Performance Prediction of Memory Access Intensive Apps with Delay Insertion: A Vision

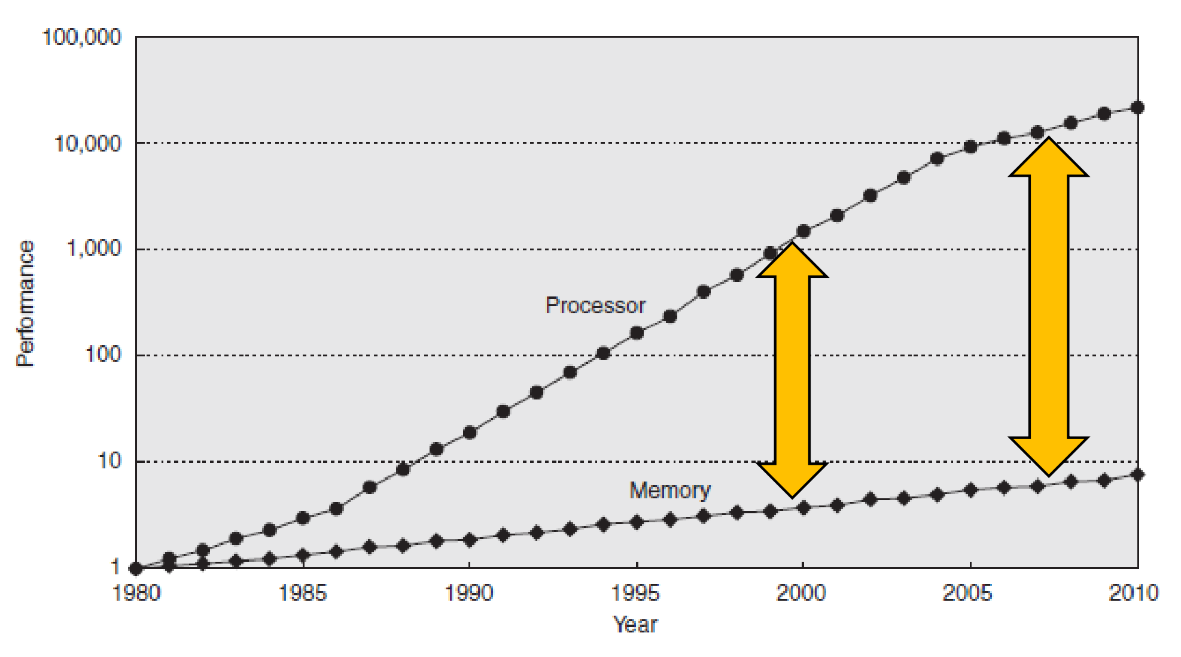
Soramichi Akiyama Takahiro Hirofuchi Hirotaka Ogawa
National Institute of Advanced Industrial Science and Technology (AIST), Japan

Background

Performance prediction of a given program is **highly important but difficult**.

Importance: environments where the program is developed and where it deployed are not the same (ex: developed in a handy laptop, then deployed to a powerful server).

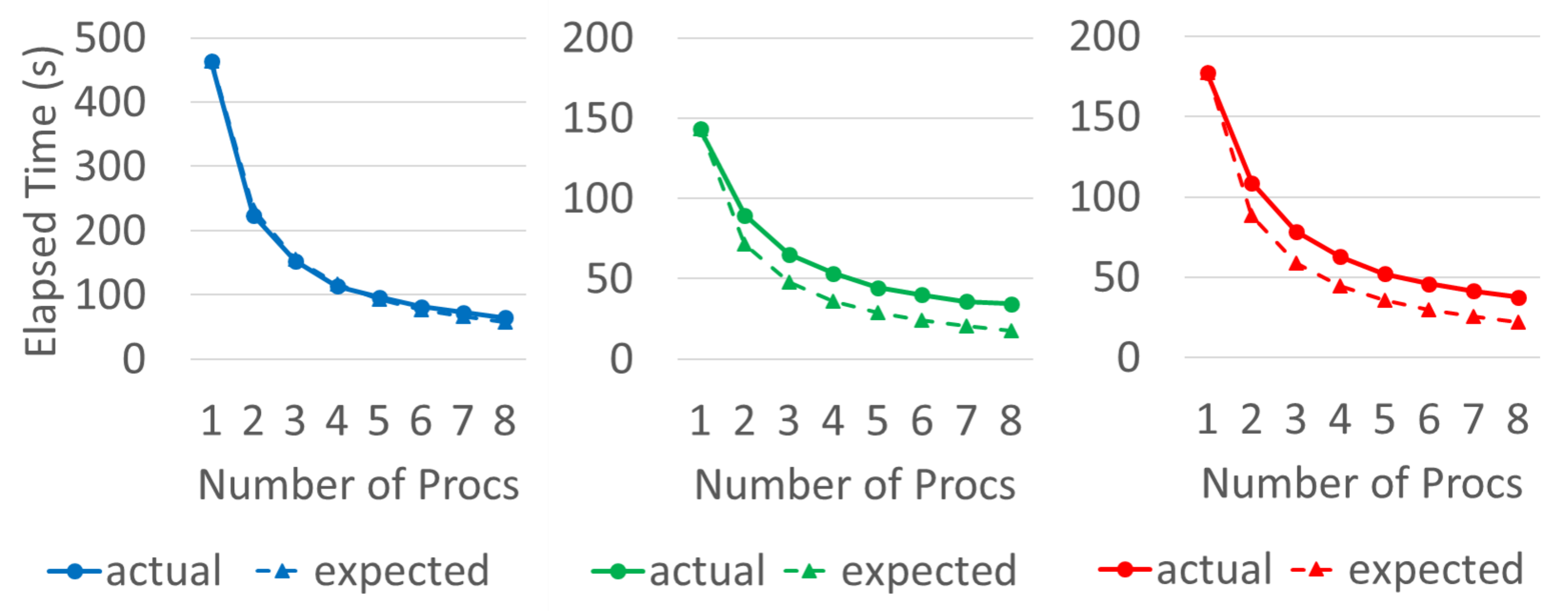
Difficulty: two machines have different performance balance among components.



Processors relatively speeding up compared to memory
→ **Newer machines are more memory latency-sensitive**

Patterson and Hennesy, "Computer Architecture, Fourth Edition: A Quantitative Approach", 2006 (Fig. 5.2 in page 289)

Real Example



8K x 8K Matrix factorization w/ Python on various machines

	Machine A	Machine B	Machine C
CPU (Xeon)	E5-2603 **v1**	E5-2699 **v3**	E5-2690 **v4**
Mem	DDR3 1600	DDR4 2133	DDR4 2133
Perfect Scale?	**Yes**	**No**	**No**

→ Exactly the same program scales differently on three machines (due to different flops/memory latency ratios).

Proposal and Proof-of-Concept Implementation

Main Ideas:

1. Emulate performance balance of the target machine with Dynamic Binary Instrumentation
2. Run the target code as-is to retrieve more useful information than model-based techniques can provide

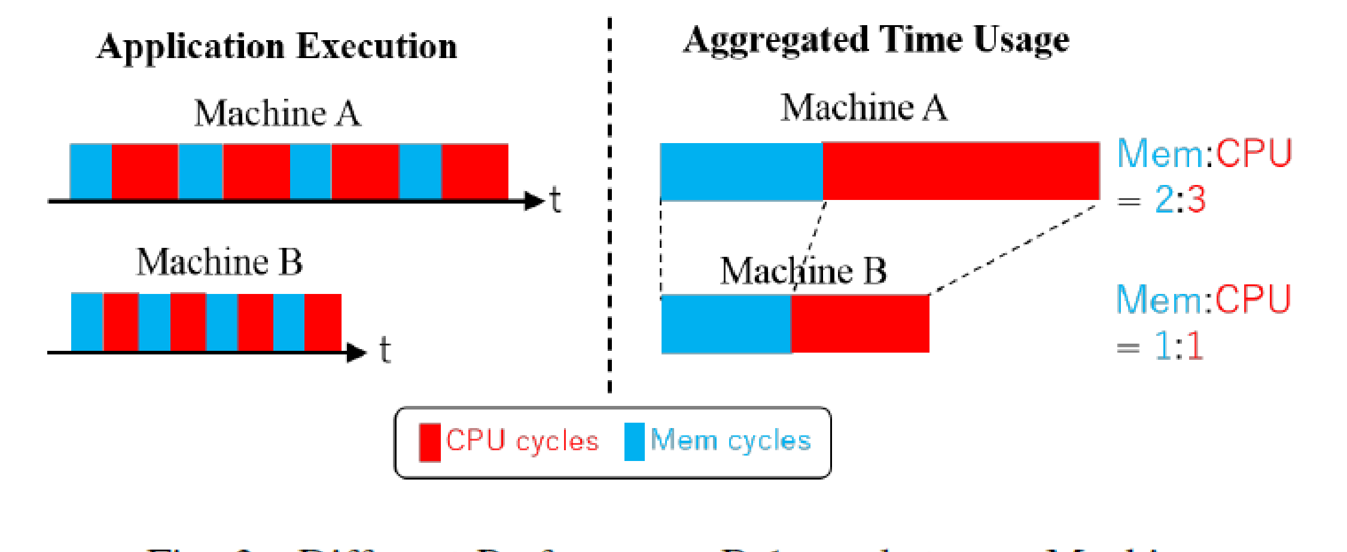
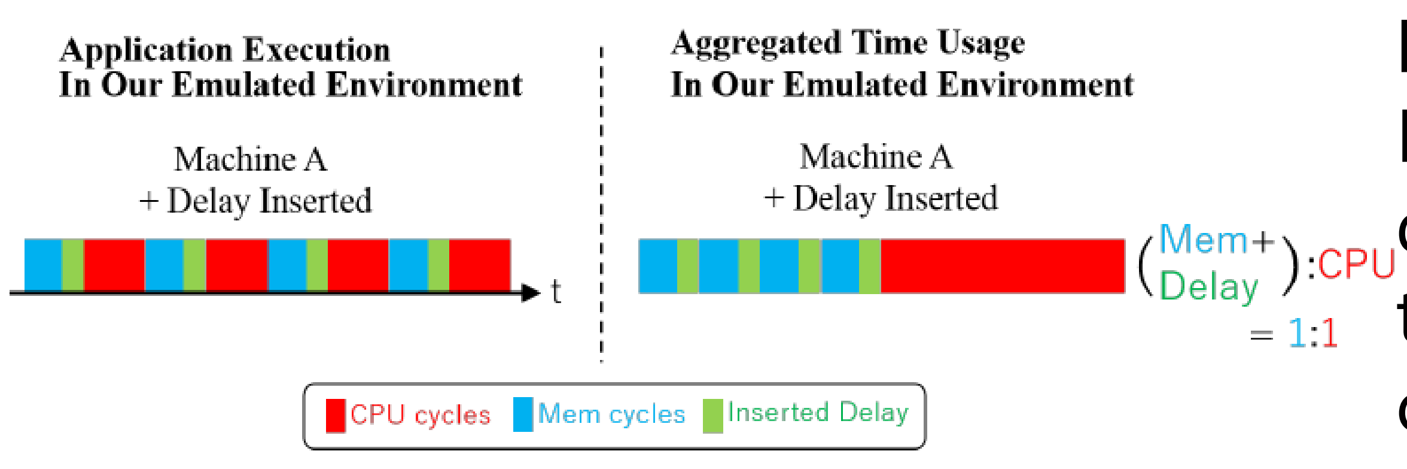


Fig. 2. Different Performance Balance between Machines

Normal Execution:
CPU stalls due to memory access 40% of time in machine A, but the stall is 50% in machine B (different perf balance).

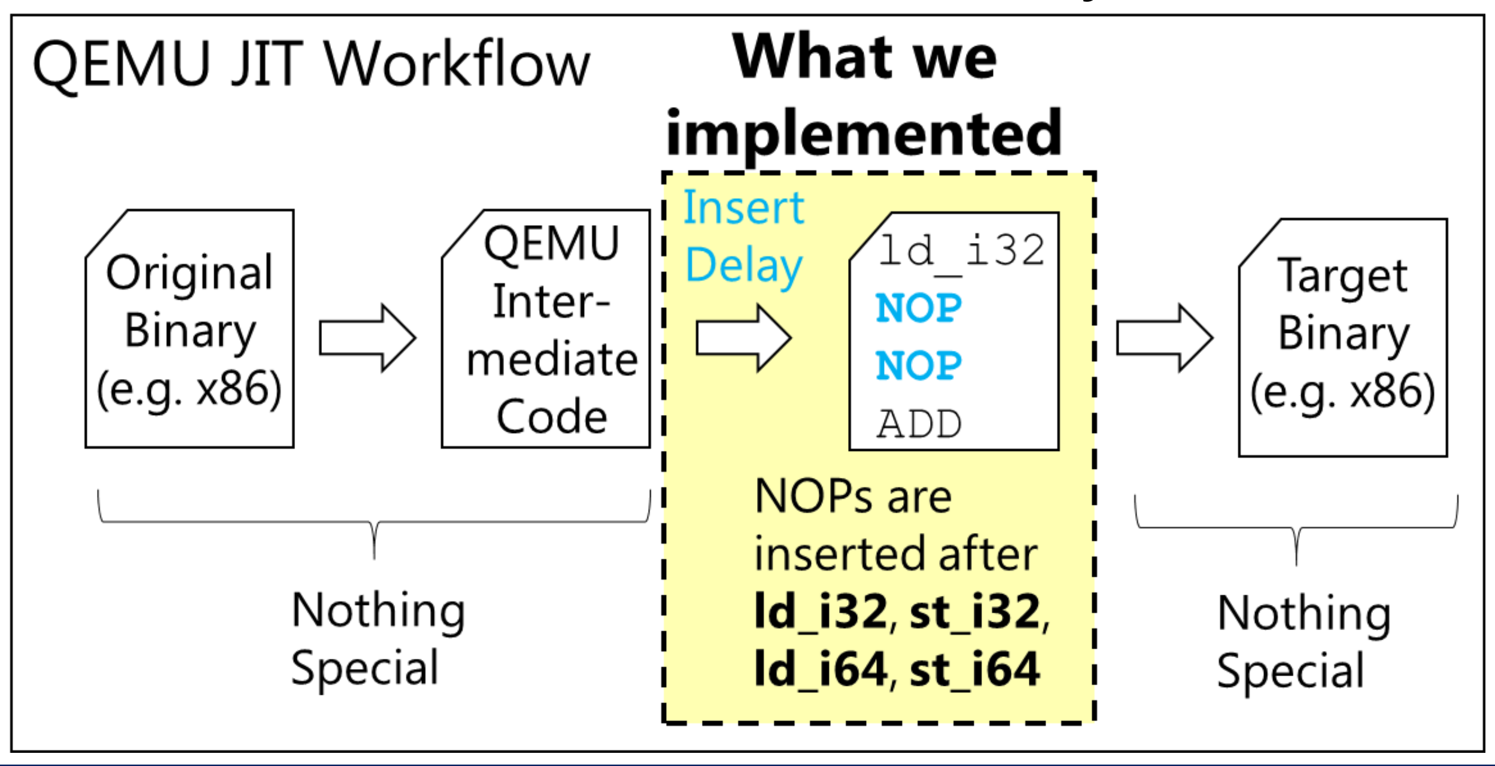

Application Execution In Our Emulated Environment
Machine A + Delay Inserted
t
Aggregated Time Usage In Our Emulated Environment
Machine A + Delay Inserted
(Mem+Delay):CPU = 1:1
CPU cycles
Mem cycles
Inserted Delay

Delayed Execution:
Memory accesses are delayed to prolong the stall to 50% in machine A.

Implementation Choices:

Based-on	Pros	Cons
Hardware	Small overhead	Less practical
Compiler	Small overhead	New compiler for all langs
DBI	**Applicable to any lang**	Large overhead

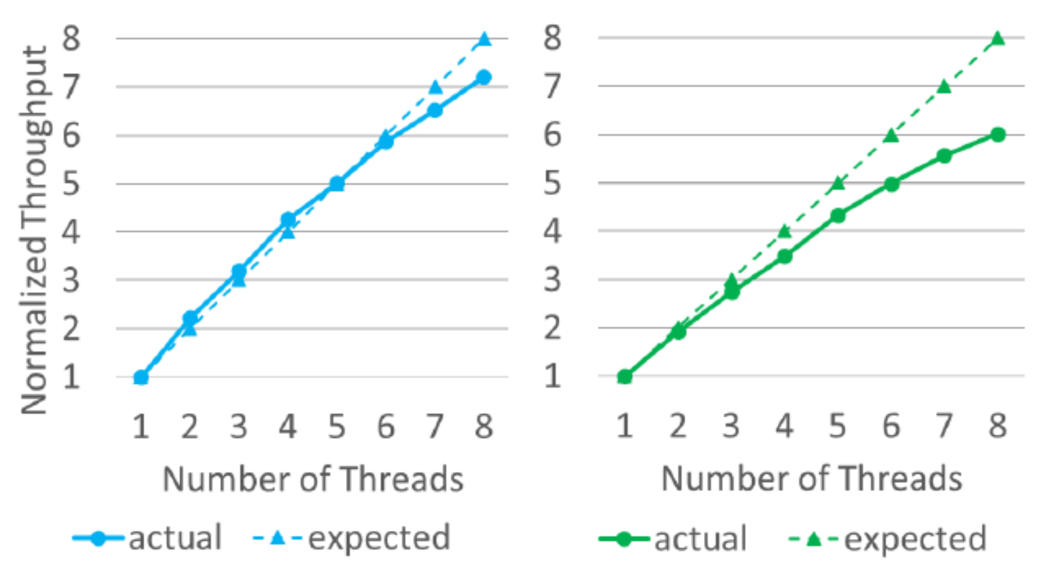
PoC Implementation:
QEMU's dynamic binary instrumentation mechanism (user-mode) is modified to insert delays after memory read/writes.



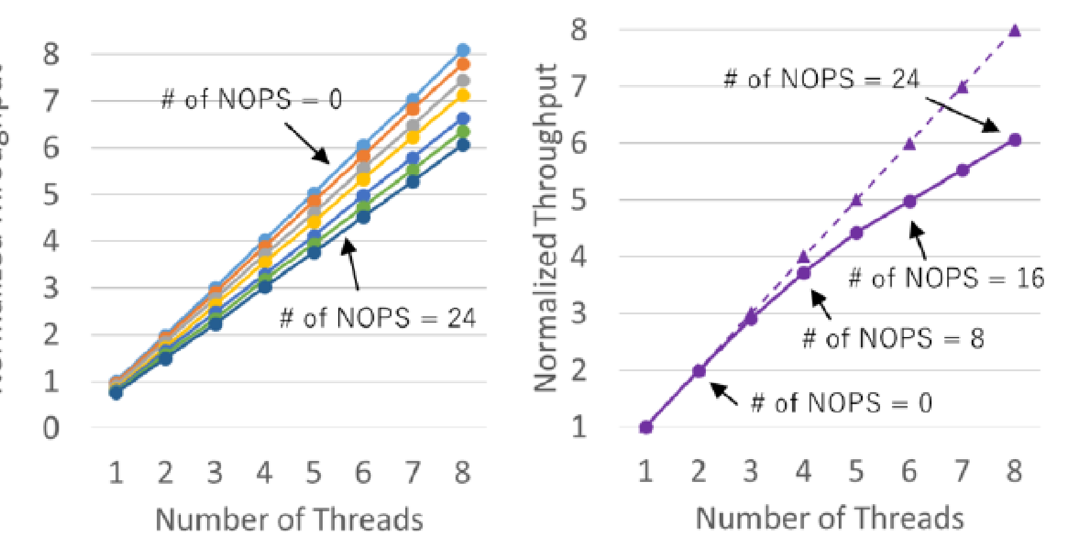
- Applicable to any langs/CPU archs
- # of memory related instructions reduced to 4
- Small extra overhead thanks to JIT cache

Preliminary Results and Future Vision

Experiment Settings:
Workload: 8K x 8K matrix-vector multiplication with no tiling
Metric: Normalized throughput (inverse of the execution time)
Q: Can our method adjust perf balance among two machines??



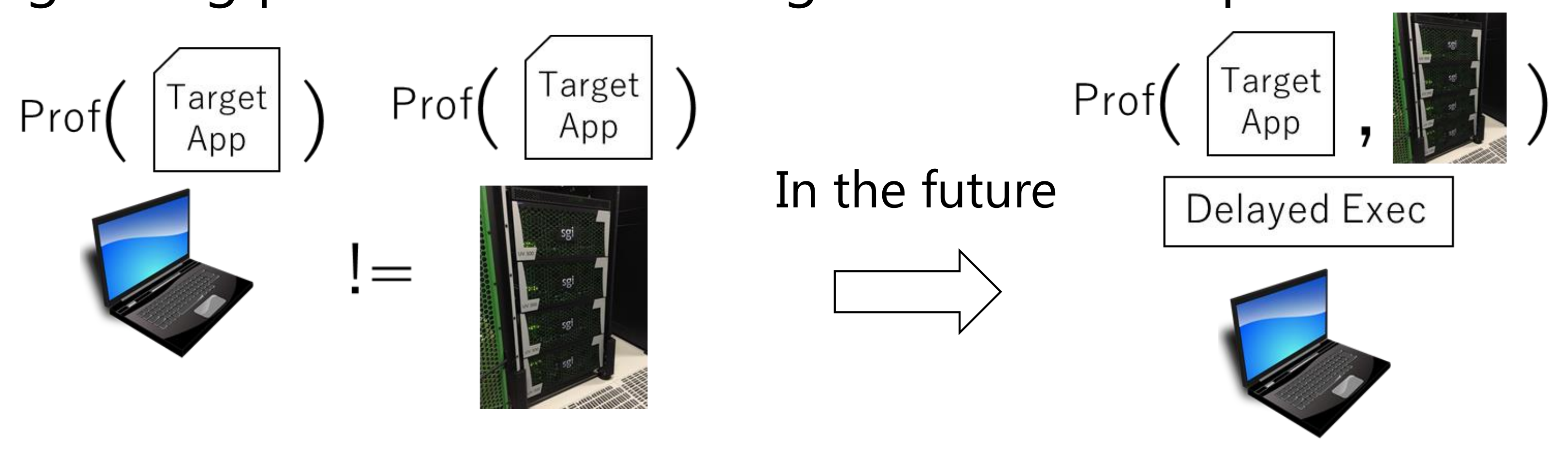
Throughput scales better in machine A (old one),since machine A is less memory access latency sensitive.



By inserting increasing # of NOPs w.r.t # of threads, **our proposal can emulate the normalized throughput achieved in machine B (using only machine A).**

Future Vision:

1. **Running profilers on our mechanism** greatly helps diagnosing perf issues stemming from different perf balances.



2. Perf models of multi-threaded apps requires explicit/implicit data-dependency analysis and often inaccurate. **Our mechanism automatically propagates the effect of prolonged critical sections as the whole code is executed.**

Technical Challenges:

1. Deciding number of NOPs to insert systematically
2. System call overhead that may break the whole balance

*This paper is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO).