siblings` will alias on purpose!
- …thus breaking `staticdeps`.

A frontend model for the Cortex A72

A72 pipeline



Proposed parametric model



In **red**, parameters of the model.

Counting μOPs

For an instruction i, denote $\#_\mu i$ its number of μOPs.

- For $k \in \mathbb{N}$, construct (if possible) $\mathcal{K}_k$ a kernel:
  - instruction i + k “simple” instructions (one μOP)
  - frontend-bound:
$$\overline{\mathcal{K}_k} = \frac{k + \#_\mu i}{3}$$
- For well-chosen k_0, we should have
$$\overline{\mathcal{K}_{k_0}} + {}^1\!/_3 = \overline{\mathcal{K}_{k_0+1}}$$
- Measure to verify
- If so,
$$\#_\mu i = 3\overline{\mathcal{K}_{k_0}} - k_0$$

Evaluation: comparison to bare `Palmed`

- Add a frontend to `Palmed`:

$$\overline{\mathcal{K}}_{\text{pred.}} = \text{max}(\texttt{palmed}(\mathcal{K}), \texttt{frontend}(\mathcal{K}))$$

- Reuse evaluation suite of `Palmed`: SPEC CPU 2017 + Polybench
- Compare to `llvm-mca`

Results

			`llvm-mca`	`Palmed` with frontend…		
				none	linear	disp. queues
SPEC	Cov.	(%)	100.00	N/A	97.21	97.16
	Err.	(%)	9.0	20.1	6.2	4.6
	τ_K	(1)	0.83	0.88	0.91	0.93
Polybench	Cov.	(%)	100.00	N/A	99.33	99.33
	Err.	(%)	13.9	12.6	8.1	8.0
	τ_K	(1)	0.47	0.82	0.88	0.90