Additional Applications and Summary

Bar-Ilan Winter School on Verifiable Computation
Class 10 (last one!)
January 6, 2016

Michael Walfish
Dept. of Computer Science, Courant Institute, NYU

(1) Costs and comparisons

(2) Additional applications

(3) Summary and wrap-up

Costs arise from the front-end, the back-end, and their interaction

Goals:

- Understand concrete costs
- Understand the different amortization regimes
- Understand current trade-offs

Plan:

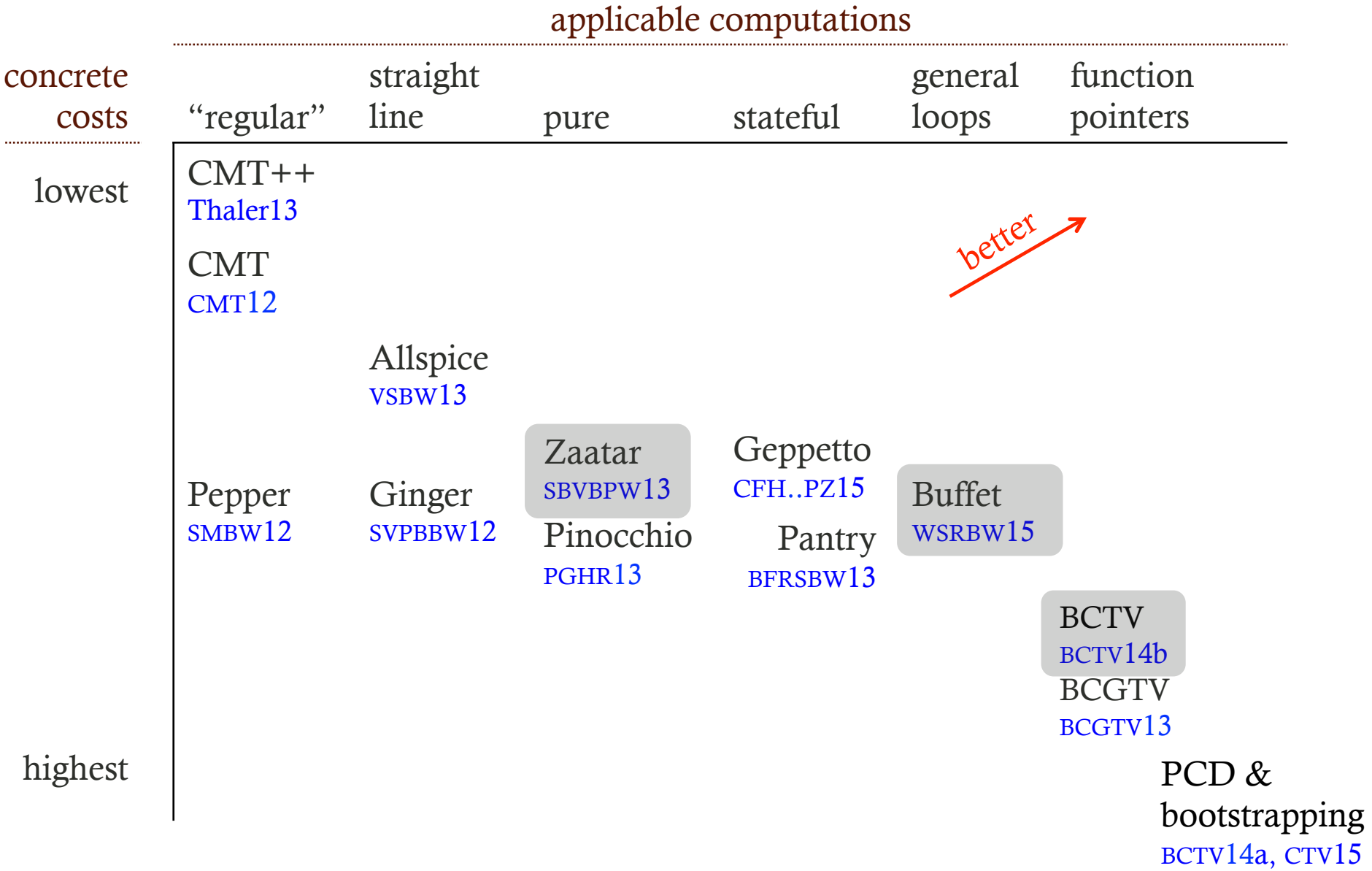
- Compare front-ends, by holding back-end constant
- Compare back-ends on two different circuits
- Examine various metrics (mostly running times)
- Examine the amortization regimes

Front-end comparison

Back-end: libsnark, which is BCTV's [BCTV14b] optimized implementation of Pinocchio/GGPR [PGHR13, GGPR13].

Front-ends: implementations or re-implementations of

- Zaatar (ASIC) [SBVBPW13]
- BCTV (CPU) [BCTV14b]
- Buffet (ASIC) [WSRHBW15]


applicable computations
concrete costs
"regular"
straight line
pure
stateful
general loops
function pointers
lowest
highest
CMT++
Thaler13
CMT
CMT12
Allspice
VSBW13
Pepper
SMBW12
Ginger
SVPBBW12
Zaatar
SBVBPW13
Pinocchio
PGHR13
Geppetto
CFH..PZ15
Pantry
BFRSBW13
Buffet
WSRBW15
BCTV
BCTV14b
BCGTV
BCGTV13
PCD & bootstrapping
BCTV14a, CTV15
better

Front-end comparison

Back-end: libsnark, which is BCTV's [BCTV14b] optimized implementation of Pinocchio/GGPR [PGHR13, GGPR13].

Front-ends: implementations or re-implementations of

- Zaatar (ASIC) [SBVBPW13]
- BCTV (CPU) [BCTV14b]
- Buffet (ASIC) [WSRHBW15]

Evaluation platform: cluster at Texas Advanced Computing Center (TACC)

Each machine runs Linux on an Intel Xeon 2.7 GHz with 32GB of RAM.

(1) What are the verifier's costs?

(2) What are the prover's costs?

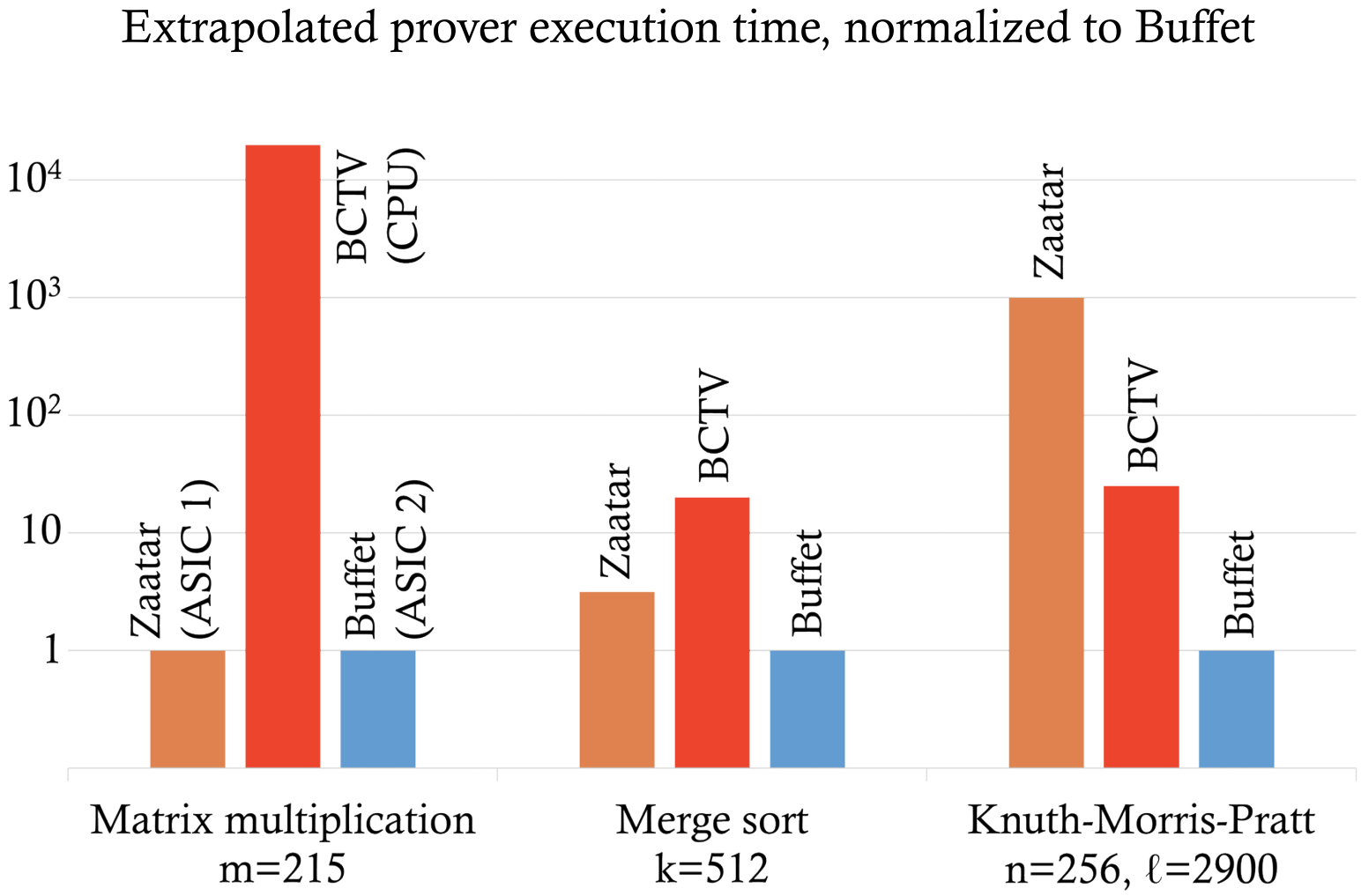
Proof length	288 bytes
V per-instance	6 ms + (|x| + |y|)·3 μs
V pre-processing	|C|·180 μs
P per-instance	|C|·60 μs + |C|log |C|·0.9μs
P's memory requirements	O(|C|log|C|)

(|C|: circuit size)

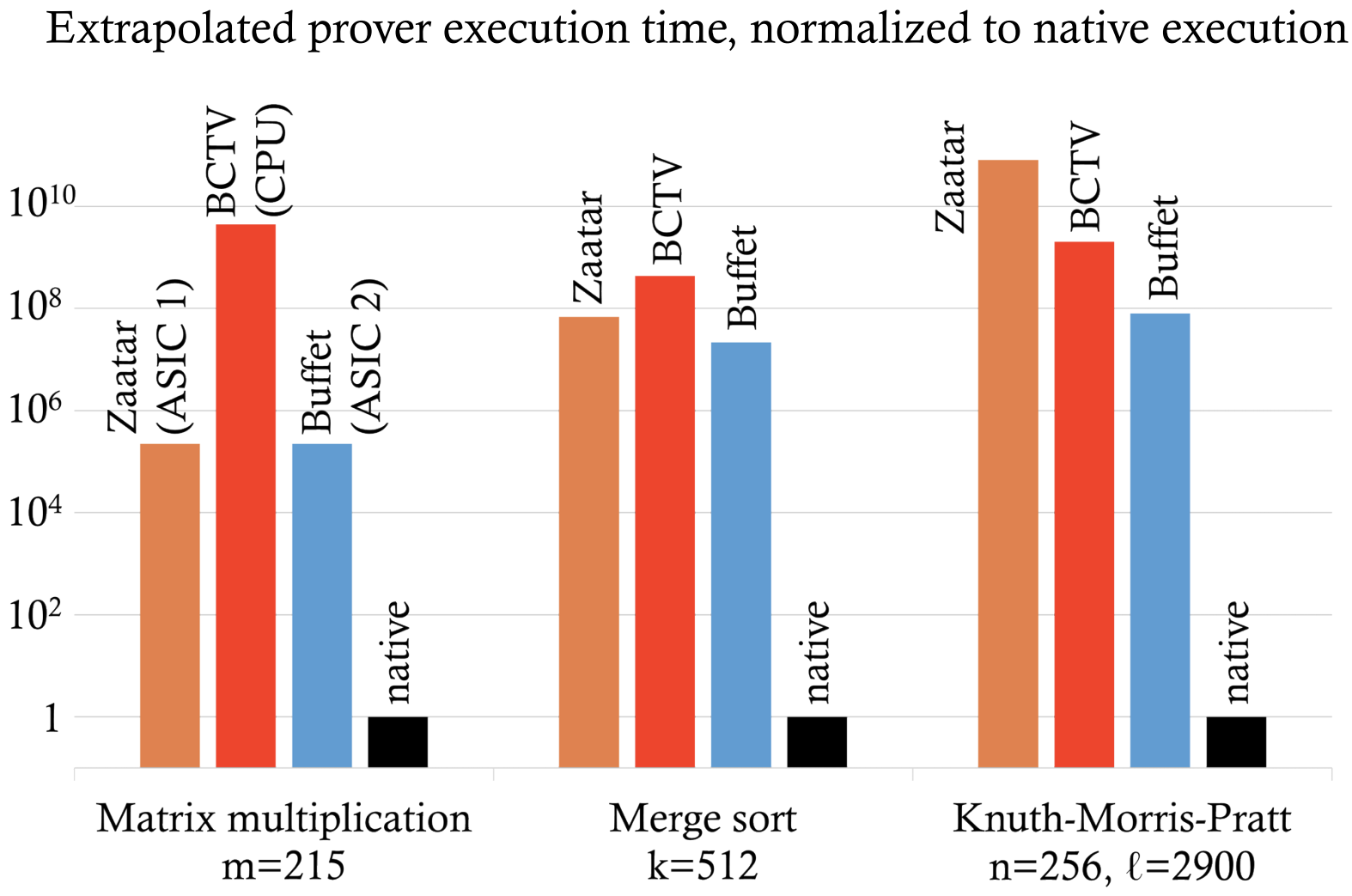
(3) How do the front-ends compare to each other?

(4) Are the constants good or bad?

How does the prover's cost vary with the choice of front-end?



All of the front-ends have terrible concrete performance



The maximum input size is far too small to be called practical

	Zaatar	BCTV	Buffet
approach	ASIC	CPU	ASIC
m × m mat. mult	215	7	215
merge sort m elements	256	32	512
KMP str len: m substr len: k	m=320, k=32	m=160, k=16	m=2900, k=256

The data reflect a "gate budget" of $\approx 10^7$ gates.

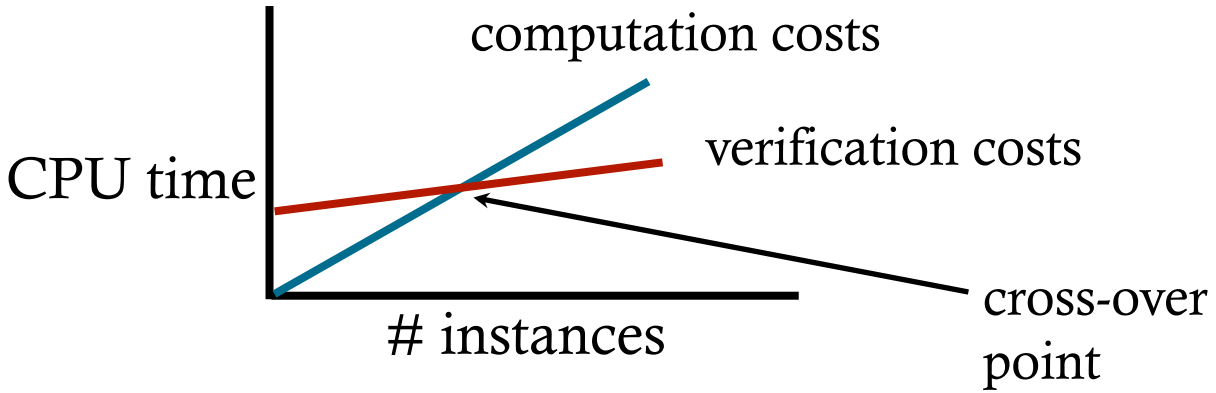
Pre-processing costs 10-30 minutes; proving costs 8-13 minutes

Back-end comparison

- Data are from our re-implementations and match or exceed published results.
- All experiments are run on the same machines (2.7Ghz, 32GB RAM). Average 3 runs (experimental variation is minor).
  - For a few systems, we extrapolate from detailed microbenchmarks
- Benchmarks: 128×128 matrix multiplication and clustering algorithm

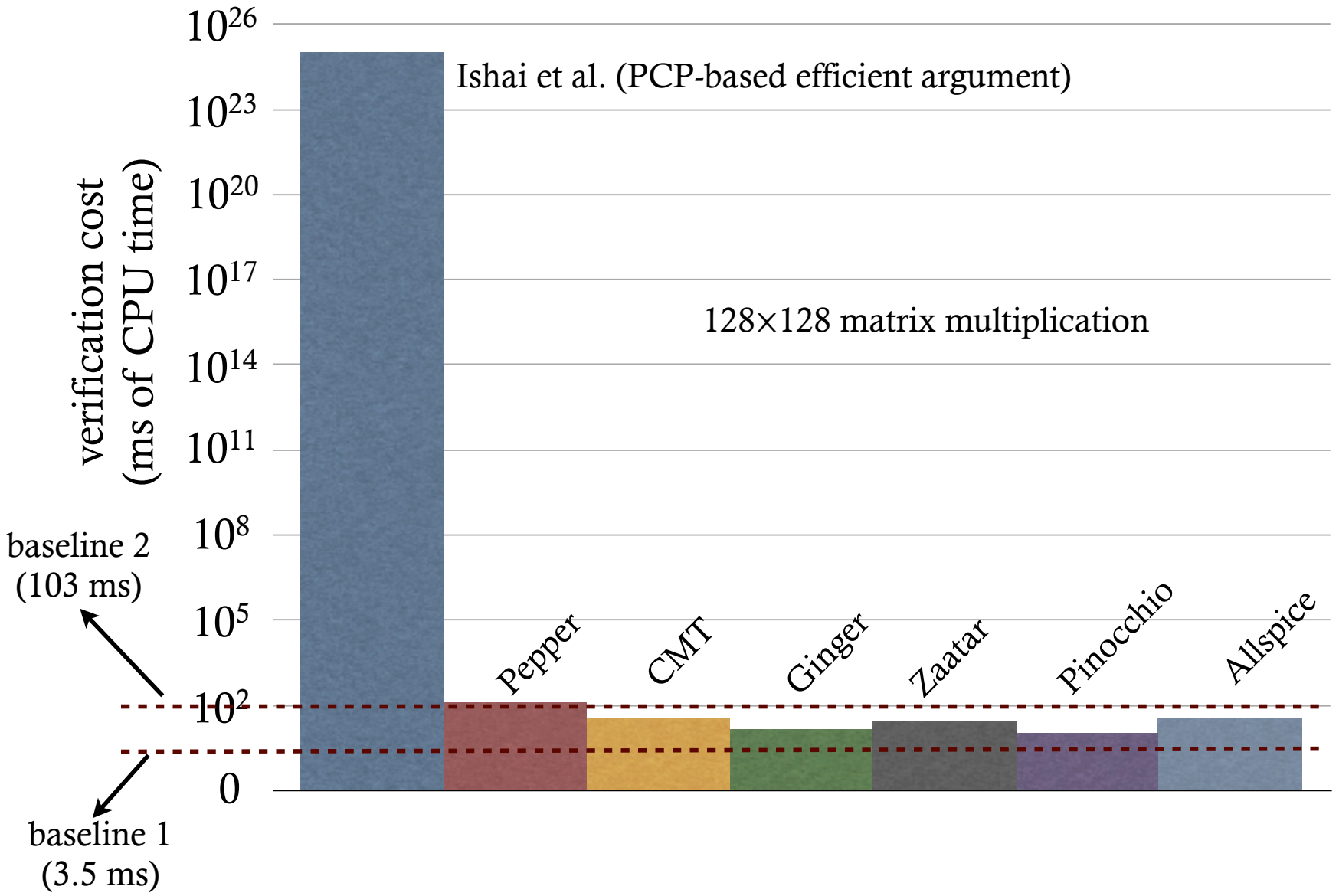
1. What is the per-instance verification cost?

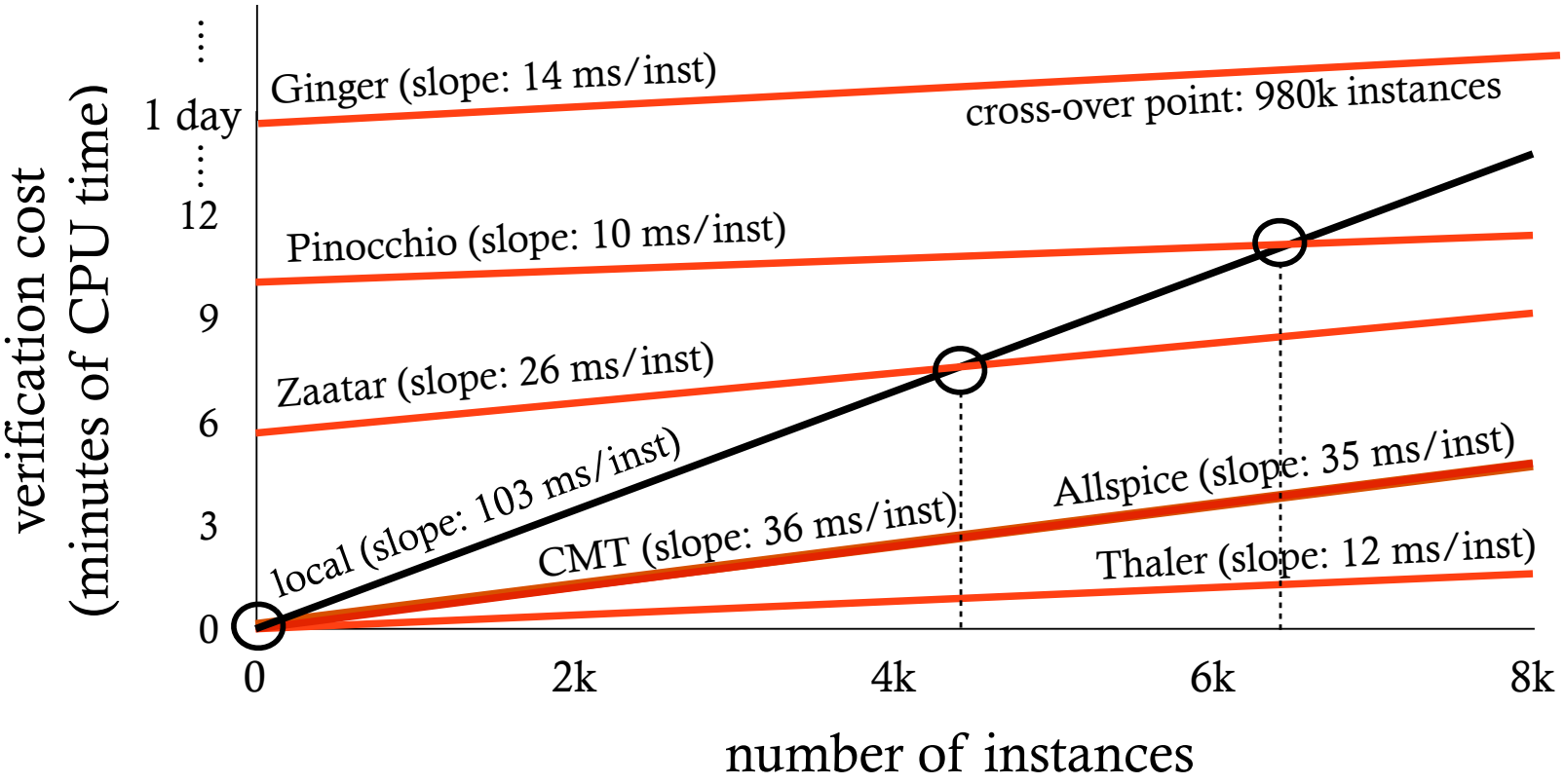
2. What are the cross-over points?



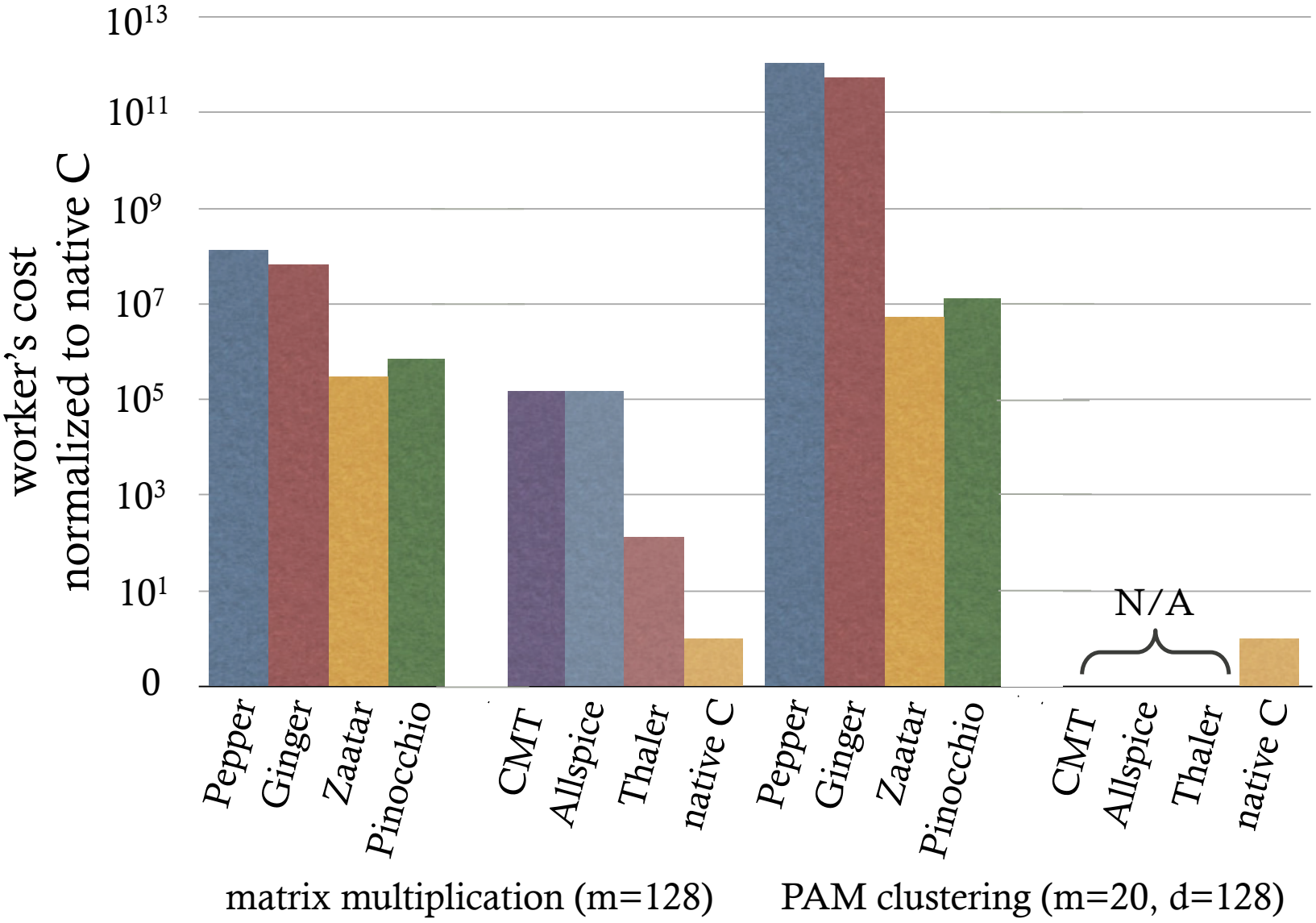
3. What is the server's overhead versus native execution?

Verification cost sometimes beats (unoptimized) native execution.

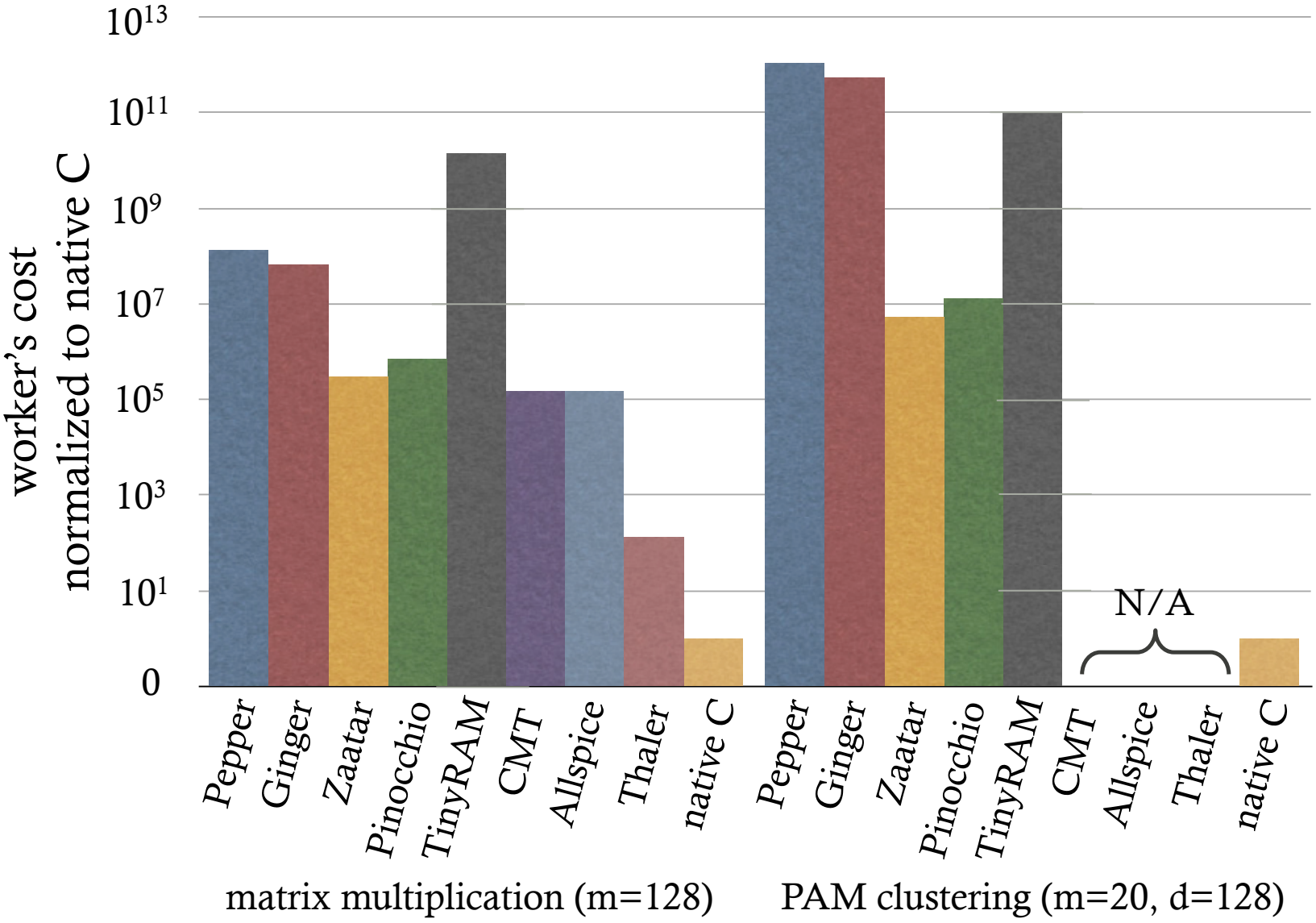



verification cost
(minutes of CPU time)
1 day
12
9
6
3
0
Ginger (slope: 14 ms/inst)
cross-over point: 980k instances
Pinocchio (slope: 10 ms/inst)
Zaatar (slope: 26 ms/inst)
local (slope: 103 ms/inst)
CMT (slope: 36 ms/inst)
Allspice (slope: 35 ms/inst)
Thaler (slope: 12 ms/inst)
0
2k
4k
6k
8k
number of instances

The prover's costs are rather high.



The prover's costs are rather high.



Amortization comparison (of built systems)

Systems [CMT12, VSBW13, Thaler13] derived from [GKR08] require little or no amortization (but have some expressivity limitations)

Of the schemes that handle arbitrary circuits (that is, those based on arguments), preprocessing costs amortize differently. Ordered best to worst:

1. Bootstrapped GGPR-based SNARKs [BCTV14a, CTV15]
   - Constant preprocessing; amortize over all computations (but concrete costs to prover are extremely high).
2. BCTV [BCTV14b]: "CPU" front-end + non-interactive GGPR back-end
   - Amortize over all future computations of the same length
3. Pinocchio [PGHR13]: "ASIC" front-end + non-interactive GGPR back-end
   - Amortize over all future uses of a given computation
4. Zaatar [SBVBPW13]: "ASIC" front-end + interactive GGPR/IKO back-end
   - Amortize over a batch of instances of a given computation

Summary of concrete performance

- Front-end: generality brings a concrete price (but better in theory)

- Verifier's "variable costs": genuinely inexpensive
- Verifier's "pre-processing": depends on setting
- Prover's computational costs: mostly disastrous

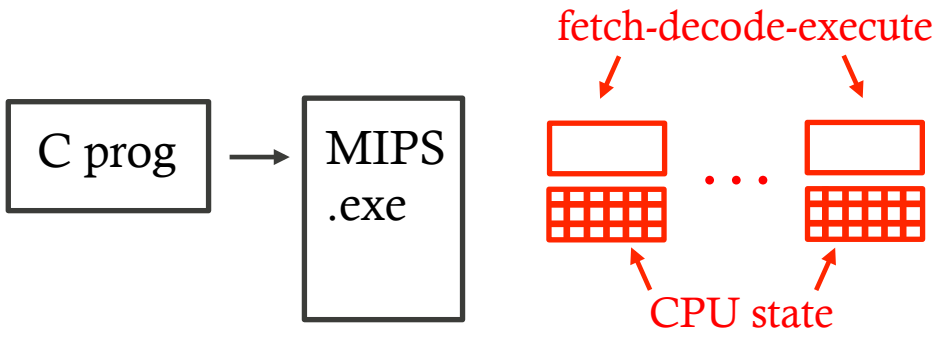
- Memory: creates scaling limit for verifier and prover

Performance is plausibly acceptable in certain settings …

- It must be "regular" (to avoid setup costs), or there must be many identical instances (to amortize setup costs)
- The given computation needs to be small

… But none of the systems is at true practicality

Summary of front-ends

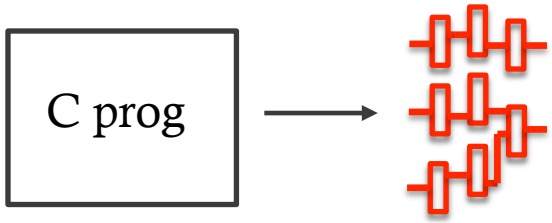


circuit is unrolled CPU execution

[BCGTV13, BCTV14a, BCTV14b, CTV15]

"CPU"

- Verbose (costly)
- Good amortization
- Great programmability



each line translates to gates/constraints

[SVPBBW12, SBVBPW13, VSBW13, PGHR13, BFRSBW13, BCGGMTV14, BBFR14, FL14, KPPSST14, WSRBW15, CFHKKNPZ15]

"ASIC"

- Concise
- Amortization worse
- Programmability not bad

(1) Costs and comparisons

(2) Additional applications

(3) Summary and wrap-up

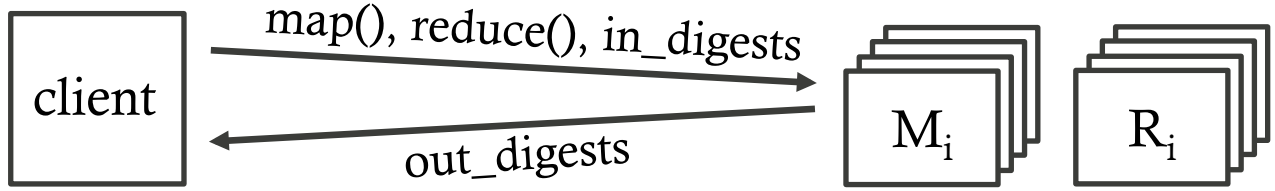
For H, we use an algebraic collision-resistant hash function

[Ajtai STOC96, Goldreich, Goldwasser, Halevi, ECCC96]

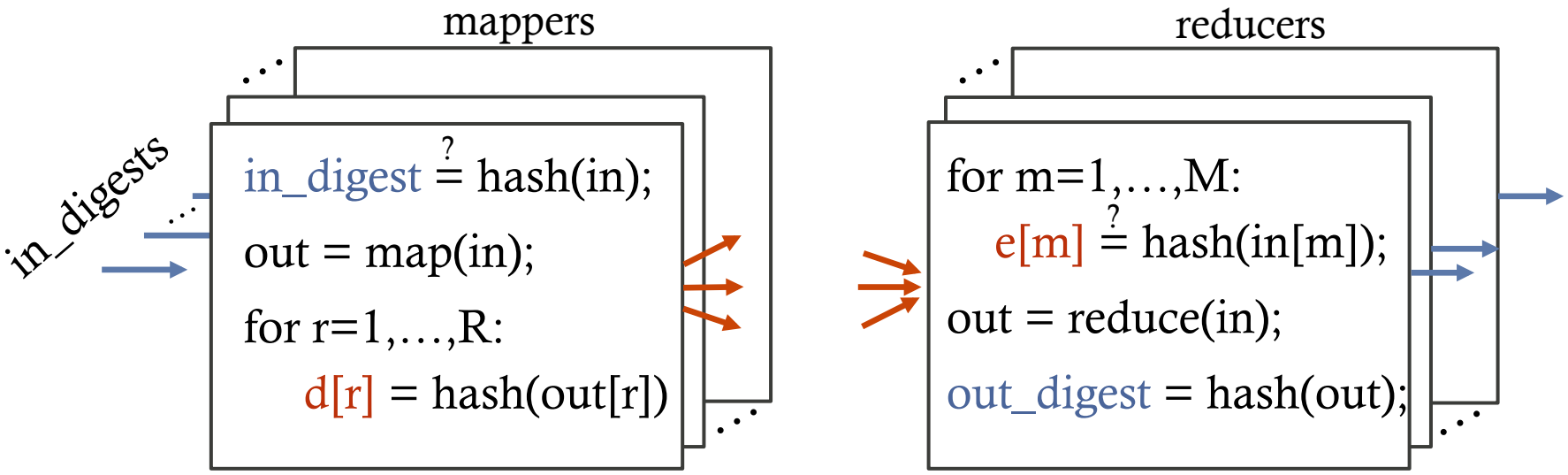
$$D = H(Z) \rightarrow \left\{ \begin{array}{l} D_1 = M_{1,1} \bullet Z_1 + \ldots + M_{1,m} \bullet Z_m \pmod q, \\ D_2 = M_{2,1} \bullet Z_1 + \ldots + M_{2,m} \bullet Z_m \pmod q, \\ \quad \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\ D_n = M_{n,1} \bullet Z_1 + \ldots + M_{n,m} \bullet Z_m \pmod q \end{array} \right\}$$

Choice of ($m=7296$, $n=64$, $q=2^{19}$) from [Micciancio & Regev 08]

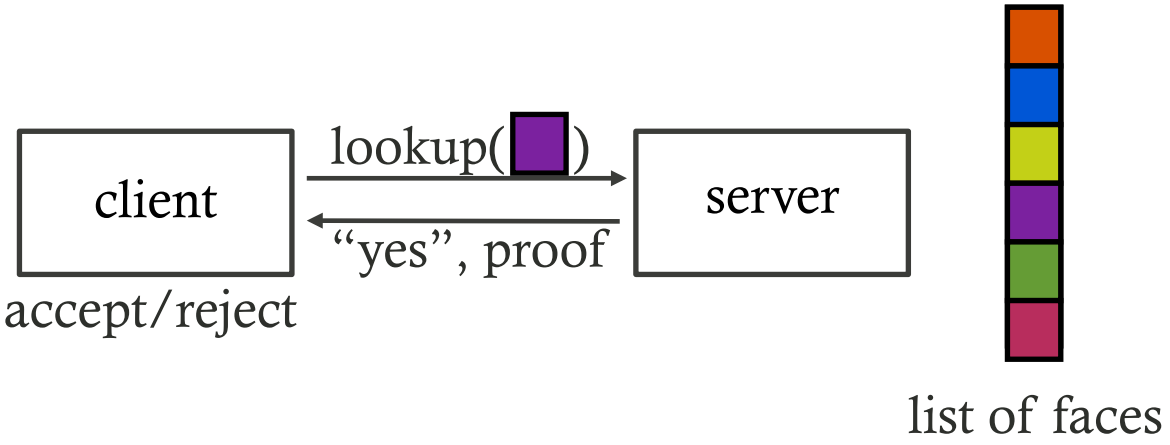
A client can be assured that a MapReduce job was performed correctly—without ever touching the data.



The two phases are handled separately:



Hidden state applications



Key idea: External storage plus zero knowledge proof variants [PGHR OAKLAND13]

Requires small adjustments in protocol (b/c digests don't hide state)

Other applications: tolling, regression analysis, etc.

Upshot: write C programs, get powerful guarantees

Evaluation questions

(1) When does the client save resources relative to locally executing the computation?

(2) What are the costs of supporting hidden state?

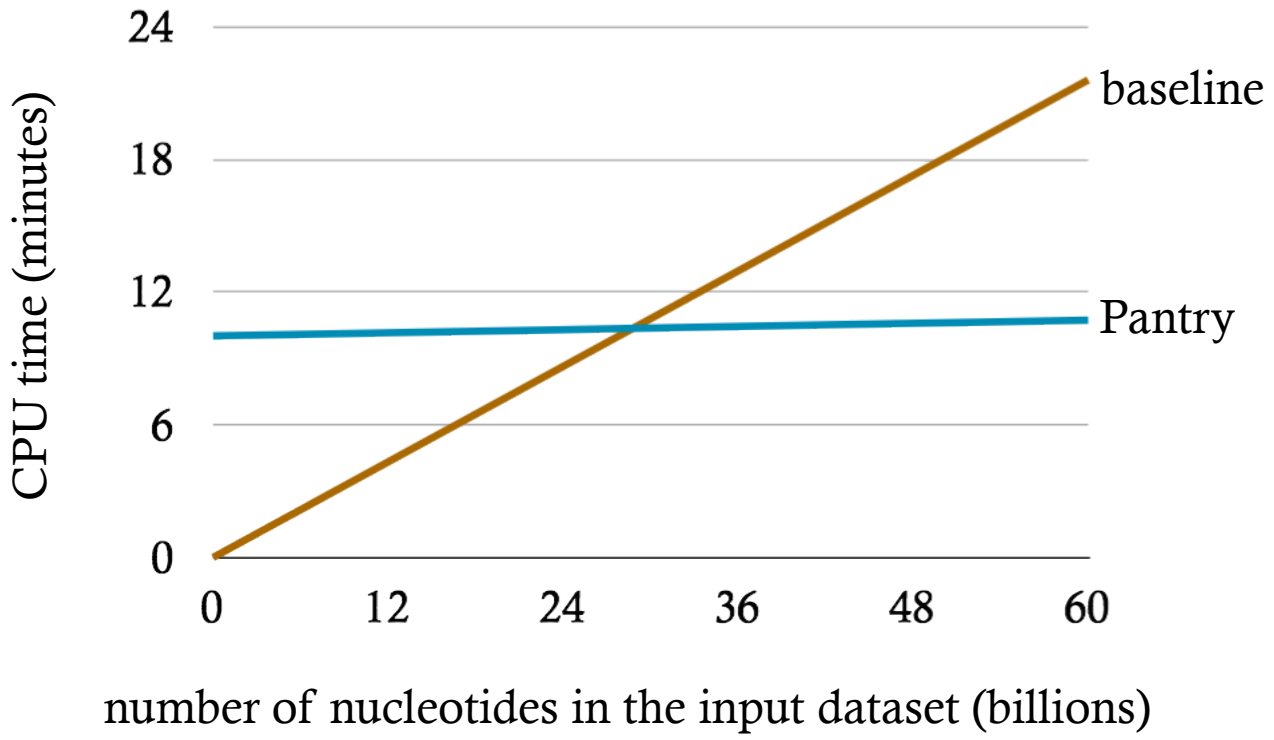
Compiler pipeline implemented in C++, Java, Go, Python

Server is distributed; communication uses OpenMPI

Evaluation platform: a cluster at Texas Advanced Computing Center (TACC)

Each machine runs Linux on an Intel Xeon 2.7 GHz with 32GB of RAM.

A client saves resources with sufficiently large inputs



- Nucleotide substring search: A mapper gets 600k nucleotides and outputs matching locations. One reducer per 10 mappers.
- The graph is an extrapolation but is nonetheless encouraging.

Cost of supporting hidden state applications

Server holds 128 face fingerprints (hidden state: 15KB)

The good news:

- Proof size: 288 bytes
- Client's CPU time: 7 ms

The bad news:

- Network cost (setup) and storage cost (ongoing): 170MB
- Server's CPU time: 7.8 minutes

(1) Costs and comparisons

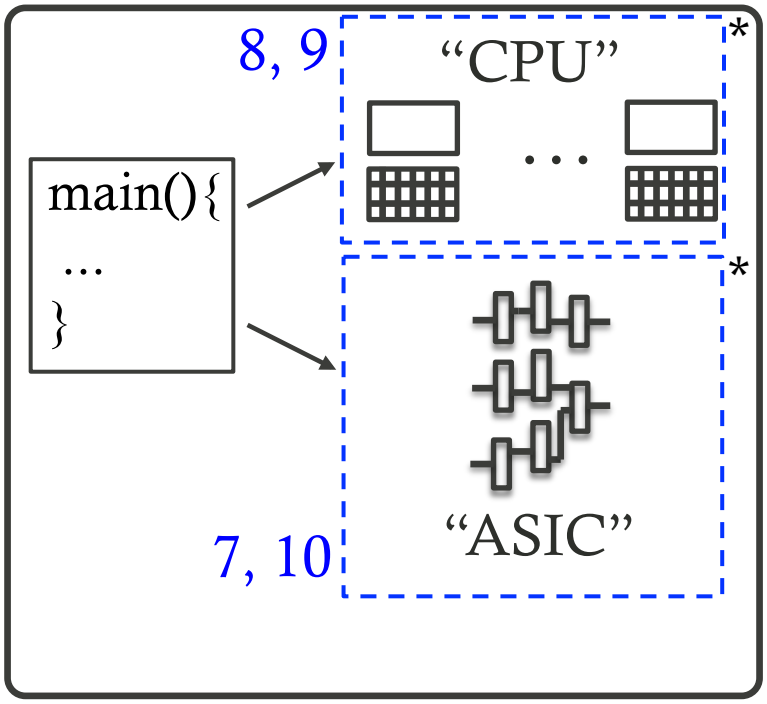
(2) Additional applications

(3) Summary and wrap-up

Classes at a glance (numbers in blue refer to class number)

front-end
(program translator)

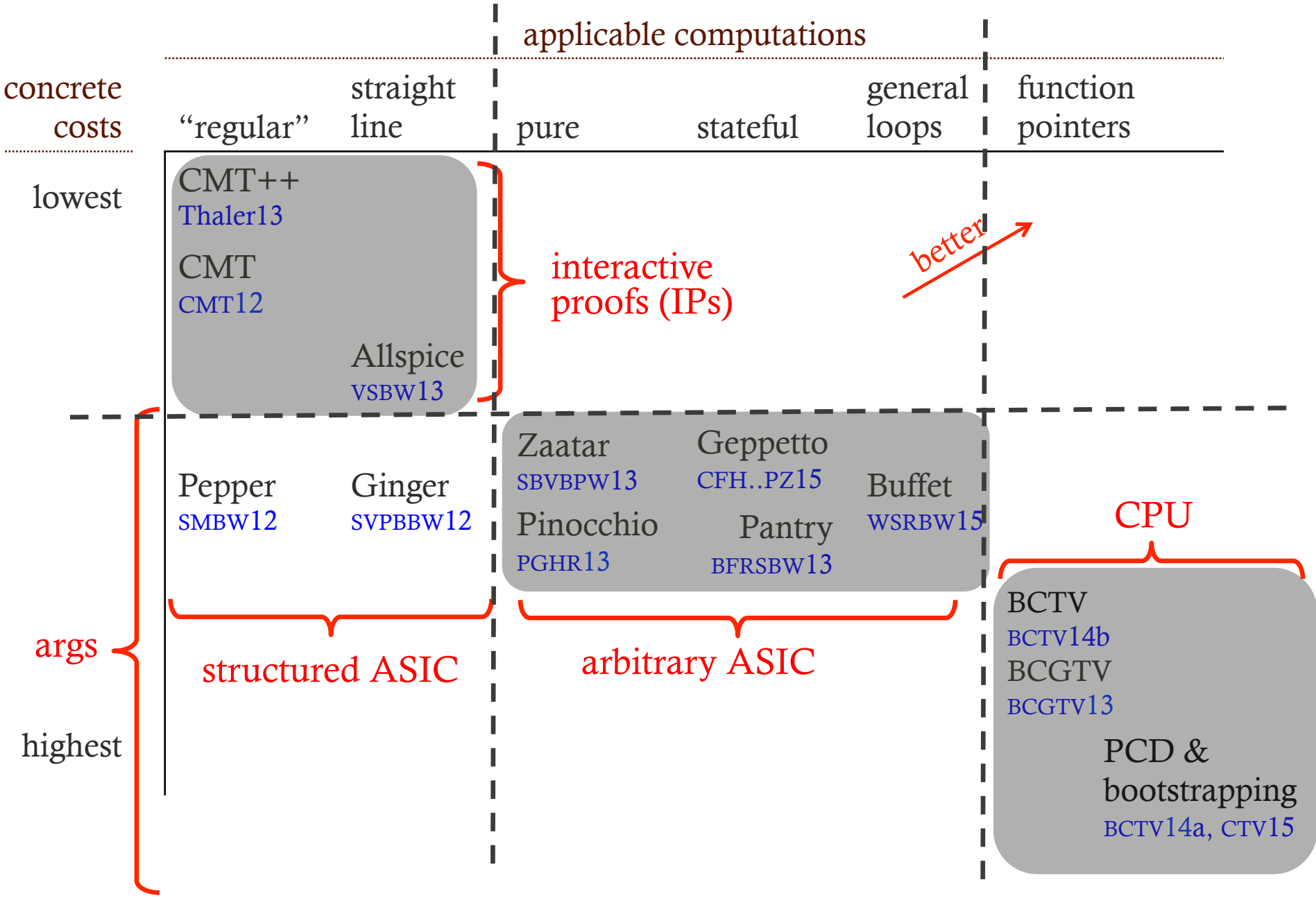
back-end
(probabilistic proof protocol)



	interactive proofs (IPs)	interactive args.	non-interactive args.
no	2 *	3, 4	8, 9
preproc	10 *	(QAPs) 5 *	(QAPs) 6 *

8, 9 bootstrapping (recursive use of the machinery) *

* Indicates that the mechanism has been implemented


applicable computations
concrete costs
straight line
"regular"
pure
stateful
general loops
function pointers
lowest
highest
CMT++
Thaler13
CMT
CMT12
Allspice
VSBW13
interactive proofs (IPs)
better
Pepper
SMBW12
Ginger
SVPBBW12
Zaatar
SBVBPW13
Geppetto
CFH..PZ15
Buffet
WSRBW15
Pinocchio
PGHR13
Pantry
BFRSBW13
CPU
BCTV
BCTV14b
BCGTV
BCGTV13
PCD & bootstrapping
BCTV14a, CTV15
args
structured ASIC
arbitrary ASIC

Lots of open problems and questions

- Unconditionally secure delegation for all of PSPACE (YTK $100)
- 2-msg delegation for $\mathcal{NP}$ with standard assumptions (YTK)
- Publicly-verif. 2-msg delegation for $\mathcal{P}$ with std. assumptions (YTK)
- Zero knowledge with standard assumptions that is inexpensive in practice
- More efficient reductions from programs to circuits
- More efficient encodings of execution traces
- Probabilistic proof protocols that do not require circuits
- Avoiding preprocessing/amortization in a way that is inexpensive in practice
- Special-purpose algorithms for outsourcing pieces of computations, which integrate with circuit verification

Final thoughts

- Exciting area with lots to do!
- A cautionary tale …
- But I am optimistic!