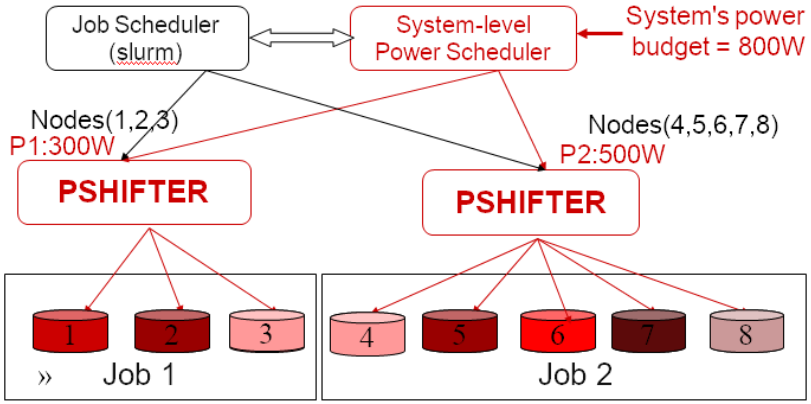
SUIT: Secure Undervolting with Instruction Traps

Frank Mueller
mueller@cs.ncsu.edu
North Carolina State University

NC STATE UNIVERSITY
Department of Computer Science

Power-Aware Computing at Scale -- RAPL

- PTune + PPartition: Enforce system-level power budget + improve both system and job performance **[PACT2016]**



- Shifter: Runtime system to reduce performance imbalance by dynamic power management **[HPDC2018]**
- Uncore Power Scavenger: Runtime system to conserve uncore power without a significant performance degradation **[SC'19]**

CASL Consortium for Advanced Simulation of LWRs

Lawrence Livermore National Laboratory

Computer Science NC STATE UNIVERSITY

ASPLOS'24 paper

SUIT: Secure Undervolting with Instruction Traps

Jonas Juffinger
Graz University of Technology
Graz, Austria
jonas.juffinger@iaik.tugraz.at

Stepan Kalinin
North Carolina State University
Raleigh, NC, USA
skalini@ncsu.edu

Daniel Gruss
Graz University of Technology
Graz, Austria
daniel.gruss@iaik.tugraz.at

Frank Mueller
North Carolina State University
Raleigh, NC, USA
fmuelle@ncsu.edu

Abstract

Modern CPUs dynamically scale voltage and frequency for efficiency. However, too low voltages can result in security-critical errors. Hence, vendors use a generous safety margin to avoid errors at the cost of higher energy overheads.

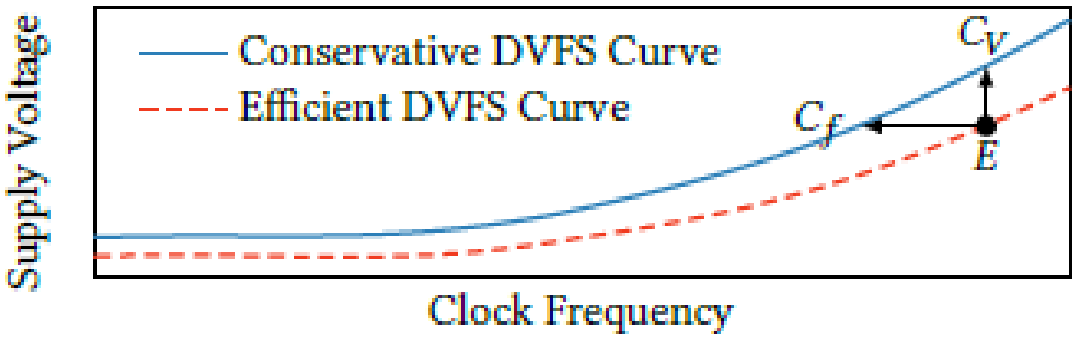
In this work, we present SUIT, a novel hardware-software co-design to reduce the safety margin substantially without compromising reliability or security. We observe that not all instructions are equally affected by undervolting faults and that most faultable instructions are infrequent in practice. Hence, SUIT addresses infrequent faultable instructions via two separate DVFS curves, a conservative and an effi-

1 In

With in
ogy [29
for mod
ous volt
ergy co
optimiz
quency
to switc
sumptio
voltage-
timal D

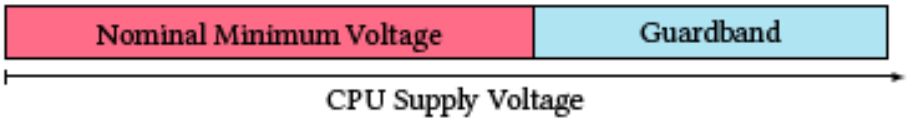
What is Undervolting?

- Power P ~ f * V^2 → higher processor frequency f, faster execution
- Gamers overvolt → increase frequency "beyond specs"
  - Higher power consumption
- Not widely know: undervolting supported as well
  - So lower f → slower execution? Well, not necessarily
    - Lower power P
    - Little performance change → occasionally even faster (?)
- This work: Undervolting for sustainable computing
  - Lower P w/o performance less

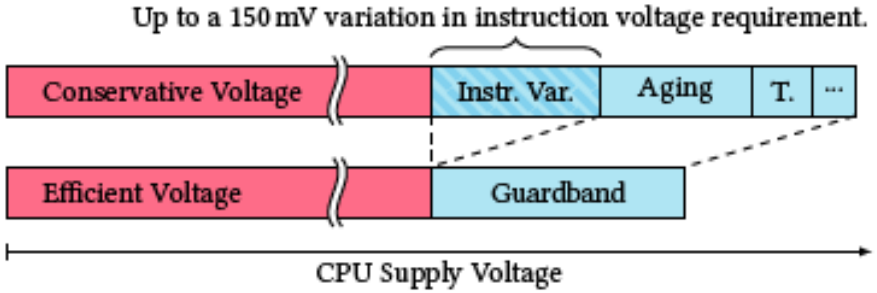


Manufacturer Guardband

- Challenge: knowing how far to undervolt before system halts
  - Manufacturer keep guardband for safety
  - Generally too conservative
    - Same for all processors



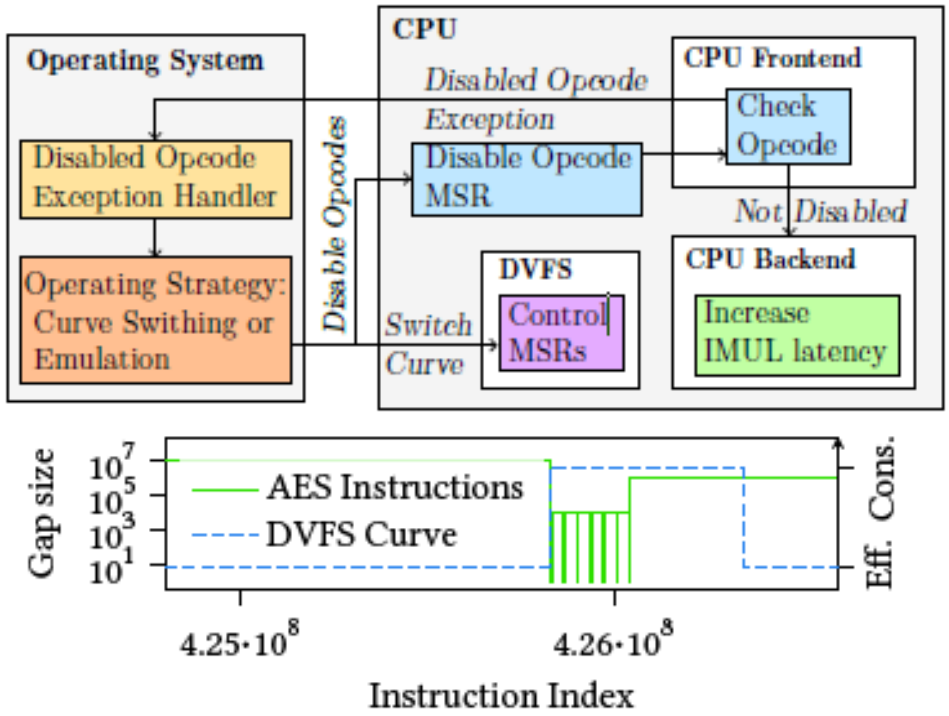
- Opportunity: redefine guardbard
  - Take process variations into account
  - Can be more aggressive to lower guardband
  - Additional P savings
- Problem: instr. require diff. P
  - Some halt system earlier



Instruction	IMUL	VOR*	AESENC	VXOR*	VANDN*	VAND*	VSQRTPD	VPCLMULQDQ	VPSRAD	VPCMP*	VPMAX*	VPADDQ
Number of Faults	79	47	40	40	30	28	24	16	9	5	3	1

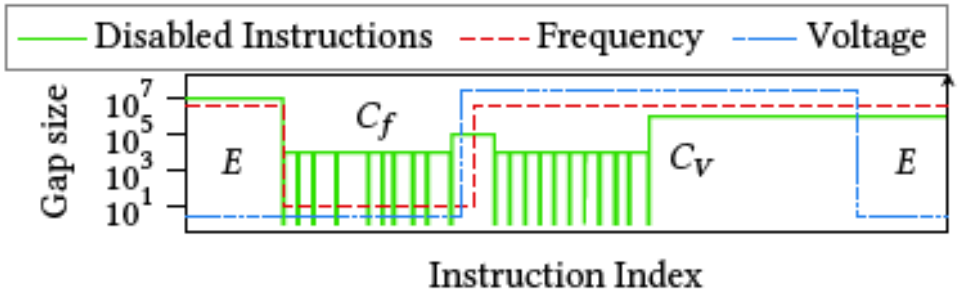
Co-Design for Undervolting Faults

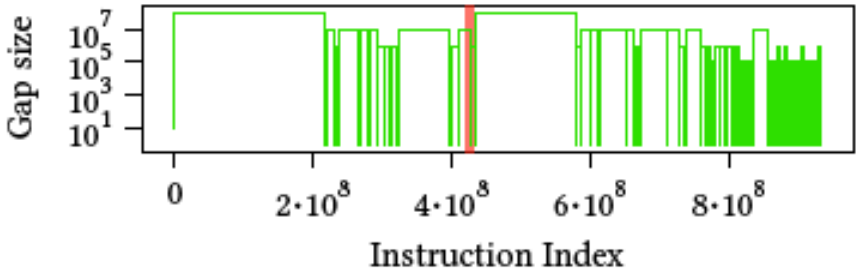
- Idea: add MSR to enable fault on power-critical instructions
  - Kernel defines handlers
  - Fault/exception on high power instr.
- 2 mitigation options:
  1. Increase f, re-execute, keep f high for some time, lower f when high-P instr. no longer seen
     → good for infrequent high-P instr. (e.g., AES, SSE, AVX, …)
  2. Modify instr. to no longer by high-P
     Ex.: multiply microcode change from 3 → 4 cycles



Operating Strategies

- Lower frequency f
  → instant but exec. Slow
- Incr. voltage V
  → need to wait for V to ramp up,
  then change f
- Lower f & incr. V
  → but be smart:
  - Lower f, keep running
  - meanwhile ramp up V
  - Once V stable, incr. f
- Challenges
  - Anticipation of bursts of high-P instr. → AES in VLC
  - Same for SIMD in SPEC CPU





Assessing X86 Potential

Intel Core i9-9900K

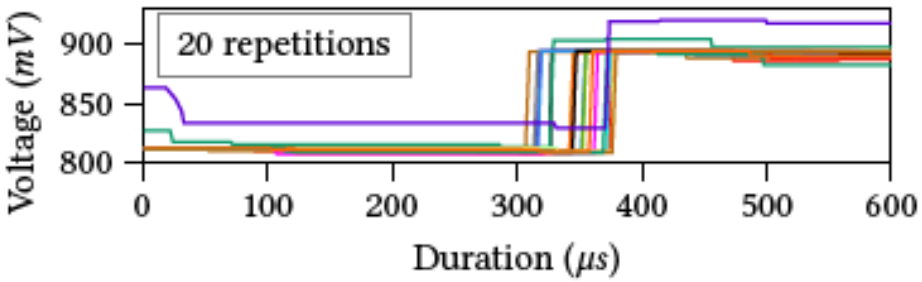
- Voltage t=0: V_{low} instr. faults, ramp up to V_{high}
  - Delay before V_{high} stable
  - t= 350-379us →then re-execute faulted instr.
- Frequency: Delay 20us (CPU stalled) → then re-execute

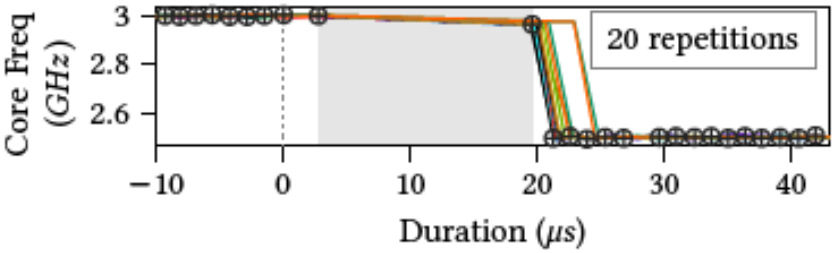
AMD Ryzen 7 7700X

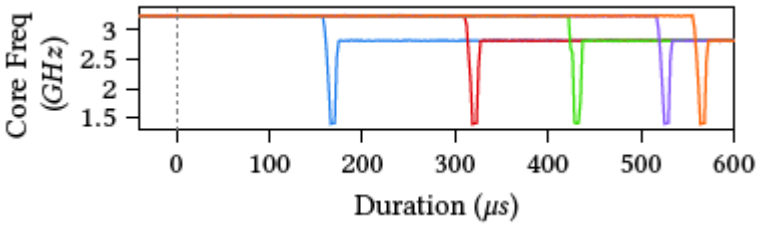
- Voltage delay ~ 668us (CPU *not* stalled)

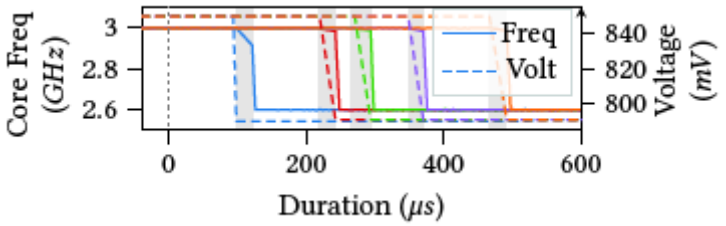
Intel Xeon Silver 4208

- 1st V change ~ 335us
- 2nd f change ~ 31 us (CPU stall for 27us)



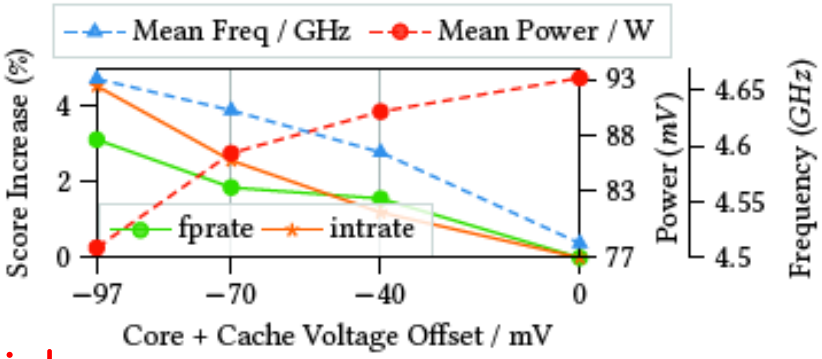




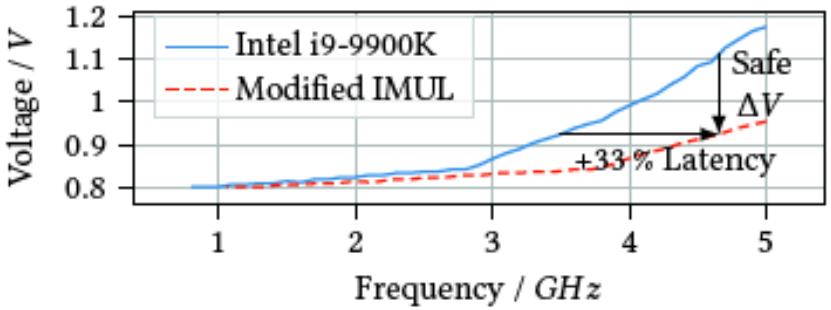


Develop Safe Guardband

- Depends on undervolting offset
  - Change in SPEC CPU scores
  - Higher potential for Xeon, AMD
  - Can speed up code → why?
  - ➢ Lower f+P=lower temp → room to TDP → can incr f again!



- IMUL 3→4 cycles: can lower V
  - 0-1.6% performance loss
- Need aging guardband → 15%/10yrs
- Need temprature guardband → 3.5%
- INT faster by 0.5-2.6%
- FP slower by 4-6%
  - But potential for short float formats



	fprate	intrate	508	521	538	554	525	548
i9-9900K	−4.1 %	0.5 %	−22 %	−1.4 %	−12 %	−3.3 %	7.0 %	7.7 %
7700X	−5.9 %	2.6 %	−35 %	−5.3 %	−9.0 %	−19 %	22 %	6.8 %

fV Operating Strategy & Eval Framework

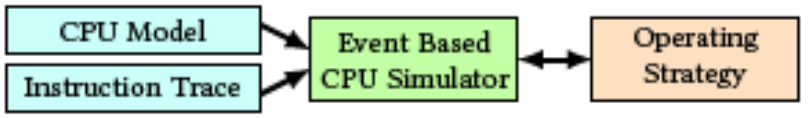
Strategy design

- p_dl: deadline b/w 2 faulting instr
- p_ts: time span for aggregate # faults
- p_ec: max. faults during p_ts to avoid trashing
- p_df: factor to incr deadline if fault during p_ts

Evaluation

- Instr. trace → simulation + strategy
  - GEM5+SPEC CPU2017
  - MSR instr disable+Linux exception

```
1  class Operating_Strategy_fV:
2    def disabled_instruction_exception_handler():
3      # we wait for the frequency to change
4      cpu.change_pstate_wait(DVFS.Cf)
5      # and request the voltage change
6      cpu.change_pstate_async(DVFS.Cv)
7
8      cpu.set_instructions_disabled(False)
9
10     # trashing prevention
11     if exception_count_in_timespan(p_ts) >= p_ec:
12       cpu.set_timer_interrupt(p_dl * p_df)
13     else:
14       cpu.set_timer_interrupt(p_dl)
15
16   def timer_interrupt_handler():
17     cpu.set_instructions_disabled(True)
18     cpu.change_pstate_async(DVFS.E)
```



CPU	x86-64, 2 Core, 3 GHz, O3 (Out-Of-Order) CPU
DRAM	2 Channel, 3 GB DDR4_2400_8x8
Cache	64 kB L1I, 32 kB L1D, 2 MB LLC
gem5 Mode	Full System
OS	Ubuntu 20.04.1 with Linux kernel v5.19.0

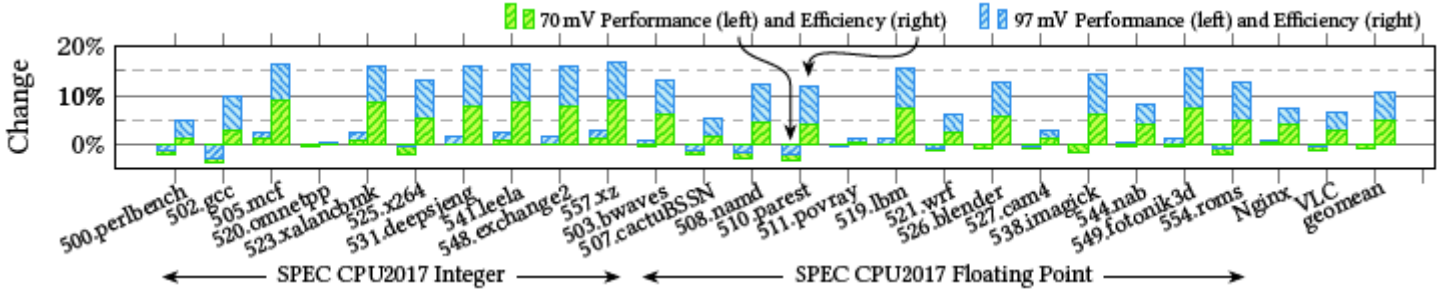
Results: Sensitivity to Architecture

- A: Intel Core i9-9900K, 1 DVSF domain total (1/4 cores)
- B: AMD Ryzen 7 7700X, 1 freq domain per core
- C: Intel Xeon Silver 4208, 1 DVFS domain per core
- Core count → best when individual DVFS domain per core
  - fV: need to control changes in both f and V, need low df delay
  - e: emulate after fault (instead of DVFS) → bad idea

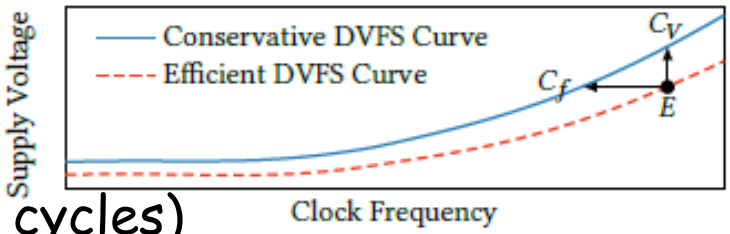
CPU	cores	OS		70 mV Undervolt						97 mV Undervolt					
				$SPEC_{gmean}$	$SPEC_{median}$	525.x264	$SPEC_{noSIMD}$	Nginx	VLC	$SPEC_{gmean}$	$SPEC_{median}$	525.x264	$SPEC_{noSIMD}$	Nginx	VLC
$\mathcal{A}_1$	fV	Pwr	−5.6 %	−7.1 %	−7.1 %	−7.1 %	−3.5 %	−3.9 %	−9.7 %	−11 %	−12 %	−15 %	−5.8 %	−6.3 %
		Perf.	−0.2 %	−1.3 %	−1.3 %	+3.0 %	+0.5 %	−0.4 %	+0.8 %	+1.3 %	0.1 %	+3.4 %	+1.2 %	+0.2 %
		Eff.	+5.7 %	+6.2 %	+6.2 %	+11 %	+4.2 %	+3.6 %	+12 %	+14 %	+14 %	+21 %	+7.4 %	+6.9 %
$\mathcal{A}_4$	fV	Pwr	−4.6 %	−0.1 %	−6.9 %	−7.4 %	−1.0 %	−1.0 %	−8.9 %	−8.7 %	−13 %	−16 %	−1.6 %	−1.6 %
		Perf.	−3.9 %	0.0 %	−7.9 %	+1.8 %	−0.3 %	−0.6 %	−3.6 %	−3.5 %	−7.2 %	+1.8 %	−0.1 %	−0.5 %
		Eff.	+0.7 %	0.1 %	−1.0 %	+10.0 %	+0.7 %	+0.4 %	+5.8 %	+5.7 %	+6.7 %	+22 %	+1.5 %	+1.1 %
$\mathcal{A}_\infty$	e	Pwr	−7.5 %	−7.6 %	−5.4 %	−7.5 %	−7.2 %	−7.2 %	−12 %	−12 %	−10 %	−17 %	−12 %	−12 %
		Perf.	−42 %	−12 %	+6.2 %	+1.4 %	−98 %	−92 %	−42 %	−12 %	+6.1 %	+1.4 %	−98 %	−92 %
		Eff.	−37 %	−4.5 %	+12 %	+9.6 %	−98 %	−91 %	−34 %	+0.6 %	+18 %	+22 %	−98 %	−91 %
$\mathcal{B}_\infty$	f	Pwr	−8.1 %	−7.8 %	−7.8 %	−9.1 %	−4.4 %	−4.4 %	−12 %	−11 %	−11 %	−14 %	−6.7 %	−6.7 %
		Perf.	−7.8 %	−7.8 %	−9.2 %	+0.4 %	−2.5 %	−2.5 %	−10 %	−11 %	−12 %	+0.6 %	−2.3 %	−2.3 %
		Eff.	+0.3 %	0.0 %	−1.6 %	+11 %	+2.0 %	+2.0 %	+1.4 %	0.1 %	−1.6 %	+17 %	+4.7 %	+4.7 %
	e	Pwr	−9.2 %	−8.0 %	−11 %	−9.2 %	−9.8 %	−9.8 %	−14 %	−13 %	−16 %	−14 %	−15 %	−15 %
		Perf.	−26 %	−5.1 %	+15 %	−0.5 %	−96 %	−80 %	−26 %	−5.2 %	+19 %	0.0 %	−96 %	−80 %
		Eff.	−19 %	+3.1 %	+28 %	+9.5 %	−95 %	−78 %	−14 %	+9.3 %	+41 %	+17 %	−95 %	−76 %
$\mathcal{C}_\infty$	fV	Pwr	−5.6 %	−7.1 %	−7.1 %	−6.1 %	−3.6 %	−4.0 %	−9.8 %	−11 %	−12 %	−14 %	−5.8 %	−6.6 %
		Perf.	−0.8 %	−1.9 %	−1.9 %	+3.5 %	+0.3 %	−1.1 %	+0.2 %	+0.2 %	−0.6 %	+3.8 %	+1.0 %	−0.6 %
		Eff.	+5.1 %	+5.5 %	+5.5 %	+10 %	+4.0 %	+3.0 %	+11 %	+13 %	+13 %	+21 %	+7.3 %	+6.4 %

Efficiency Gains

- -70mV (green), -97mV (blue)
- Efficiency e=1/(dt*dP)
- Avg. efficiency gain 11% (efficient DFVS 72% of time)



Co-design for Undervolting…



- Redefines guardband → allows lower V
- Requires co-design
  - Changed instruction set (IMUL 3→4 cycles)
  - MSR protection of high-P instr
  - Exception support in OS → handler implements fV policy
- More sustainable computing → lower operating costs (win-win:)
  - Less power consumption
  - Little impact on performance (sometimes even faster)
- On-going work w/ GPUs, short data formats
- Ideas for facility power management
  - Allow P over-provisioning
  - Dyn cap power profiles to maintain stability and meet cost targets

My (our) 30 years of tools, runtime systems,…

- FSU

- HUB

- LLNL

- NCSU

- Parallel and Distributed Systems
  - Anomaly detection/prediction
  - Reliability, availability, serviceability
- Programming Lang./Compilers
  - Data-intensive computing
  - Machine learning
  - Heterogeneous computing
  - Software reliability and bug analysis
- Computer Arch / Operating Systems:
  - Extreme Storage systems
  - Deep Memory Hierarchies
  - Virtualization, Multi-core
- **Quantum Computing**
  - Noise mitigation
  - Domain-spec. languages
  - Simulation
  - Compilation
- **Big Data and Cloud Computing**
  - Machine learning middleware
  - Data-intensive computing
  - Enterprise Storage Systems
- **High Perf. Comp.**
  - Performance analysis/tuning
  - Fault-tolerant HPC
  - Parallel I/O
  - Power-aware HPC
  - Heterogeneous (GPUs, FPGAs
- **Embedded /Real-Time**
  - Real-time scheduling
  - Timing analysis
  - Embedded/real-time OS
  - Cyber-physical systems

USENIX ATC'93

- 1st POSIX threads implementation ever
- Needed by Culler's Network of Workstations (NOW) project
- Gnu Ada compiler needed it for validation (tasks)
- Later: DSM-Threads (RTS4PP'98 → HIPS)

A Library Implementation of POSIX Threads under UNIX

Frank Mueller[1] – Florida State University

ABSTRACT

Recently, there has been an effort to specify an IEEE standard for portable operating systems for open systems, called POSIX. One part of it, the POSIX 1003.4a threads extension (Pthreads for short) [12], describes the interface for light-weight threads that rely on shared memory and have a smaller context frame than processes.

This paper describes and evaluates the design and implementation of a library of Pthreads calls that is solely based on UNIX. It shows that a library implementation is feasible and can result in good performance. This work can also be used as a comparison of the performance of other implementations, or as a prototyping, testing, and debugging system in the regular UNIX environment. Finally, some problems with the Pthreads standard are identified.

Binary rewriting to extract shmem traces

- TOPLAS'06, PPoPP'06, ICS'05+04, CGO'03
- Cache coherence bottlenecks in OpenMP

METRIC: Tracking Down Inefficiencies in the Memory Hierarchy via Binary Rewriting *

Jaydeep Marathe[1], Frank Mueller[1], Tushar Mohan[2],
Bronis R. de Supinski[4], Sally A. McKee[3], Andy Yoo[4]

[1] Dept. of Computer Science
North Carolina State University
Raleigh, NC 27695-7534

[2] School of Computing
University of Utah
Salt Lake City, UT 84112

[3] School of ECE
Cornell University
Ithaca, NY 14853

[4] Lawrence Livermore National Lab
Center for Applied Scientific Computing
L-561, Livermore, CA 94551

mueller@cs.ncsu.edu, phone: (919) 515-7889

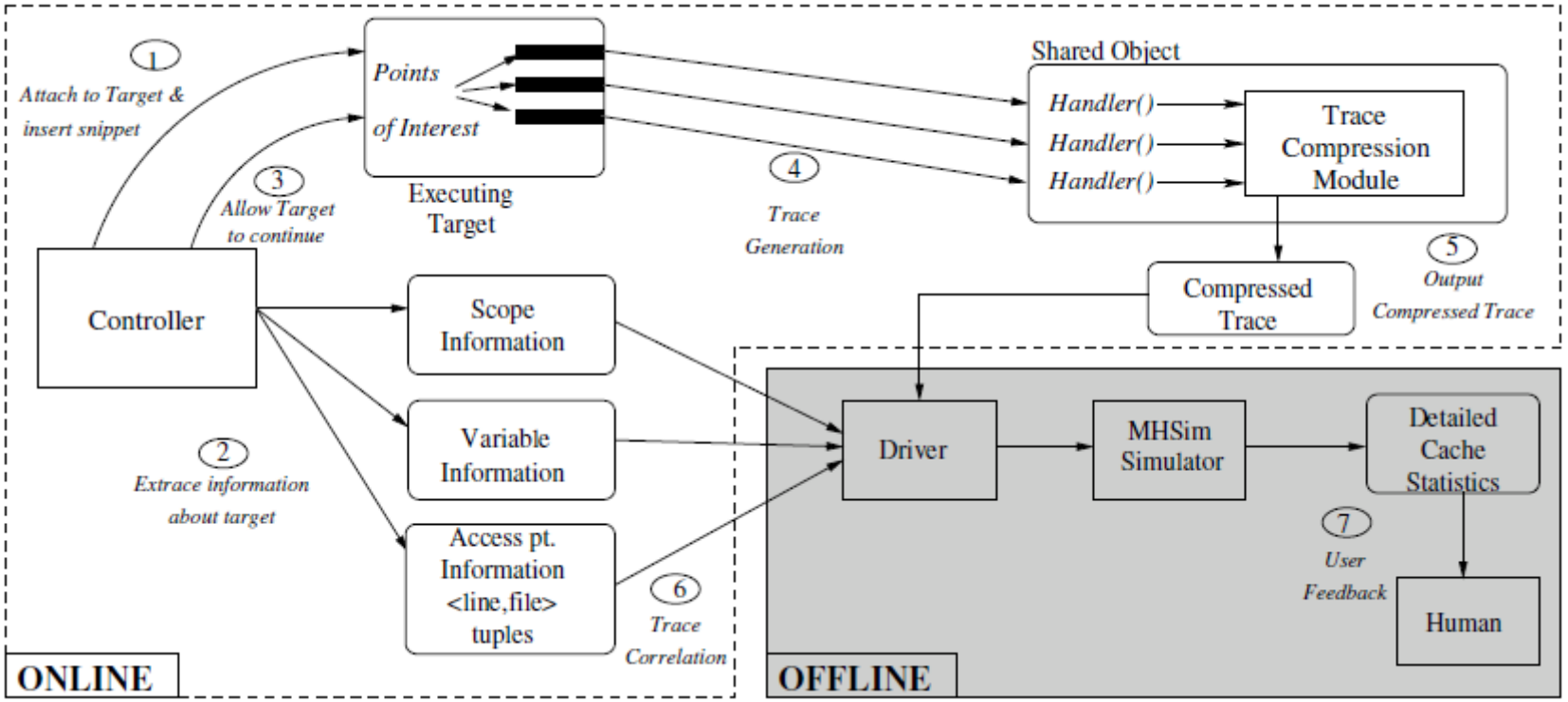
Abstract

In this paper, we present METRIC, an environment for determining memory inefficiencies by examining data traces. METRIC is designed to alter the performance behavior of applications that are mostly constrained by their tween processor speeds and memory latencies. Thus, locating and eliminating sources of inefficiencies in the memory hierarchy can potentially impact application performance to a significant degree.

Incremental memory hierarchy simulation by capturing the address trace of an application is a highly accurate

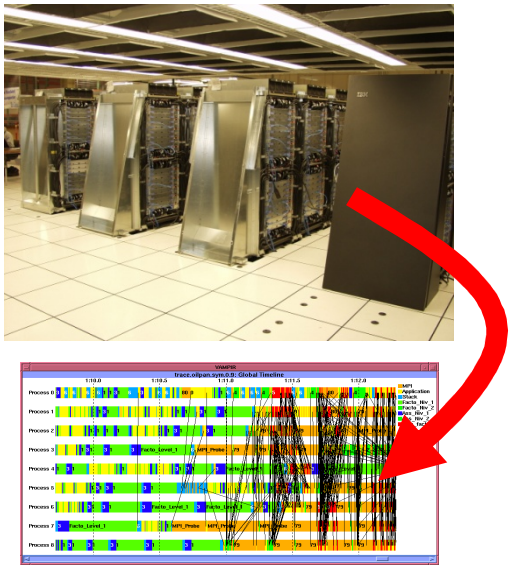
Binary rewriting to extract shmem traces

- TOPLAS'06, PPoPP'06, ICS'05+04, CGO'03
- Cache coherence bottlenecks in OpenMP
- PRSD: Power regular section descriptor
- Itanium+later x86 PEBS (probabilistic event-base sampling)



ScalaTrace: MPI Traces

- Scalably capturing full trace of communication
  - Near constant trace sizes for some apps (others: more work)
  - Near constant memory requirement
- Rapid analysis via replay mechanism (w/o app)
  - Record Δt, retaining timing → scalable
  - Fast timeline search, easy outlier detection
- ➢ Lossless MPI tracing, any # node feasible
  - May store&visualize MPI traces on desktop

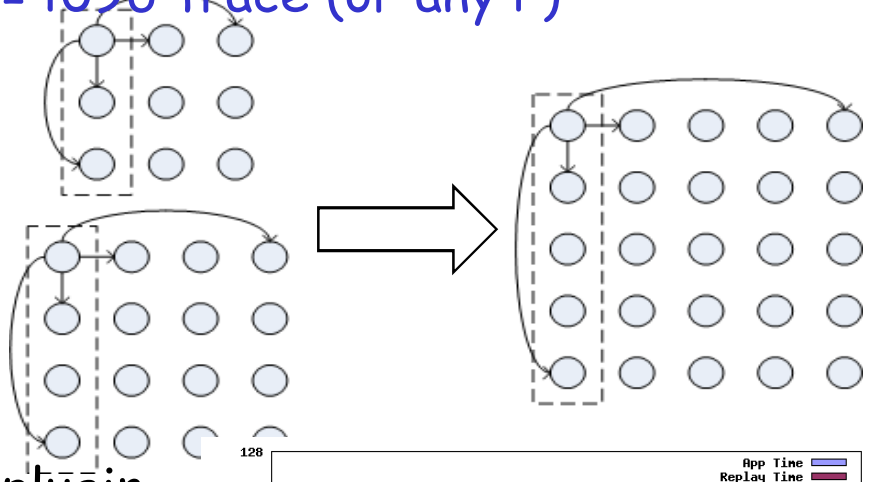


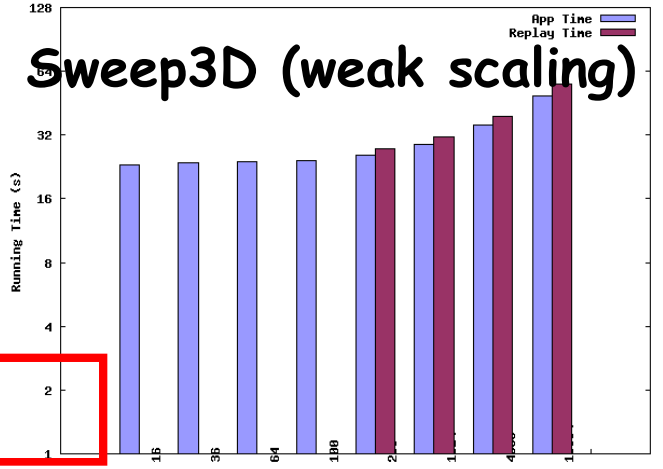
- ScalaTrace [IPDPS'07] best paper
- Timed relay [ICS'08]
- Code availabile under BSD license:
  moss.csc.ncsu.edu/~mueller/scala.html
- NCSU: Mike Noeth, Prasun Ratn; LLNL: Bronis de Supinski, Martin Schulz

ScalaExtrap: Comm+IO Extrapolation

PPoPP'11+IPDPS'17(IO)

- Motivation: Comm.+IO analysis at scale – without running app!
- Idea: synthetically generate **elastic** comm. traces:
  - P=8,16,32,64 nodes trace → P=4096 trace (or any P)
- Replay large trace/analyze it
- Challenges:
  - Topology detection
  - Message payloads, IO files
  - Time extrapolation
- Workload: any MPI, stencil/mesh+plugin
- Machines: Clusters/HPC
- Resources: Comm.+I/O+compute
- Scale: 16k nodes BG/P: > 92% Accuracy





Extrapolation not so elusive anymore

(3) Signature Clustering [ICS'14]

Objective:

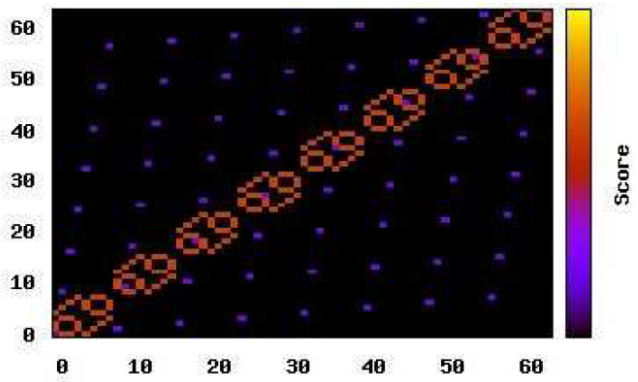
- trace few (representative) nodes
- replay over all nodes

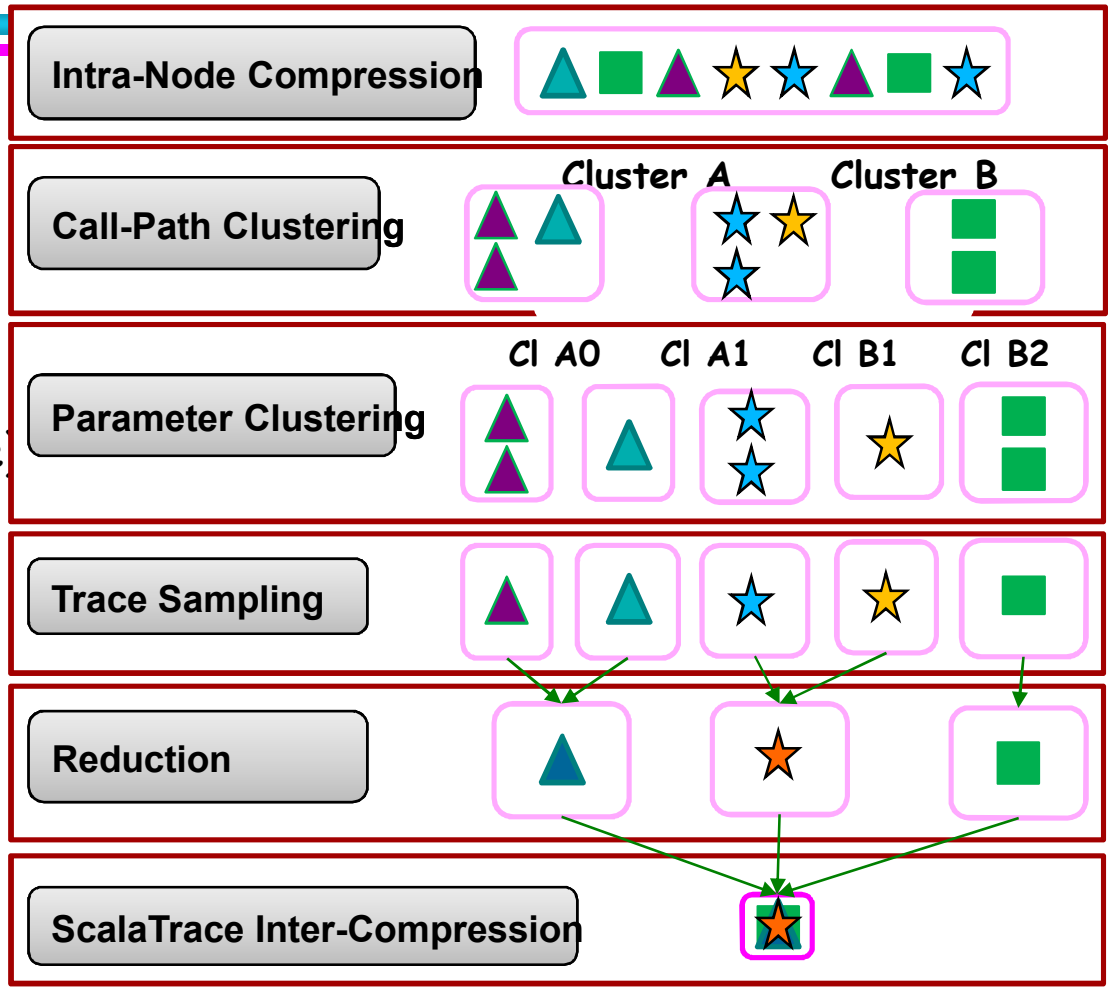
Hierarchical:

1. Call path Clustering: stack signature (XORed backtrace)
2. Parameter Clustering: parameter signature (range composed)

User plug-in functions

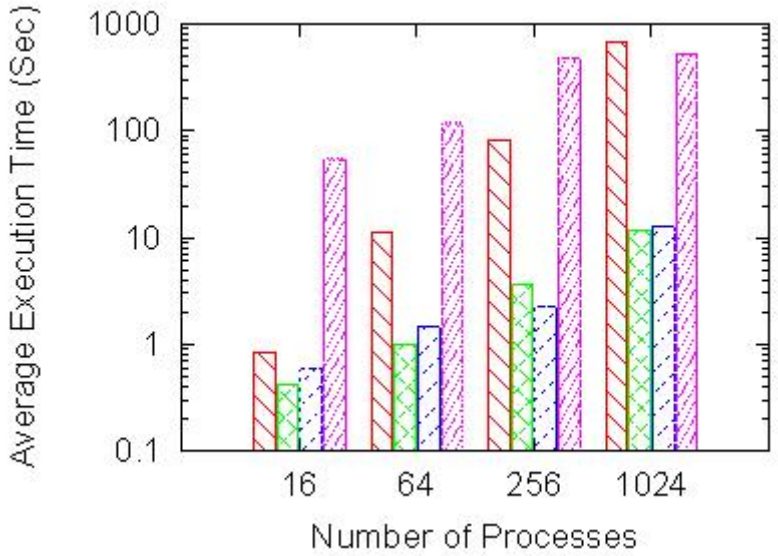
- 2nd comm. pattern in *CG*:

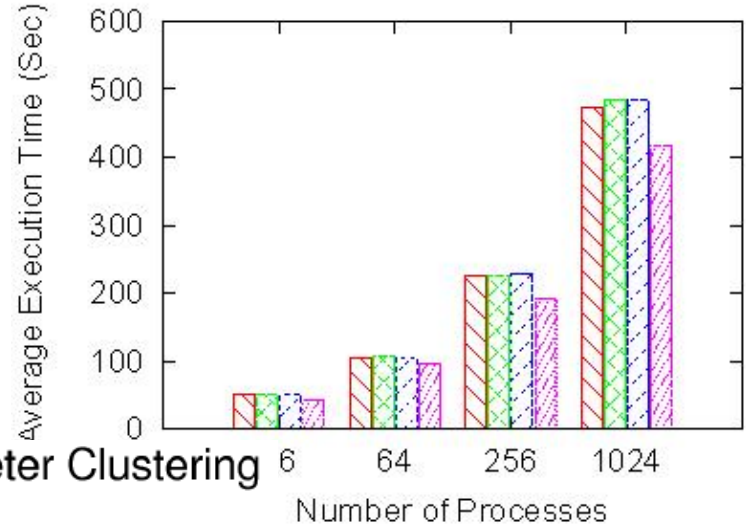




Clustering Contributions

- Novel hierarchical clustering algorithm
  - log P time complexity+low overhead
- Compared to w/o clustering
  - 1 order of magnitude less exec. overhe
  - 2-3 orders of magnitude less space
- Trace accuracy:
  - Over 85% for strong scaling
  - Over 93% for weak scaling
- Hierarchical clustering
  → very effective technique
  - suitable for extreme-scale computing





Without Clustering
Call-path+Parameter Clustering
Reference Clustering
Application Time

HPC Resilience

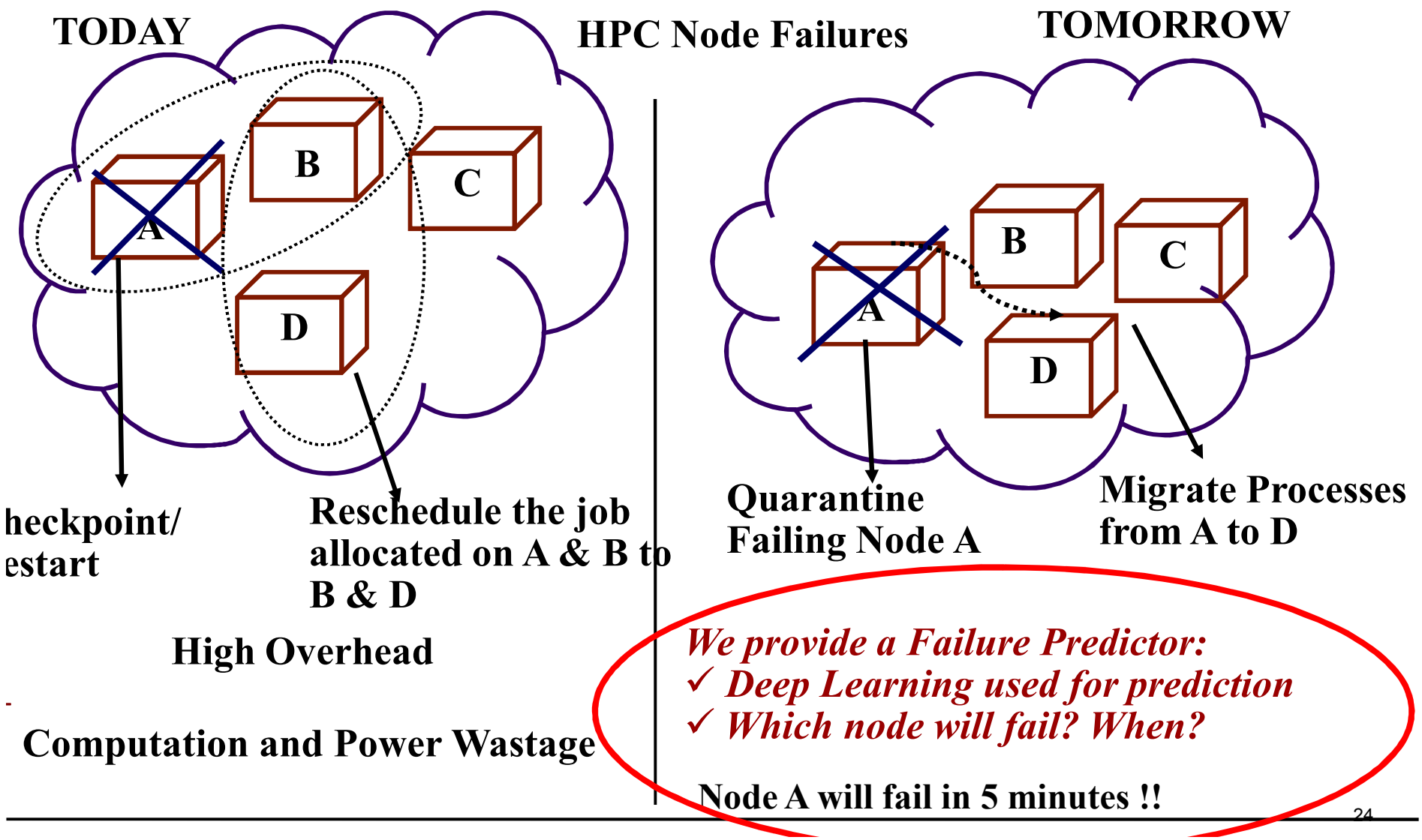
1. Scalable network overlay (ICS'06)
   - track live nodes, group communication
2. Reactive fault tolerance (IPDPS'07, Linux'11, ICPADS'11)
   - job pause → 70% reduced resubmit overhead
   - Incr. Chkpts → 1:9 full/incr. Ratio best, reduce I/O
3. Proactive fault tolerance (ICS'07, SC'08, JPDC'12)
   - process virt. → ½ overhead of OS, health monitor
   - live migration → ½ # chkpts
   - back migration → wins if >10% work left
4. Redundancy + SDC Handling (ICDCS'12, SC'12, Cluster'15, ICS'16)
   - 2x # nodes → 2x # jobs: capacity not capability comp.
   - dual for SDC check / triple SDC correction (msgs, RAM, I/O)
5. Algorithm-based Fault tolerance (IPDPS'14. Chen et al., HPDC'15)
   - Complements above, sign. less overhead, only dense linear algebra
   - Model SDC for numerical algorithms → Sandbox: run thru errors
6. Predict which exact node will fail when (HPDC'18+SC'18)
   - Enables proactive actions → can reduce C/R frequency even more

➢ Code contributed to BLCR, available for Open MPI, later RedMPI

Machine Learning to Predict Failures in HPC

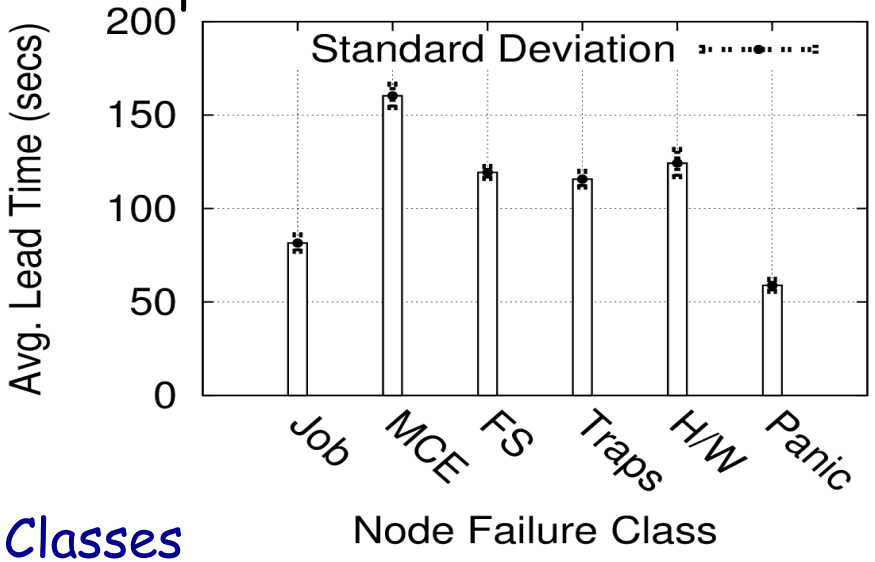
- Collaboration w/ several Nat'l Labs
  - ORNL Summit Supercomputer: #1 fastest in the world
- Problem: failures happen, compute nodes stop working
  - Same as Cloud computing when you scale out
  - more nodes/components → more failures
- Hypothesis: can predict failures timely for evasive action

Proactive Fault Tolerance



How to Predict Failures in HPC [SC'18,HPDC'18…]

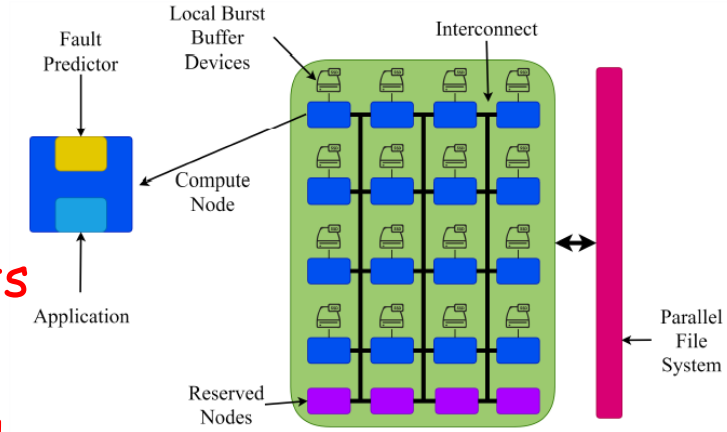
- Approach: Long-Short-Term-Memory based machine learning
  - Train to recognize anomaly patters on systems logs
  - Predict which component is about to fail
  - Determine location of failing component
  - Take evasive action, e.g., migrate computation to new node
- 2-3 min. lead time to failure
  - Accuracy ≥ 83%, F1 <= 89.99%
  - FP Rate:16.66% to 25%
  - Proactive: cloning (90 secs), job migration (13-24 secs)



- Lead Time Sensitivity with Failure Classes
  - Lead times to *Kernel Panics* are short (58.87 secs)
  - *MCE* and *Hardware-caused* failures longer keads (124-160 secs)

HPC Resilience + ML to Predict Faults (Subhendu Behera, ORNL+LLNL)

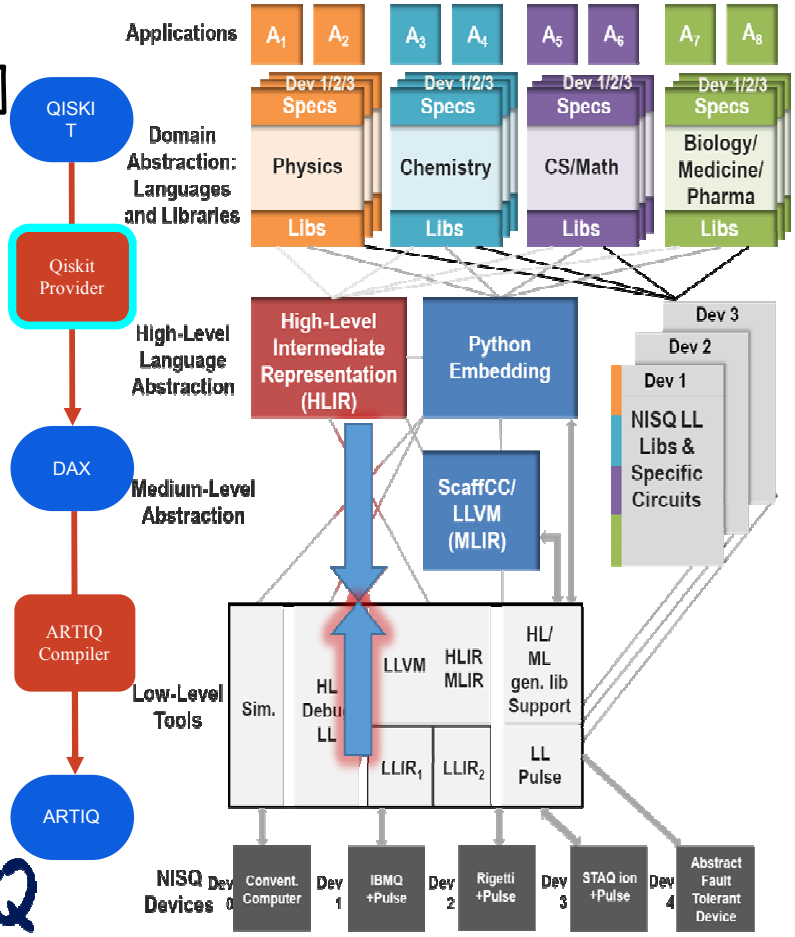
- Problem: failures happen, compute nodes stop working
- Hypothesis: can predict failures timely for evasive action
- Approach: Long-Short-Term-Memory based ML [HPDC'18,SC'18,…]
- Live migration vs. checkpointing w/ failure prediction [HPDC'20]
  - considers burst buffers
  - 20-86% reduced overhead



- IPDPS'22: asynchronous safeguard chkpts
  - Prioritized, add'l savings 4-60%
- Fault propagation within Flux job scheduler
  - within+across jobs → coordinated
  - dynamically adaptive workflow scheduling
  - persistent storage abstraction for workflow resilience

Quantum HW/SW Stack

- Part of NSF STAQ (Duke) and NSF QLCI RQS (UMD)
- Domain-specific quantum abstractions: SAT-problems, Physics, Chemistry [SC'22]
- Circuit opts.: at gate+pulse levels
- Algo+noise-aware problem solving [QCE'22
- Semantic tracking during pgm translation
- HW/SW stack for ion traps [QCE'23]
- Create/exploit multi-qubit native gates
- Sparse tensor networks for quantum simulation → SIMD+multi-node



NC STATE UNIVERSITY
Department of Computer Science

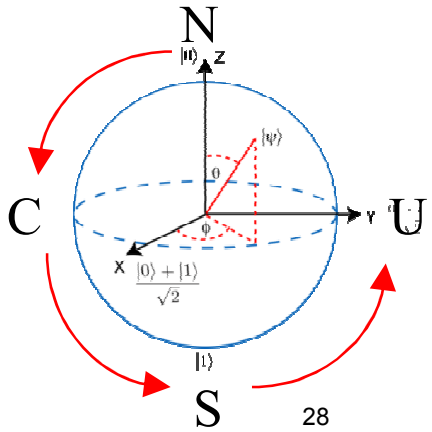
Institute for Robust Quantum Simulation

STAQ

Thanks for the Invitation! Questions?

Pubs:

https://moss.csc.ncsu.edu/~mueller/publications



NC STATE UNIVERSITY
Department of Computer Science