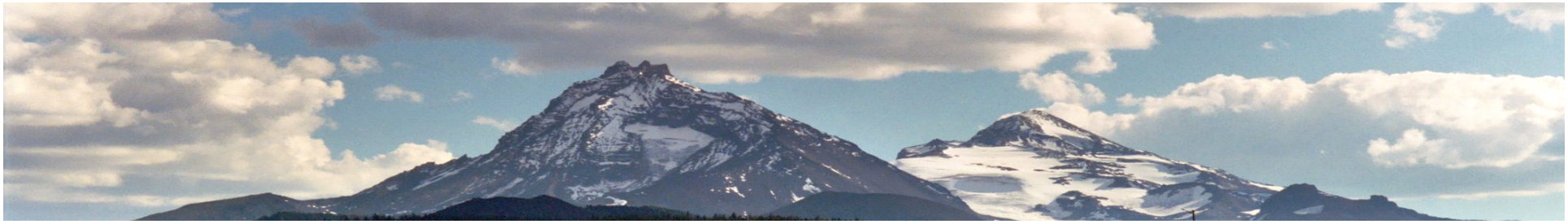


Lecture 8:
TAU Advances

Allen D. Malony
Department of Computer and Information Science

UNIVERSITY OF OREGON

Outline

- General hybrid parallel profiling
- Cross-platform OpenMP performance analysis
- Empirical autotuning

General Hybrid Parallel Profiling

- Two types of measurement approaches
  - Sampling-based measurement (SBM)
    - measure performance by statistical observation
    - a.k.a. statistical sampling or event-based sampling
  - Probe-based measurement (PBM)
    - measure performance by insertion of code into a program
- Is it possible to combine approaches?
- Idea is to investigate a hybrid approach
  - Integrate sampling-based and probe-based methodology
  - Improve performance observability

A. Morris, A. Malony, S. Shende, K. Huck, "Design and Implementation of a Hybrid Parallel Performance Measurement Systems," International Conference on Parallel Processing (ICPP), San Diego, Septemer 2010.

A. Malony, K. Huck, "General Hybrid Parallel Profiling," International Conference on Parallel, Distributed, and Network-based Processing (PDP), Turin, Italy, February 2014.

Approach Assumptions

- SBM can measure/attribute performance accurately if:
  - Code segments are executed repeatedly
  - Execution is long enough to collect a large # of samples
  - Sampling frequency is uncorrelated with execution

    ⇨ SBM is grounded in *statistical sampling theory*
- PBM can measure performance accurately if:
  - Overhead of executing the measurement code is sufficiently small relative to the size of the entity being measured
  - Intrusion due to measurement overhead in one thread does not affect the performance of any other threads

    ⇨ PBM is grounded in *measurement theory*

Hybrid Performance Measurement

- Combine SBM and PBM techniques
  - Look for opportunities for synergies

Alternatives

1) Just use both at the same time, but independently
   - works fine (as long as measurements tools are compatible)
   - lose benefit of integration between SBM and PBM data
2) Base combined system on SBM
   - Dynamic statistical sampling of PBM data
   - Example: mpiP (MPI communication), HPCToolkit, ...
3) Base combined systems on PBM
   - Augment PBM view with SBM data
   - Dynamic measures with event correlation from PBM

Hybrid Trace Measurement (TAUebs)

- Inspired by Barcelona Supercomputing Center (BSC) work
- Combines TAU, PerfSuite, and HPCToolkit
  - TAU for probe-based instrumentation and measurement
  - PerfSuite technology for timer-based sampling
  - HPCToolkit for call stack unwinding on fully-optimized codes
- Foundation is TAU (PBM) with linked SBM capabilities
  - Synergy through "context" linking
  - Determination of dynamic offset from event start (BSC)
- Tradeoff of degree of instrumentation versus SBM type 
  - Utilize SBM to observe fine-grained performance
  - Use SBM to sample performance not easily obtained
- Augment PBM with SBM performance views

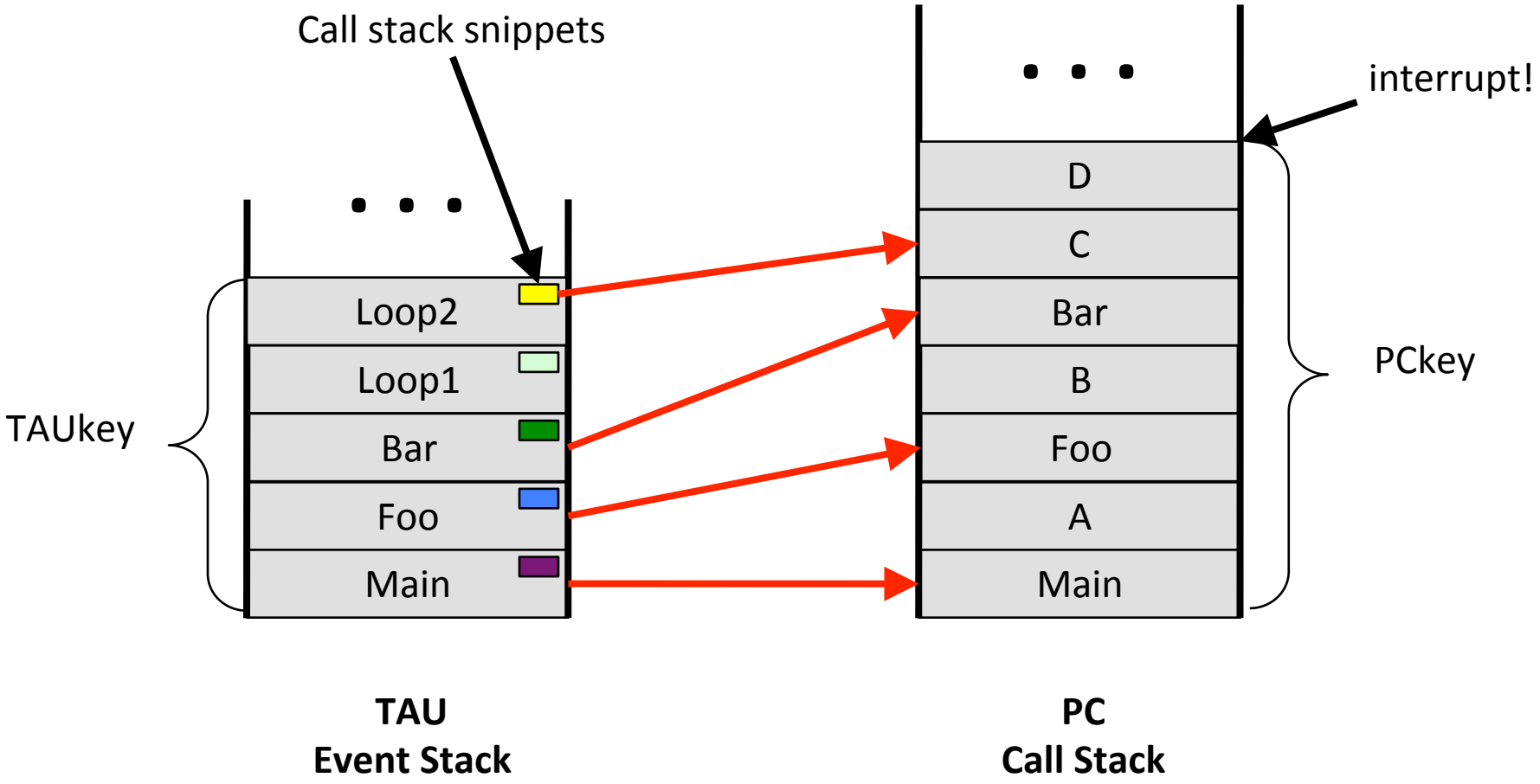
The Notion of Context in TAUebs

- SBM methods capture context on interrupts (traps)
  - Program counter (PC) and possibly PC call path
    - walk the call stack to see all active *routines*
  - Performance data and other state information
  - Together it gives the sample context (*PC context*)
- PBM methods also have context
  - Current event and possibly a TAU event path
    - any instrumented *event* (routine or user-defined)
    - walk the event stack to see all active events (TAU event path)
  - Performance data and other state information
  - Together it forms the event context (*TAU context*)

Context Merging

- Get more information if can merge contexts
- At interrupt, SBM can “reach over” into PBM
  - Current TAU context contains event state
  - PC context can be qualified with TAU context
    - think about "sampling" TAU context and save with PC context
- There may be overlap in PC call path and TAU event path
  - Offers opportunity to optimize retention of context information
  - Need to figure out how to resolve overlap
  - Alternative 1: store both
    - problems: handle C++ name mangling, non-routines not handled
  - Alternative 2: walk PC call stack to event at top of event stack
    - problem: need to make efficient at runtime

TAU Event Stack and PC Callstack



Loop1 and Loop2 are not routines!

TAUebs Measurement

- What measurement is made with each sample?
- Capture a trace of EBS samples
- Each TAUebs sample contains:
  - Timestamp
  - *TAUkey* (key to current TAU event stack)
  - *PCkey* (key to current PC call stack)
  - Hardware counters
  - Delta (optional)
    - subtract start time of top TAU event from timestamp
    - support BSC code folding analysis

TAUebs Data Analysis (Trace)

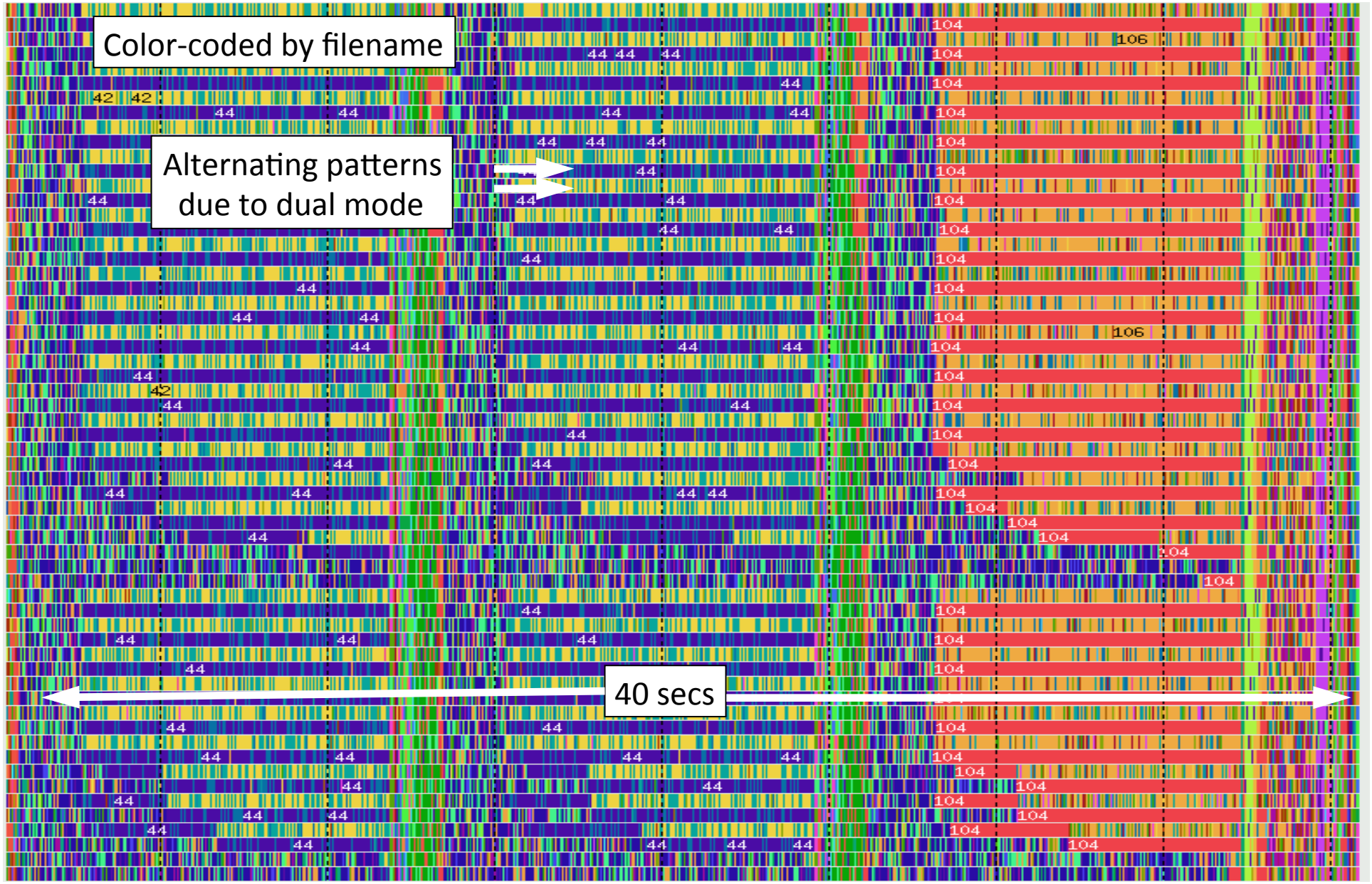
- EBS to OTF trace converter
- Analyze EBS trace with trace analysis tools
- For each sample
  - Place timestamp in trace record
  - Merge TAU event stack and PC call stack
    - merged call path
  - Create event ID for merged call path
    - put in trace record
  - Put collected PAPI metrics in trace record
  - Can store PC locations in trace record as well

TAUebs FLASH Experiments on IBM BG/P

- FLASH is a nuclear astrophysics simulation
- Compare measurement overhead
  - TAU only
  - Hybrid
- Adding EBS tracing leads to high overhead at high core counts

Description	Procs	Runtime	Overhead
uninstrumented	240	636s	–
TAU measurement	240	648s	12s (1.9%)
TAU measurement and sampling	240	672s	36s (5.6%)
uninstrumented	484	674s	–
TAU measurement	484	689s	15s (2.2%)
TAU measurement and sampling	484	713s	39s (5.8%)
uninstrumented	1004	649s	–
TAU measurement	1004	670s	21s (3.2%)
TAU measurement and sampling	1004	697s	48s (7.4%)
uninstrumented	2176	656s	–
TAU measurement	2176	695s	39s (5.9%)
TAU measurement and sampling	2176	699s	42s (6.6%)
uninstrumented	4416	669s	–
TAU measurement	4416	729s	60s (9%)
TAU measurement and sampling	4416	771s	102s (15.2%)
uninstrumented	8192	729s	–
TAU measurement	8192	847s	118 (16.2%)
TAU measurement and sampling	8192	863s	134s (18.4%)
uninstrumented	15812	781s	–
TAU measurement	15812	997s	216s (27.7%)
TAU measurement and sampling	15812	1144s	363s (46.5%)

TAUebs Trace for FLASH (240 processes)

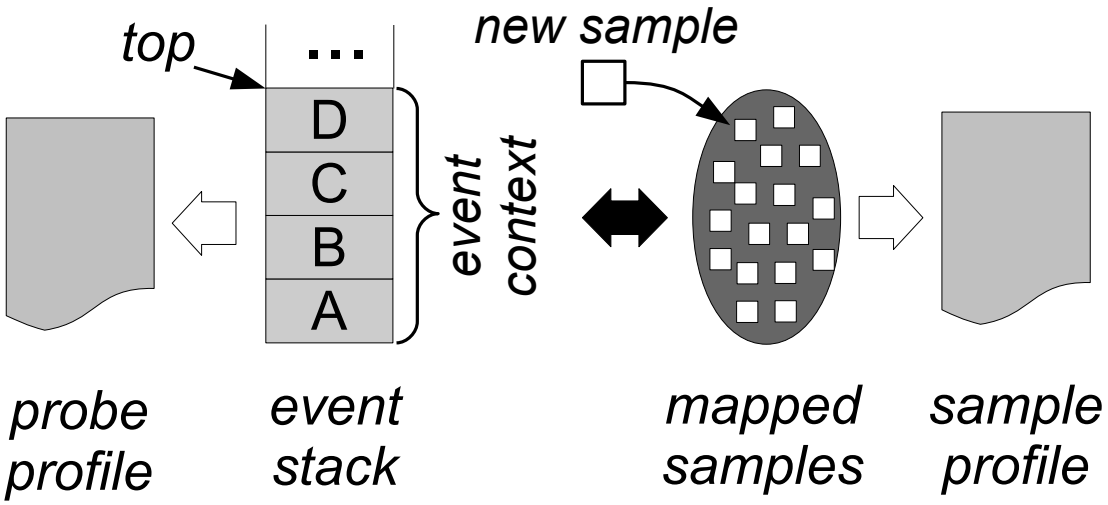


Hybrid Profiling

- Hybrid tracing with TAUebs is not scalable
  - Need to have a parallel profiling methodology
- However, hybrid profiling is more challenging
  - Online analysis to is more complex
- Can leverage the hybrid ideas in TAUebs
  - Associate samples with the active probe context
- Goal is to capture all of the performance information possible at runtime to produce equivalent profile results from trace-based analysis

Approach

- TAU profiling
- EBS at 100ms
  - Per thread
  - Determine PC
  - Unwind callstack
  - Use Linux *backtrace* facility
- Sample mapping
  - Determine event context (TAU context)
  - Update hybrid profile for every unique tuple: {event context, PC, call path}
  - Done for every thread of execution



Hybrid Profile Generation

- At the end of application's execution
  - Sample addresses are resolved to symbolic names
    - use GNU binutils (if fail, display as "UNRESOLVED")
  - Profile metric statistics are calculated
  - Hybrid profiles are written to files
- For each probed event context for each thread
  - Profile entry created to represent samples in the context
    - each unique sample represented as "SAMPLE" event
    - "SUMMARY" entry aggregates across samples
  - "context events" share the name of probed parent
    - labeled as "CONTEXT" in the profile
  - "UNWIND" event nodes are call sites that eventually end with a sample event

A Simple Hybrid Profiling Example

- Sequential matrix multiplication
  - 3 matrix sizes: 128x128, 256x256, 512x512
  - Use small routines that are hard to probe
- Insert probes to record phases and parameters
  - `PHASE_START`
    `PHASE_STOP`
  - `PARAM_PROFILE_START`
    `PARAM_PROFILE_STOP`
- Compare sampling-only, probe-only, and hybrid
- Machine: Intel Xeon X5650, 2.67 GHz, 72 GB

Sequential Matrix Multiplication

```
main() {
  int sizes[] = {128, 256, 512};
  for (i = 0; i < max_iter; i++) {
    PHASE_START;
    for (s = 0; s < 3; s++) {
      int size = sizes[s];
      a = createMatrix(size);
      b = createMatrix(size);
      c = createMatrix(size);
      PARAM_PROFILE_START;
      matrixMultiply(a, b, c, size);
      PARAM_PROFILE_STOP;
      freeMatrix(a, size);
      freeMatrix(b, size);
      freeMatrix(c, size);
    }
    PHASE_STOP;
} }
void matrixMultiply(double **a,**b,**c, int size) {
  int i,j,k;
  double temp;
  for (i=0; i<size; i++) {
    for (k=0; k<size; k++) {
      for (j=0; j<size; j++) {
        temp = multiplyElement(a[i][k], b[k][j]);
        c[i][j] = addElement(c[i][j], temp);
} } } }
```

Sample-only

Name	Exclusive TIME	Inclusive TIME	Calls	Child Calls
main [{matmult.cpp} {46}]	14.772	14.772	1	0
[CONTEXT] main [{matmult.cpp} {46}]	0	14.77	1,477	0
[SAMPLE] matrixMultiply(double**, double**, double**, int) [{matmult.cpp} {37}]	6.38	6.38	638	0
[SAMPLE] addElement(double, double) [{matmult.cpp} {25}]	5.03	5.03	503	0
[SAMPLE] multiplyElement(double, double) [{matmult.cpp} {27}]	2.69	2.69	269	0
[SAMPLE] matrixMultiply(double**, double**, double**, int) [{matmult.cpp} {36}]	0.62	0.62	62	0
[SAMPLE] UNRESOLVED /lib64/libc-2.12.so ADDR 0x39f3e79d5d	0.03	0.03	3	0
[SAMPLE] UNRESOLVED /lib64/libc-2.12.so ADDR 0x39f3e791ed	0.01	0.01	1	0
[SAMPLE] createMatrix(int) [{matmult.cpp} {10}]	0.01	0.01	1	0

Probe-only

Name	Exclusive TIME	Inclusive TIME	Calls	Child Calls
main [{matmult.cpp} {46}]	0	14.748	1	5
Iteration[0]	0.006	2.95	1	3
Iteration[1]	0.005	2.952	1	3
Iteration[2]	0.006	2.949	1	3
Iteration[3]	0.005	2.95	1	3
Iteration[4]	0.004	2.947	1	3
matrixMultiply [{matmult.cpp} {32} <size> = <128>]	0.04	0.04	1	0
matrixMultiply [{matmult.cpp} {32} <size> = <256>]	0.323	0.323	1	0
matrixMultiply [{matmult.cpp} {32} <size> = <512>]	2.58	2.58	1	0

Hybrid

Name	Exclusive TIME	Inclusive TIME	Calls	Child Calls
main [{matmult.cpp} {46}]	0	14.756	1	5
Iteration[0]	0.007	2.953	1	3
Iteration[1]	0.005	2.951	1	3
Iteration[2]	0.005	2.951	1	3
Iteration[3]	0.005	2.951	1	3
Iteration[4]	0.005	2.951	1	3
matrixMultiply [{matmult.cpp} {32} <size> = <128>]	0.041	0.041	1	0
matrixMultiply [{matmult.cpp} {32} <size> = <256>]	0.323	0.323	1	0
matrixMultiply [{matmult.cpp} {32} <size> = <512>]	2.581	2.581	1	0
[CONTEXT] matrixMultiply [{matmult.cpp} {32} <size> = <512>]	0	2.58	258	0
[SAMPLE] matrixMultiply(double**, double**, double**, int) [{matmult.cpp} {36}]	0.09	0.09	9	0
[SAMPLE] matrixMultiply(double**, double**, double**, int) [{matmult.cpp} {37}]	1.03	1.03	103	0
[UNWIND] matrixMultiply(double**, double**, double**, int) [{matmult.cpp} {37}]	0	1.46	146	0
[SAMPLE] addElement(double, double) [{matmult.cpp} {25}]	0.87	0.87	87	0
[SAMPLE] multiplyElement(double, double) [{matmult.cpp} {27}]	0.59	0.59	59	0

Benefits of Hybrid Profiling

- PBM can not be used observe small events
  - Overhead of measurement affects accuracy
  - High-frequency probes also accumulate overhead and can intrude on code behavior
  - Disabling events is necessary
- Hybrid profiling uses sampling to see lightweight code execution performance
- Probe context allows richer sample projection performance on program regions and semantics
- TAU's unique event naming allows semantics to be extended to hybrid profiling data

Evaluation of Hybrid Profiling Overhead

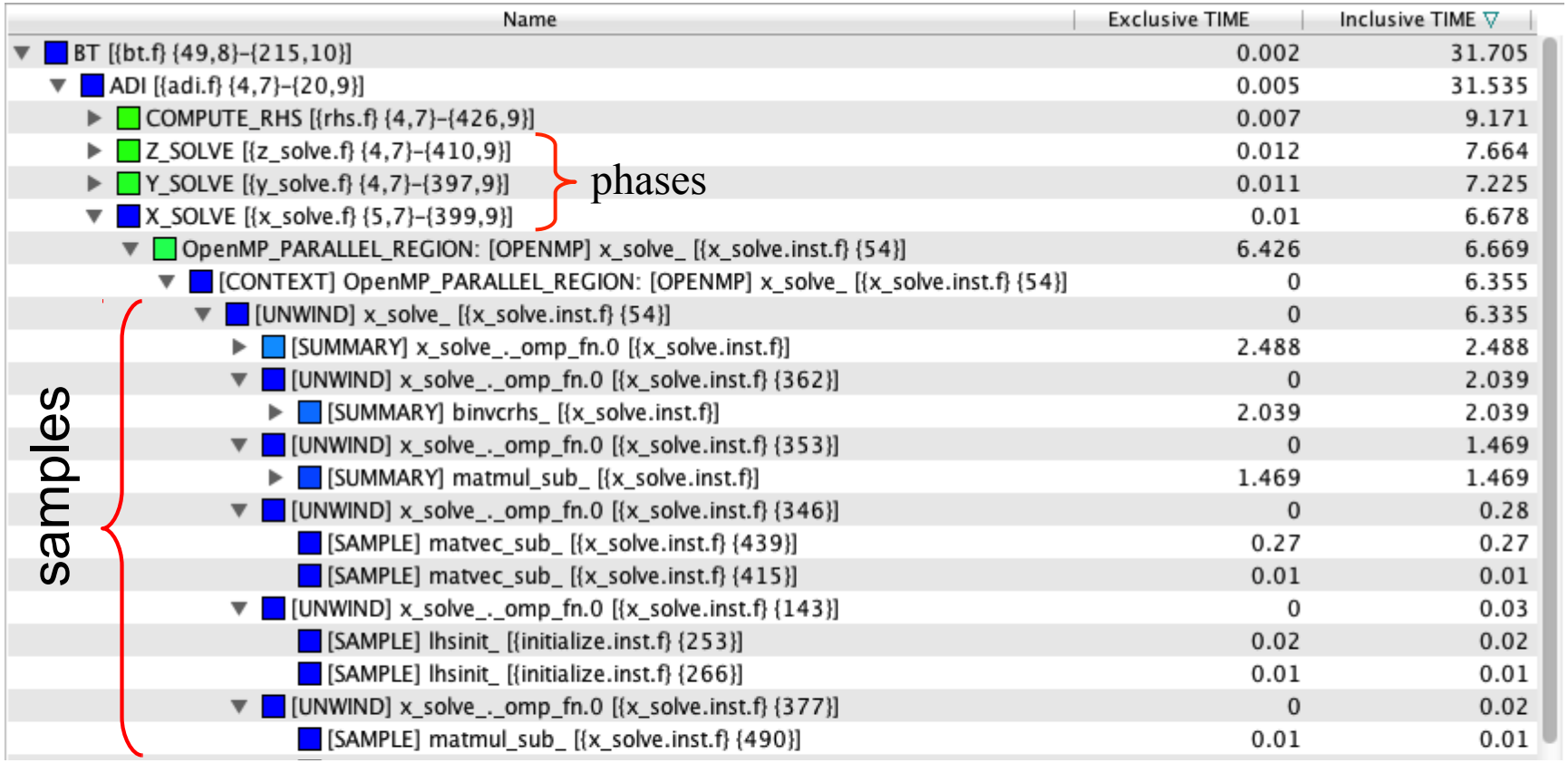
- Test on OpenMP version of NAS parallel benchmarks
- Characterize overhead against uninstrumented runs (*Clean*):
  - *Full* / *Select* (probe)
  - *Sampling*
  - *Hybrid*
- Interested in relative overhead of hybrid

BT	*Clean*	*Sampling*	*Select*	*Hybrid*	*Full*
1	246.20	248.71	250.61	244.97	2355.43
2	125.24	125.38	125.82	124.95	1644.89
4	64.90	64.67	65.26	64.49	875.45
8	37.65	37.77	37.99	38.19	1033.80
12	30.15	31.37	30.43	30.48	1002.12
EP	*Clean*	*Sampling*	*Select*	*Hybrid*	*Full*
1	61.46	64.40	61.43	63.17	61.89
2	30.96	31.81	30.88	31.76	31.03
4	15.53	16.01	15.51	15.90	15.58
8	8.10	8.34	8.11	8.33	8.15
12	5.44	5.54	5.43	5.55	5.42
FT	*Clean*	*Sampling*	*Select*	*Hybrid*	*Full*
1	51.68	51.24	52.62	51.43	56.46
2	26.62	26.68	26.83	26.63	29.44
4	14.00	14.06	14.01	13.95	15.27
8	9.42	9.57	9.42	9.50	9.96
12	8.76	9.05	9.09	9.12	9.76
LU-HP	*Clean*	*Sampling*	*Select*	*Hybrid*	*Full*
1	248.02	278.16	253.27	280.81	251.80
2	121.24	125.90	125.75	125.57	127.26
4	64.22	68.47	69.24	68.70	69.64
8	37.81	42.43	43.52	44.05	45.11
12	29.92	34.30	36.26	36.16	37.61

TABLE I
NAS PARALLEL BENCHMARKS OVERHEADS.

NPB BT Experiments

- Naïve full instrumentation has high overhead
- Want to see hybrid data with respect to X,Y,Z phases

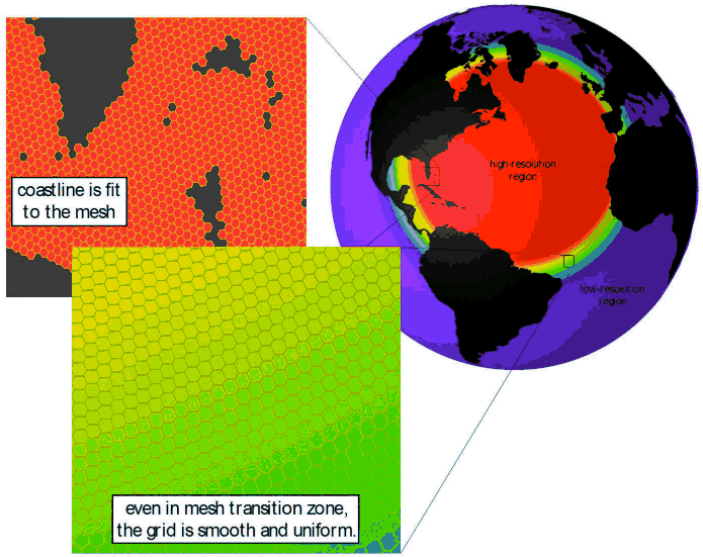


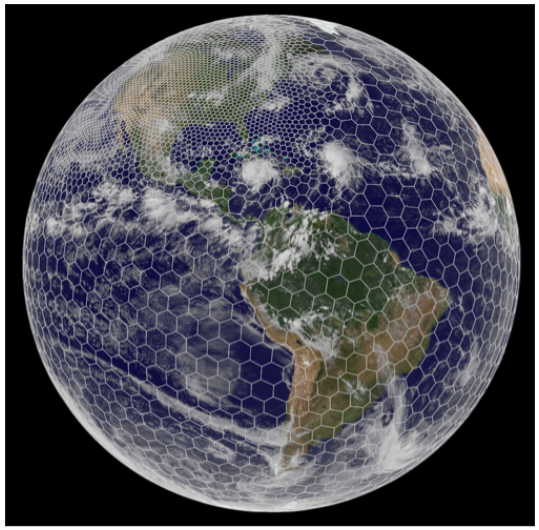
Name	Exclusive TIME	Inclusive TIME ▽
BT [{bt.f} {49,8}-{215,10}]	0.002	31.705
ADI [{adi.f} {4,7}-{20,9}]	0.005	31.535
COMPUTE_RHS [{rhs.f} {4,7}-{426,9}]	0.007	9.171
Z_SOLVE [{z_solve.f} {4,7}-{410,9}]	0.012	7.664
Y_SOLVE [{y_solve.f} {4,7}-{397,9}]	0.011	7.225
X_SOLVE [{x_solve.f} {5,7}-{399,9}]	0.01	6.678
OpenMP_PARALLEL_REGION: [OPENMP] x_solve_ [{x_solve.inst.f} {54}]	6.426	6.669
[CONTEXT] OpenMP_PARALLEL_REGION: [OPENMP] x_solve_ [{x_solve.inst.f} {54}]	0	6.355
[UNWIND] x_solve_ [{x_solve.inst.f} {54}]	0	6.335
[SUMMARY] x_solve_._omp_fn.0 [{x_solve.inst.f}]	2.488	2.488
[UNWIND] x_solve_._omp_fn.0 [{x_solve.inst.f} {362}]	0	2.039
[SUMMARY] binvcrhs_ [{x_solve.inst.f}]	2.039	2.039
[UNWIND] x_solve_._omp_fn.0 [{x_solve.inst.f} {353}]	0	1.469
[SUMMARY] matmul_sub_ [{x_solve.inst.f}]	1.469	1.469
[UNWIND] x_solve_._omp_fn.0 [{x_solve.inst.f} {346}]	0	0.28
[SAMPLE] matvec_sub_ [{x_solve.inst.f} {439}]	0.27	0.27
[SAMPLE] matvec_sub_ [{x_solve.inst.f} {415}]	0.01	0.01
[UNWIND] x_solve_._omp_fn.0 [{x_solve.inst.f} {143}]	0	0.03
[SAMPLE] lhsinit_ [{initialize.inst.f} {253}]	0.02	0.02
[SAMPLE] lhsinit_ [{initialize.inst.f} {266}]	0.01	0.01
[UNWIND] x_solve_._omp_fn.0 [{x_solve.inst.f} {377}]	0	0.02
[SAMPLE] matmul_sub_ [{x_solve.inst.f} {490}]	0.01	0.01

MPAS-Ocean Engagement

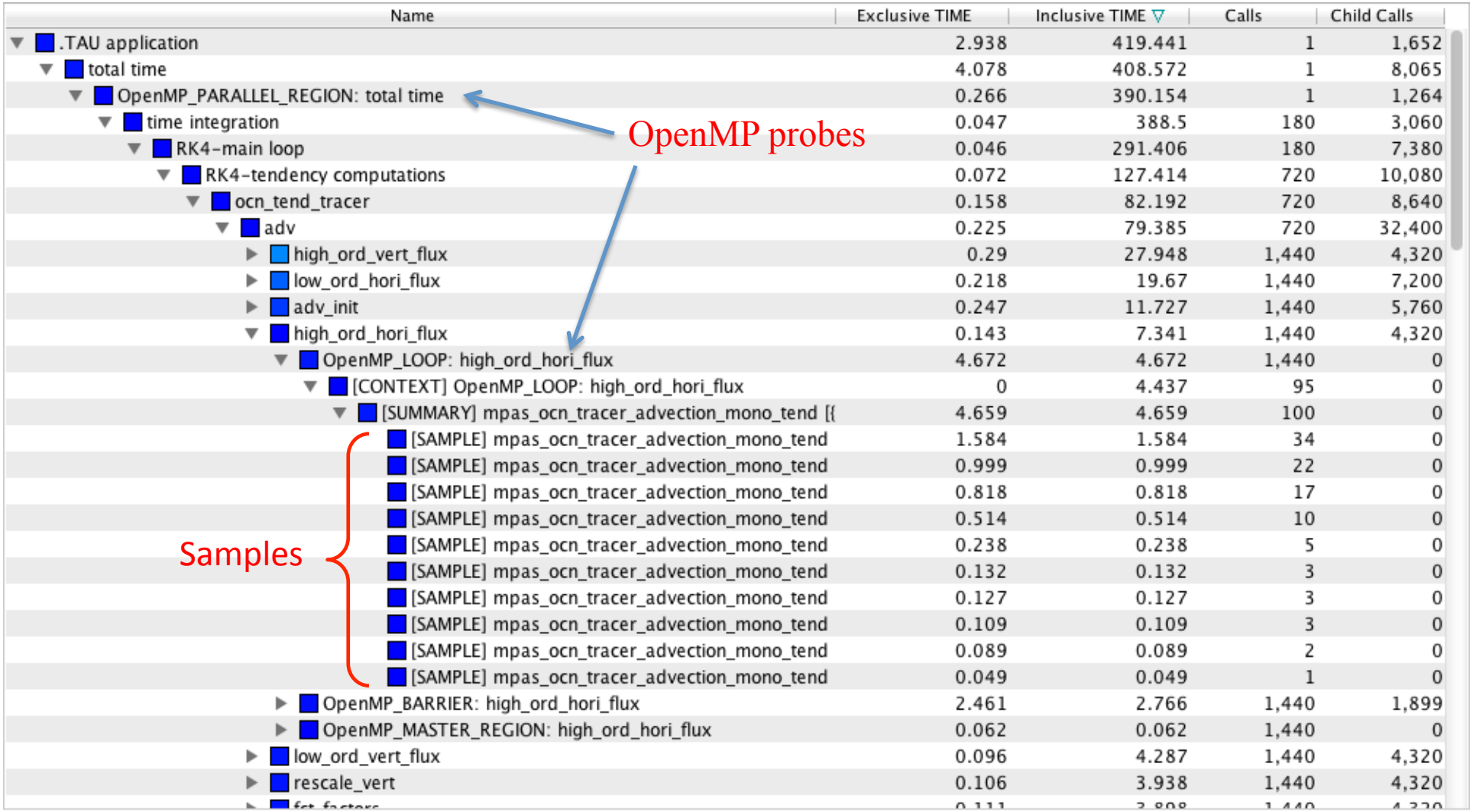
MPAS
Model for Prediction Across Scales

- Use multiscale methods for accurate, efficient, and scale-aware models of the earth system
- MPAS-O uses a variable resolution irregular mesh of hexagonal grid cells
- Cells assigned to MPI processes as "blocks"
  - Each cell has 1-40 vertical layers, depending on ocean depth
- MPAS-O has demonstrated scaling limits when using MPI alone
- Look at increasing concurrency
  - OpenMP
  - Vectorization
- Use hybrid profiling to better understand performance behavior
- Experiments performed on Hopper
  - Cray XE6





MPAS Hybrid Profile



Name	Exclusive TIME	Inclusive TIME ▽	Calls	Child Calls
.TAU application	2.938	419.441	1	1,652
total time	4.078	408.572	1	8,065
OpenMP_PARALLEL_REGION: total time	0.266	390.154	1	1,264
time integration	0.047	388.5	180	3,060
RK4-main loop	0.046	291.406	180	7,380
RK4-tendency computations	0.072	127.414	720	10,080
ocn_tend_tracer	0.158	82.192	720	8,640
adv	0.225	79.385	720	32,400
high_ord_vert_flux	0.29	27.948	1,440	4,320
low_ord_hori_flux	0.218	19.67	1,440	7,200
adv_init	0.247	11.727	1,440	5,760
high_ord_hori_flux	0.143	7.341	1,440	4,320
OpenMP_LOOP: high_ord_hori_flux	4.672	4.672	1,440	0
[CONTEXT] OpenMP_LOOP: high_ord_hori_flux	0	4.437	95	0
[SUMMARY] mpas_ocn_tracer_advection_mono_tend [{	4.659	4.659	100	0
[SAMPLE] mpas_ocn_tracer_advection_mono_tend	1.584	1.584	34	0
[SAMPLE] mpas_ocn_tracer_advection_mono_tend	0.999	0.999	22	0
[SAMPLE] mpas_ocn_tracer_advection_mono_tend	0.818	0.818	17	0
[SAMPLE] mpas_ocn_tracer_advection_mono_tend	0.514	0.514	10	0
[SAMPLE] mpas_ocn_tracer_advection_mono_tend	0.238	0.238	5	0
[SAMPLE] mpas_ocn_tracer_advection_mono_tend	0.132	0.132	3	0
[SAMPLE] mpas_ocn_tracer_advection_mono_tend	0.127	0.127	3	0
[SAMPLE] mpas_ocn_tracer_advection_mono_tend	0.109	0.109	3	0
[SAMPLE] mpas_ocn_tracer_advection_mono_tend	0.089	0.089	2	0
[SAMPLE] mpas_ocn_tracer_advection_mono_tend	0.049	0.049	1	0
OpenMP_BARRIER: high_ord_hori_flux	2.461	2.766	1,440	1,899
OpenMP_MASTER_REGION: high_ord_hori_flux	0.062	0.062	1,440	0
low_ord_vert_flux	0.096	4.287	1,440	4,320
rescale_vert	0.106	3.938	1,440	4,320

Hybrid Profiling to Measure Vectorization

- Solver vectorization gave ~20s reduction (mean)
- Worst case solver performance went from 170s to 44s
- Subsequent ~52s reduction in MPI_Wait time! Why?

	Before		After	
Metric	Mean	Max	Mean	Max
Time	39.165s	170.232s	19.807	44.620s
TOT_INS	2.64E10	1.01E11	1.77E10	4.58E10
TOT_L1_DCM	4.57E8	2.74E9	6.12E7	1.75.E8
FP_INS	i5.89E9	2.29E10	2.43E9	6.03E9

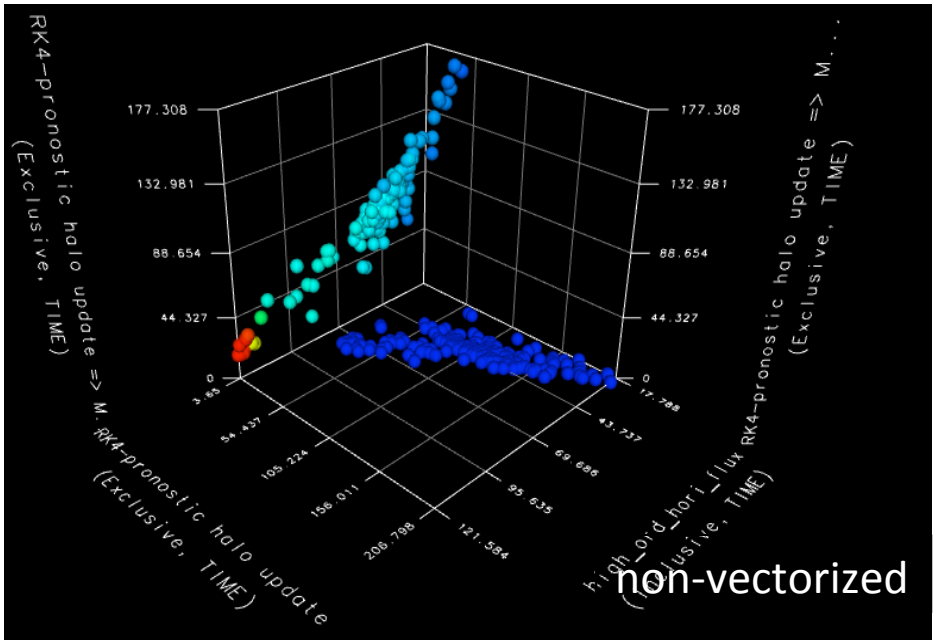
TABLE II

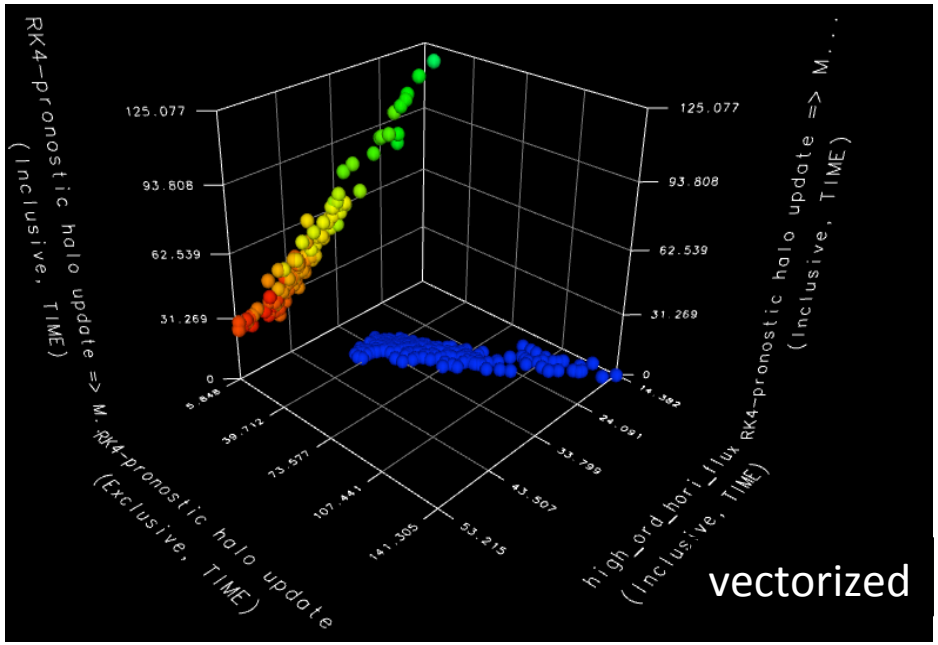
BEFORE AND AFTER VECTORIZATION OF

MPAS_OCN_TRACER_ADVECTION_MONO_TEND.

Vectorization Analysis with Hybrid Data

- Computation and synchronization are correlated
- Load imbalances seen in larger MPI wait (unvectorized)
- Vectorization tightens the imbalances
  - Depth + color show vectorization effect (faster, more uniform)
  - Smaller communication waiting (height and width)





Next Steps in Hybrid Parallel Profiling

- Hybrid parallel profiling provides the user with various options to control measurement
  - Probing: select events, enable event paths and depth
  - Sampling: interrupt period, call stack unwinding depth
  - Capture of timing information and hardware counters
- These are also dynamic hybrid controls
  - Unwinding of call stack to last probed event
    - reduces time spent unwinding
  - Need to determine an event's parent quickly
  - Useful for event contextualization and callsite profiling
- Hybrid techniques can be used to implement "folding"
  - Collect samples with counters for a region to create a statistical model of performance

Cross-Platform OpenMP Performance Analysis

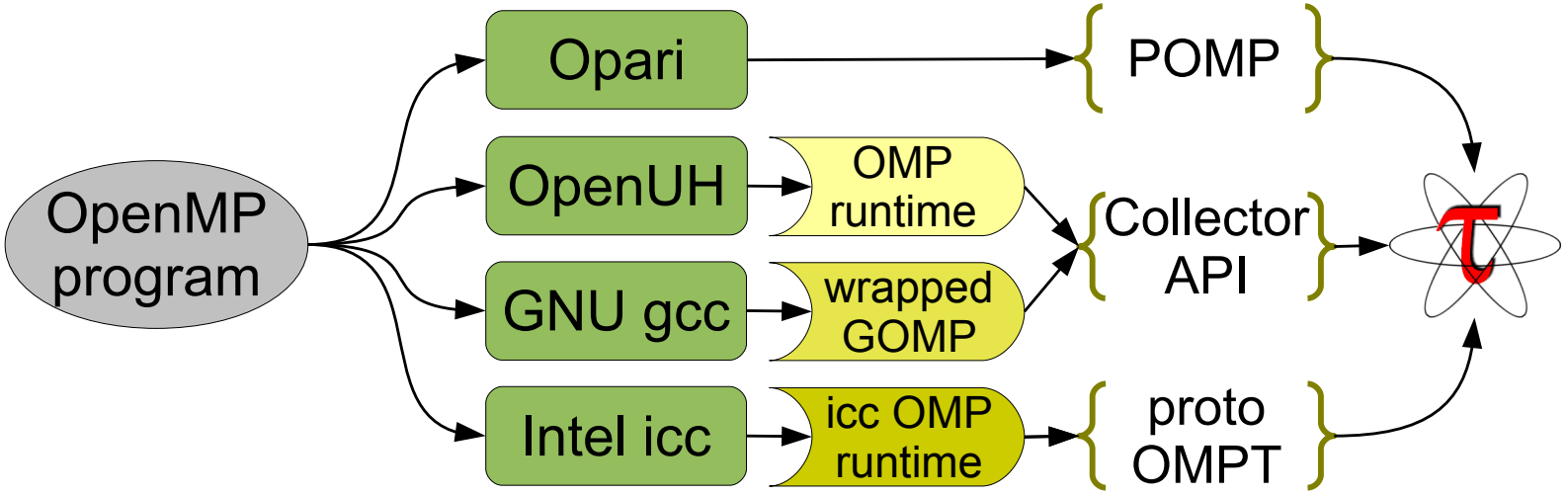
- Parallel language performance observability requires both visibility of performance data and the semantic context in which the performance occurs
- Portable OpenMP performance observability is problematic because of lack of a standard
  - Language semantics hide low-level parallel operation
  - Compilers hide complexity of runtime implementation
  - Different OpenMP compilers do things differently
- How can portable OpenMP performance measurement and analysis be provided?

OpenMP Performance API Efforts

- There has always been a strong interest in the OpenMP community about performance observability
- Several efforts have been investigated previously
  - POMP interface (Research Centre Juelich, U. Oregon)
  - OpenMP Runtime API (ORA) (a.k.a. Collector API) (Sun)
- Neither approach caught on as a standard
- OpenMP Tools Working Group
  - Focusing on defining an OpenMP API for tools (OMPT)
- Other interesting developments
  - GCC OpenMP runtime library (GOMP)
  - Intel OpenMP runtime system is now open source

Approach

- Even if have an OpenMP performance interface, still need to develop tools to use it
- Consider the present compiler landscape
- Implement support in TAU for different methods
  - Leverage TAU's portable measurement and analysis
- Utilize portable OpenMP measurement for cross-compiler and cross-platform OpenMP performance analysis



POMP / Opari

- Source code instrumentation approach
- POMP / POMP2 defines events that expose OpenMP operational semantics at the source level
  - Provides an interface for a measurement system
  - C/C++ and Fortran
- OpenMP Pragma and Region Instrumentation (Opari)
  - Source-to-source transformation utility
  - Passes semantic information through region descriptors
- Tools supporting POMP and Opari
  - TAU, Kojak, Scalasca
- POMP / Opari can only observe source-level events
  - Constrained from seeing anything about runtime
  - Has no insight on uninstrumented OpenMP libraries

POMP / Opari Example

```
#pragma omp parallel for
  for (i=nStart; i<=nEnd; ++i) {
    foo();
  }
```

```
int pomp2_num_threads = omp_get_max_threads();
int pomp2_if = 1;
POMP2_Task_handle pomp2_old_task;
POMP2_Parallel_fork(&pomp2_region_1, pomp2_if, pomp2_num_threads,
  &pomp2_old_task, pomp2_ctc_1);
#pragma omp parallel POMP2_DLIST_00 firstprivate(pomp2_old_task)
if (pomp2_if) num_threads(pomp2_num_threads)
{ POMP2_Parallel_begin(&pomp2_region_1 );
  { POMP2_For_enter(&pomp2_region_1, pomp2_ctc_1);
     #pragma omp for nowait
     for (i=nStart; i<=nEnd; ++i) { foo(); }
     { POMP2_Task_handle pomp2_old_task;
       POMP2_Implicit_barrier_enter(&pomp2_region_1, &pomp2_old_task );
       #pragma omp barrier
       POMP2_Implicit_barrier_exit( &pomp2_region_1, pomp2_old_task );
       POMP2_For_exit( &pomp2_region_1 ); }
     POMP2_Parallel_end( &pomp2_region_1 ); }
  POMP2_Parallel_join( &pomp2_region_1,pomp2_old_task );
}
```

OpenMP Runtime API (ORA, Collector API)

- First attempt at a standard OpenMP tools API
- External API consists of a single function

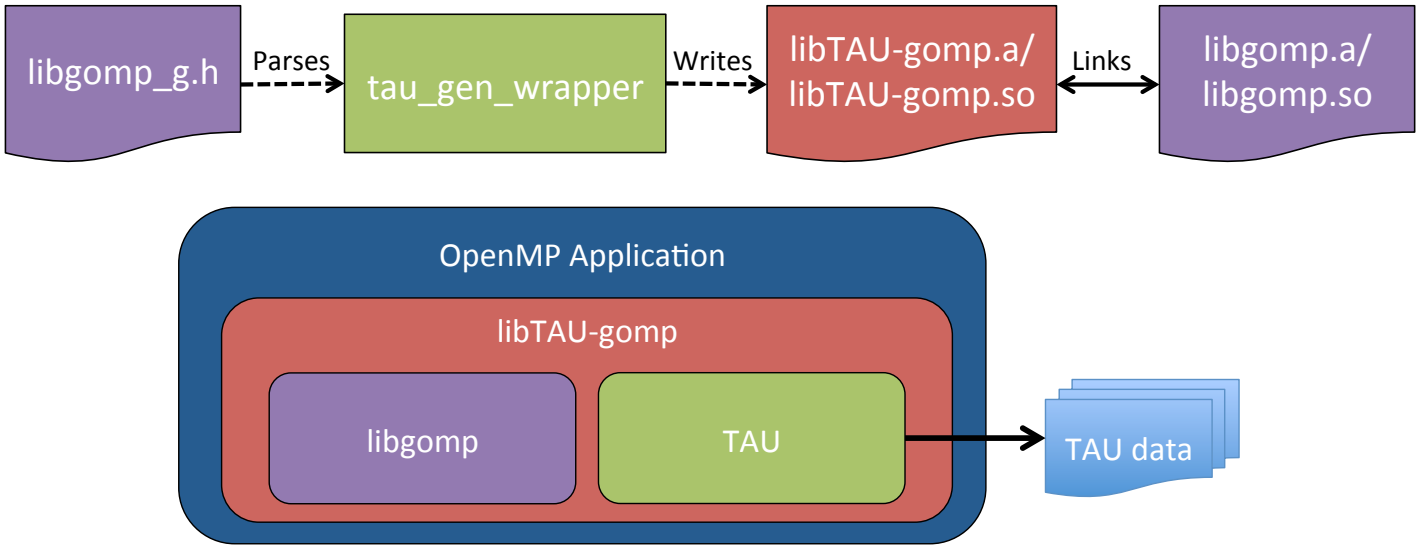
```
__omp_collector_api(void *)
```

  - Allows a tool to interact with the runtime system
  - Manages the *collector*
    - initialize, pause, resume, terminate
  - Registers a callback function for predefined events (22)
  - Query an existing thread's state (11 states)
- OpenMP ARB sanctioned the Collector API
  - Only supported in OpenUH and Solaris Studio compilers
  - Full support available in TAU with OpenUH and GOMP

Wrapping GCC OpenMP Runtime (GOMP)

- GCC OpenMP runtime is *libgomp* (GOMP)
  - 60 API calls to use to implement an OpenMP runtime
  - Use by GCC and other compilers
  - Use by source transformation tools (e.g., ROSE)
- Apply TAU's library wrapping technology to instrument for OpenMP performance measurement
  - `tau_gen_wrapper` wraps external libraries
  - Generates proxy functions implementing ORA
- TAU now supports OpenMP for GNU compilers!
  - Fully portable across all systems supporting GCC
- Wrapping still has limitations though
  - None of the GOMP internals are visible for measurement

Wrapping GCC OpenMP Runtime (GOMP)




```
typedef void (*GOMP_barrier_p) ();
void tau_GOMP_barrier(GOMP_barrier_p real_GOMP_barrier {
  OMP_COLLECTOR_API_THR_STATE previous;
  previous = set_state(THR_EBAR_STATE);
  event_callback(OMP_EVENT_THR_BEGIN_EBAR);
  (*real_GOMP_barrier)();
  event_callback(OMP_EVENT_THR_END_EBAR);
  set_state(previous);
}
```

OMPT

- New standard effort that is likely to be adopted
  - Complete draft specification (see OpenMP Forum)
- Design objectives
  - Low-overhead observation (application, runtime
  - Supports events and states (superset of ORA)
  - Stack frame for sampling
  - *Blame shifting* logic to resource synchronization
- IBM has support in experimental XL compilers
- Intel open sourced their OpenMP runtime and OMPT draft committed implemented for Intel compilers
  - TAU used as backend measurement system

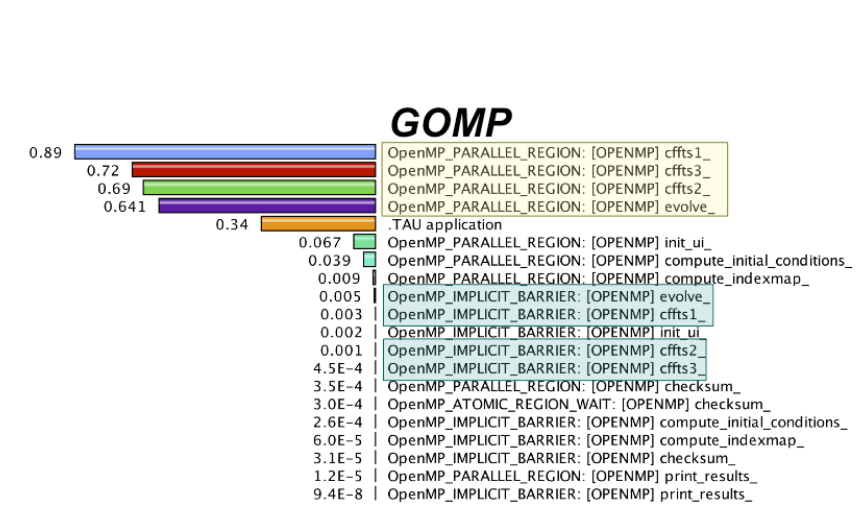
Comparison of Measurement Support

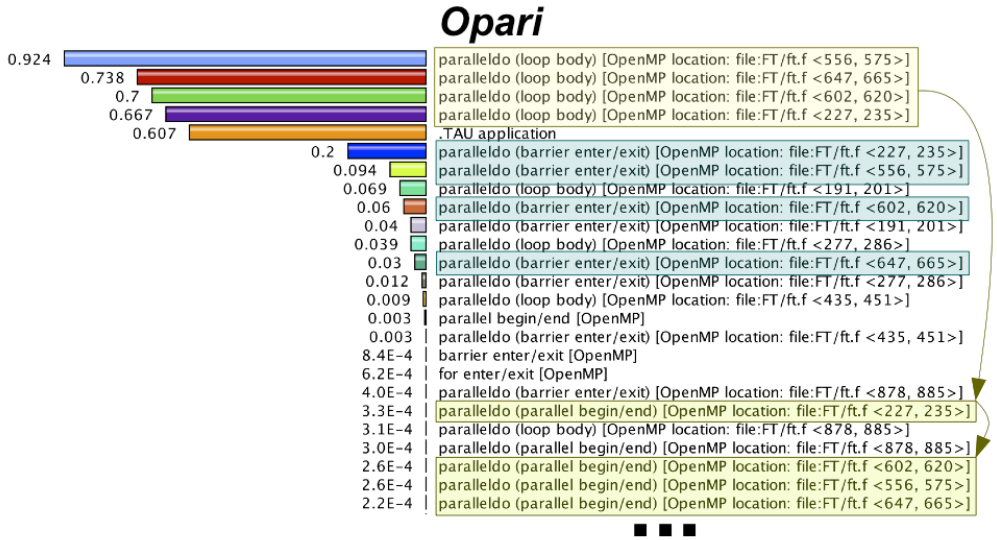
TABLE I

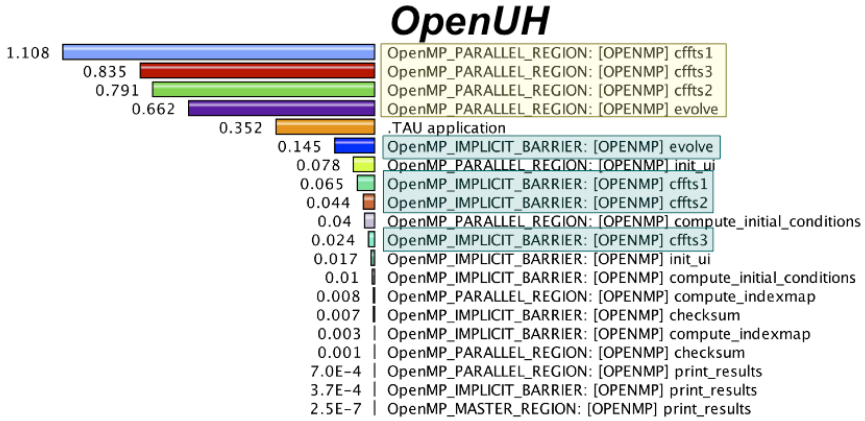
OPENMP PRAGMA / EVENT COVERAGE USING DIFFERENT MEASUREMENT TECHNIQUES.

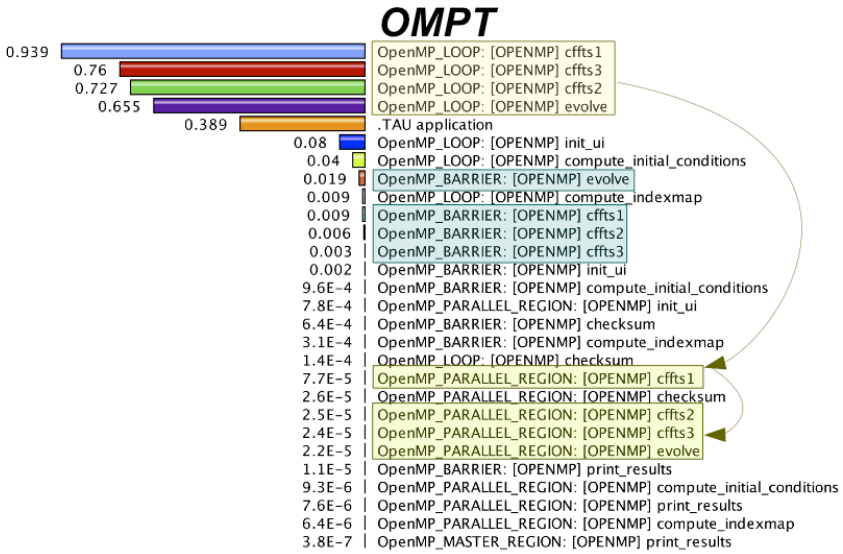
OpenMP Feature	Opari	ORA	OpenUH ORA	GOMP ORA	OMPT
Parallel Region	enter/exit, begin/end	Yes	Yes	Yes	Yes
Thread create/exit	pthread create, pthread exit	No	No	No	Yes
Work Sharing	enter/exit, begin/end	Yes	Yes	Yes	Yes
Atomics	enter/exit	waiting state	waiting state	waiting state	Yes
Barriers	explicit only	Yes	Yes	explicit, some implicit	Yes
Critical	enter/exit, begin/end	waiting state	waiting state	waiting state	Yes
Master	begin/end	Yes	Yes	Yes	Yes
Ordered	enter/exit, begin/end	Yes, wait state	wait state	wait state	Yes
Sections	enter/exit, begin/end	No	No	No	Yes
Single	begin/end	Yes	Yes	Yes	Yes
Locks	lock wrappers	waiting state	waiting state	waiting state	Yes
Task Creation	begin/end	Yes	Yes	Yes	Yes
Task Schedule	No	No	Yes	No	switch event
Task Wait	begin/end	Yes	Yes	Yes	Yes
Task Execution	begin/end	Yes	Yes	Yes	Yes
Task Completion	No	No	Yes	No	Yes
Task Yield	No	No	Yes	No	switch event

Flat Profiles for NPB FT.B Benchmark









Evaluation Experiments

- Objective
  - Evaluate performance coverage
  - Not intended to compare differences between OpenMP compilers or runtime systems
  - Show that cross-compiler and cross-platform OpenMP performance measurement is possible
- Benchmarks
  - NAS Parallel SPEC 2012 OpenMP Benchmarks
  - Barcelona OpenMP Task Suite (BOTS)
- Compilers
  - GNU GCC versions 4.4.7-4.8.0
  - OpenUH 3.0.26 (with Collector API task extensions)
  - Intel 13.1.2
  - Default runtime settings
- Platform
  - 4x Intel X7560, 2.27 GHz, 8-core CPUs, 384 GB memory
  - All experiments done with `OMP_NUM_THREADS = 32`

Overhead Measurements for 32 Threads

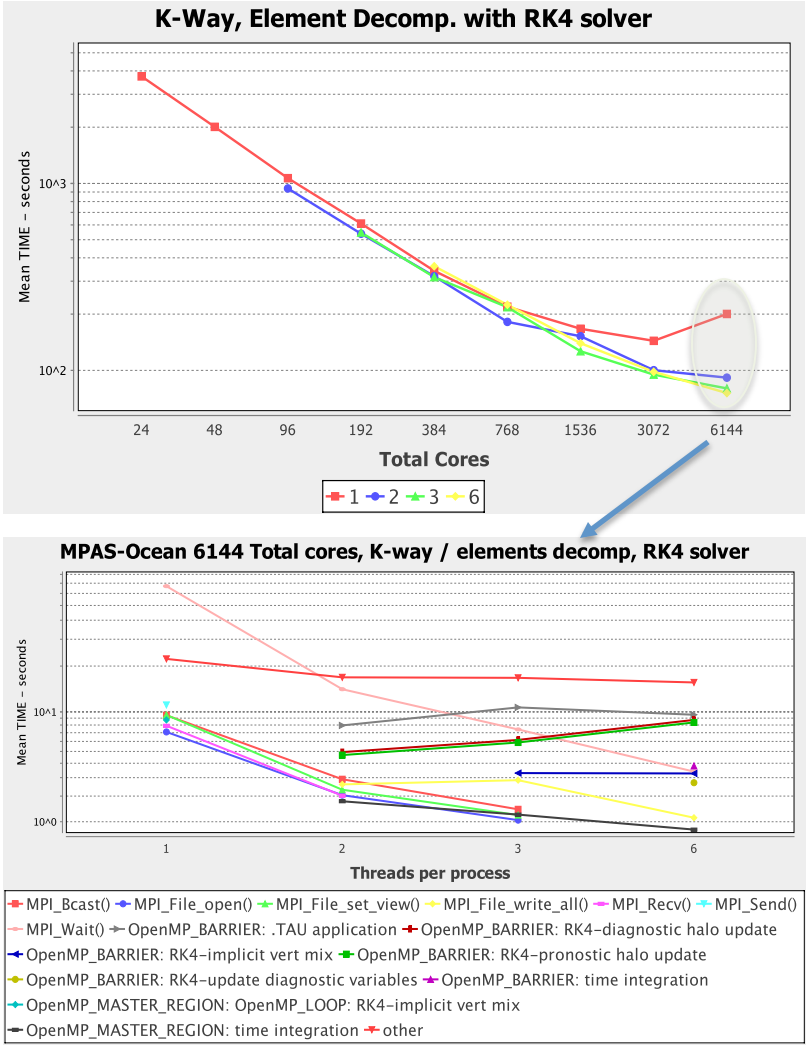
	GNU			INTEL			OPENUH		
NPB 3.2.1 Benchmark	Baseline	Opari	GOMP/ORA	Baseline	Opari	OMPT	Baseline	Opari	ORA
BT.B	17.92	19.96	22.42	16.18	16.56	16.80	18.58	18.85	19.37
CG.B	7.49	7.68	7.51	7.31	7.96	7.63	8.02	8.93	9.39
EP.B	2.92	2.99	2.96	1.40	1.42	1.40	1.47	1.51	1.51
FT.B	3.17	3.13	3.13	3.21	3.27	3.29	3.60	3.57	3.65
IS.B	0.75	0.74	0.74	0.75	0.75	0.75	0.81	0.83	0.82
LU.B	12.37	**17.43**	12.38	12.25	**26.71**	13.22	13.31	**26.37**	14.32
LU-HP.B	22.00	**42.86**	**31.26**	19.24	**51.74**	**30.93**	23.13	**86.17**	**39.04**
MG.B	1.21	1.61	1.30	1.08	1.27	1.13	1.16	1.51	1.31
SP.B	16.02	16.10	16.53	14.55	15.53	15.59	16.61	18.07	19.62
BOTS 1.1.2 Benchmark	Baseline	Opari	GOMP/ORA	Baseline	Opari	OMPT	Baseline	Opari	ORA
alignment.single-omp-tasks 1000	70.59	73.99	70.77	48.85	48.90	48.86	56.19	56.44	56.20
fft.omp-tasks 134217728	132.55	124.93	127.94	3.57	5.52	4.61	11.69	13.05	13.78
fib.omp-tasks 30	23.91	27.11	28.89	0.10	1.71	1.33	0.97	4.16	3.18
floorplan.omp-tasks 15	226.58	225.71	234.35	0.70	**11.44**	**6.06**	18.95	**24.09**	21.38
health.omp-tasks small	38.78	37.04	29.47	0.46	1.78	0.98	2.41	2.70	2.72
nqueens.omp-tasks 12	120.18	134.28	156.44	0.13	**5.14**	**3.87**	10.17	11.92	11.18
sort.omp-tasks 134217728	20.76	26.83	22.73	2.44	2.49	2.55	2.90	3.15	3.17
sparselu.single-omp-tasks 100x100	4.11	4.19	4.10	11.49	11.52	11.60	3.82	3.85	3.83
strassen.omp-tasks 8192	0.02	0.02	0.03	0.03	0.03	0.04	0.01	0.04	0.03
uts.omp-tasks small	53.54	**212.57**	**218.78**	11.00	**64.84**	**34.26**	*segv*	n/a	n/a
SPEC 2012 Benchmark	Baseline	Opari	GOMP/ORA	Baseline	Opari	OMPT	Baseline	Opari	ORA
351.bwaves	23.99	24.38	24.18	*segv*	n/a	n/a	*segv*	n/a	n/a
352.nab	24.00	25.85	29.41	20.40	22.96	21.40	35.13	35.80	35.68
357.bt331	20.69	build error	21.44	18.27	22.49	18.75	21.41	build error	build error
358.botsalgn	1.07	1.21	3.65	1.07	1.25	1.07	1.05	1.08	1.15
359.botsspar	2.67	2.97	2.87	2.78	2.31	1.88	2.87	1.12	2.89
360.ilbdc (test)	**502.76**	**282.18**	**313.58**	13.33	13.47	13.48	14.16	14.36	15.65
362.fma3d	14.31	15.05	47.73	13.15	13.58	13.16	19.88	24.74	24.57
363.swim	10.18	11.13	10.21	10.26	10.97	10.29	22.36	build error	23.00
367.imagick	19.40	19.44	19.52	5.84	5.86	6.04	145.97	145.85	146.06
370.mgrid331	0.72	1.37	0.86	0.65	1.85	1.10	0.75	2.71	2.57
371.applu331	3.82	12.26	6.16	3.45	13.07	4.29	*segv*	n/a	n/a
372.smithwa	1.89	2.19	2.18	1.45	1.45	1.50	2.58	2.62	2.61

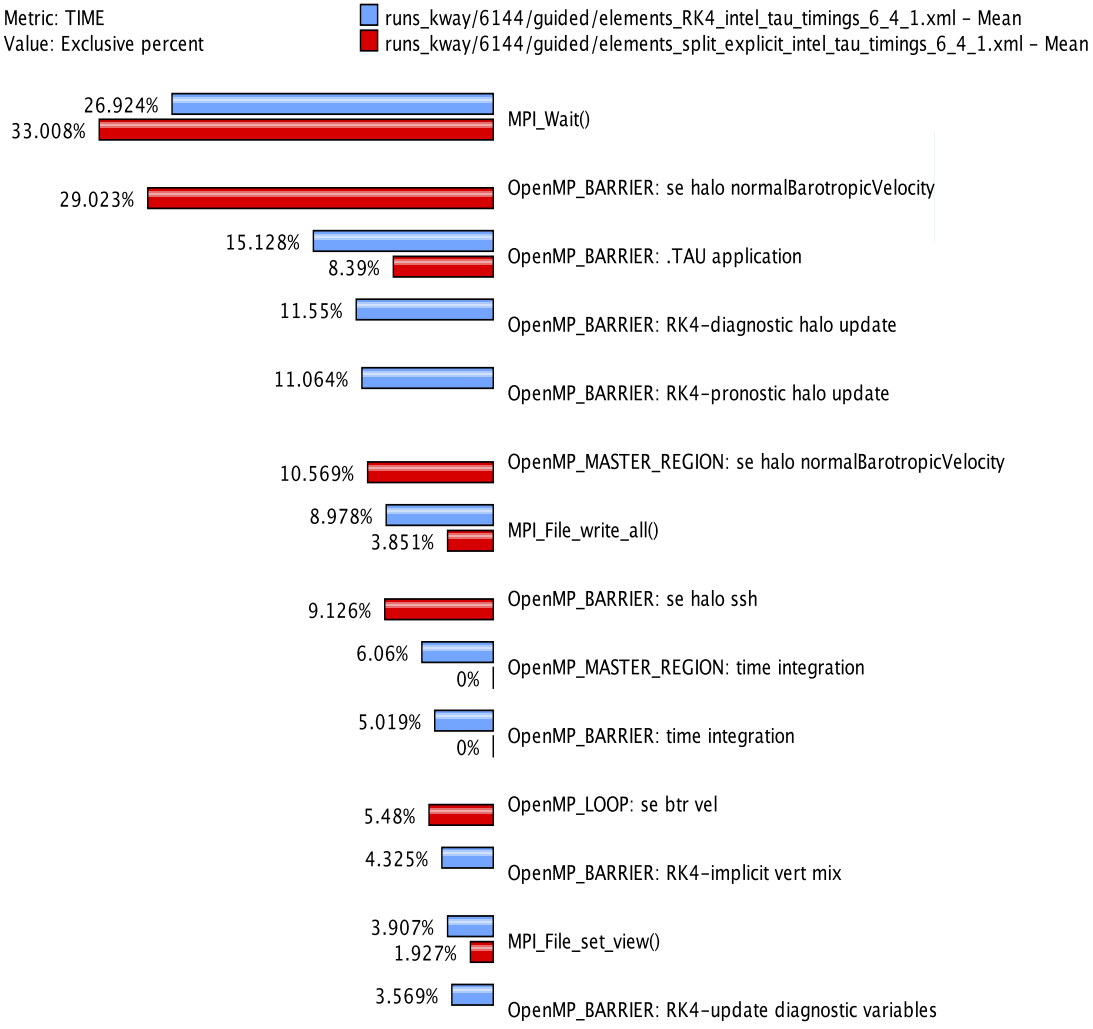
Mean Exclusive (floorplan.omp_tasks(15))

MPAS-Ocean Case Study

- Demonstrate portable measurement of OpenMP application across compilers and platforms
- Utilize different features to investigate performance characteristics
- Scaling study with respect to design parameters
  - Decomposition
  - Solver implementation
  - OpenMP runtime scheduler
- Platform
  - Hopper, Cray XE6

MPAS-Ocean Results





Next Steps for OpenMP Performance Tools

- Important to provide portable OpenMP performance tools through integrated measurement
- OMPT is likely to become an adopted standard
- Work with OpenMP Tools Working Group
  - Implement OMPT specification with TAU as the measurement framework
- Explore other OpenMP instrumentation avenues
  - Binary instrumentation can be important
  - Working with Dyninst and MAQAO
- Cross-platform application performance evaluation
  - MPAS-O port to IBM BG/Q will require OMPT support
  - GOMP can be used on MPAS-O port to Cray Titan

Autotuning is a Performance Engineering Process

- Autotuning methodology incorporates aspects common to "traditional" application performance engineering
  - Empirical performance observation
  - Experiment-oriented
- Autotuning embodies progressive engineering techniques
  - Automated experimentation and performance testing
  - Guided optimization by (intelligent) search space exploration
  - Model-based (domain-specific) computational semantics
- However, autotuning is based on optimization and search, not performance diagnosis
- There are shared objectives for performance technology and opportunities for tool integration

TAU Integration with Empirical Autotuning

- Autotuning components (DOE SUPER project)

  - Active Harmony autotuning system (Hollingsworth, UMD)
    - software architecture for optimization and adaptation
  - CHiLL compiler framework (Hall, Utah)
    - CPU and GPU code transformations for optimization
  - Orio annotation-based autotuning (Norris, University of Oregon)
    - code transformation (C, Fortran, CUDA) with optimization
- Goal to integrate TAU with existing autotuning frameworks
  - Use TAU to gather performance data for autotuning/ specialization
  - Store performance data with metadata for each experiment variant and store in performance database (TAUdb)
  - Use TAU Python analysis scripts, data mining, and machine learning to increase automation of autotuning and specialization

CHiLL: Compiler-based Autotuning

- CHiLL is a compiler framework for loop transformations based on the polyhedral model
- Used to automatically generate specialized versions according to specified optimization strategies
  - CHiLL provides a high-level scripting interface
  - Allows a user or a compiler to specify a sequence of code transformations and parameters
  - Describes the search space of specialized implementations
  - Heuristics trim the search space
- Variants are generated, measured, and analyzed

CHiLL Transformation Scripts and Parameters

- Transformation scripts are parameterized
- Consider matrix multiply

```
for(j=0; j < n; j++) {
  for(k=0; k < n; k++) {
    for(i=0; i < n; i++) {
       c[i][j] =c[i][j] + a[i][k]*b[k][j];
    }
  }
}
```

(a) Naïve implementation of matrix multiply

```
permute([3,1,2])
tile(0,2,TJ)
tile(0,2,TI)
tile(0,5,TK)
datacopy(0,3,a,false,1)
datacopy(0,4,b)
unroll(0,4,UI)
unroll(0,5,UJ)
```

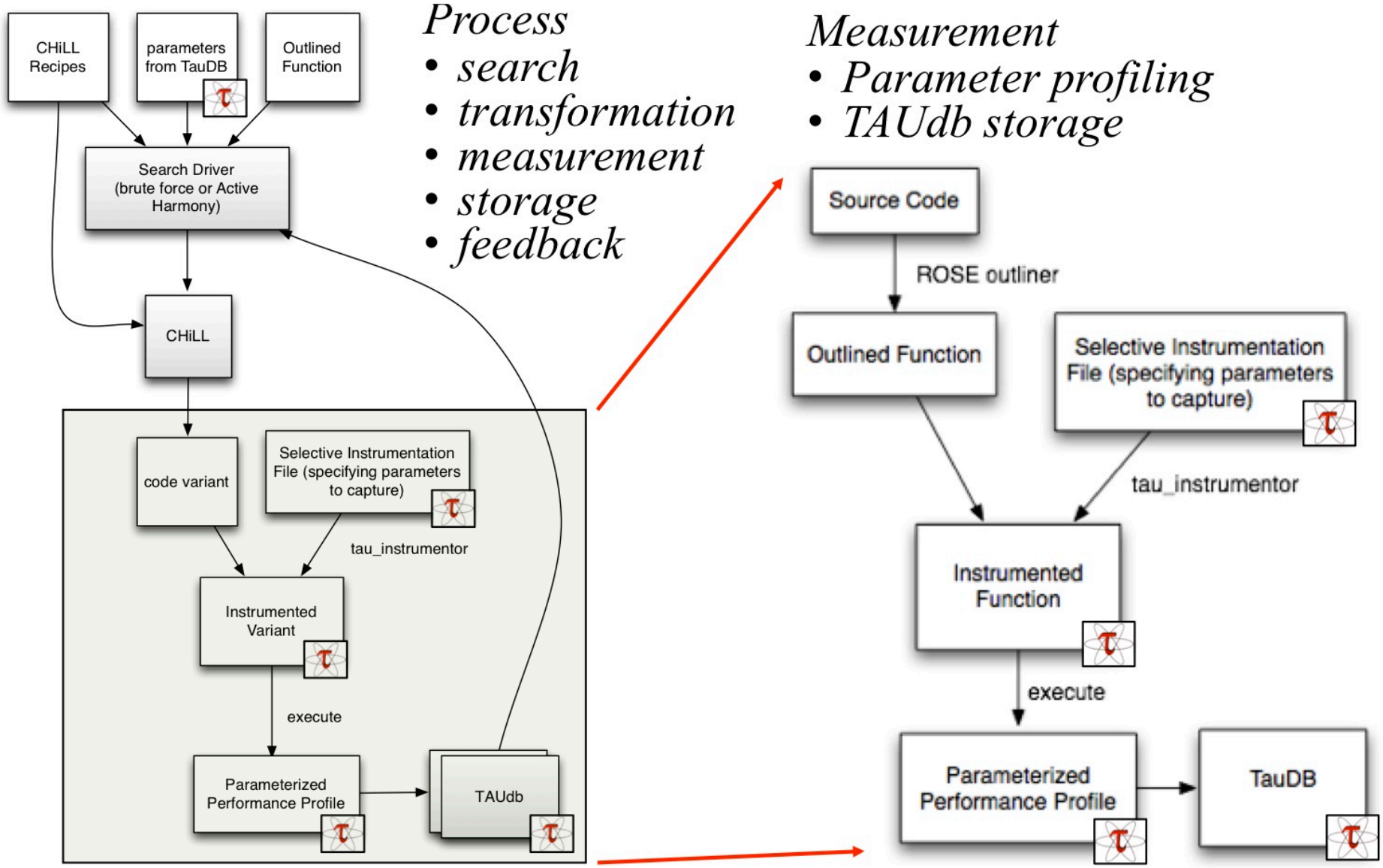
(b) CHiLL recipe for large matrices

```
permute([1,2,3])
unroll(1,1,U1)
unroll(1,2,U2)
unroll(1,3,U3)
```

(c) CHiLL recipe for small matrices

- TJ, TI, TK determine tile sizes
- UI, UJ determine unroll factors
- Overall search space is {parameters} x {scripts}
  - Each describes a transformation that produces a code variant

CHiLL + Active Harmony + TAU

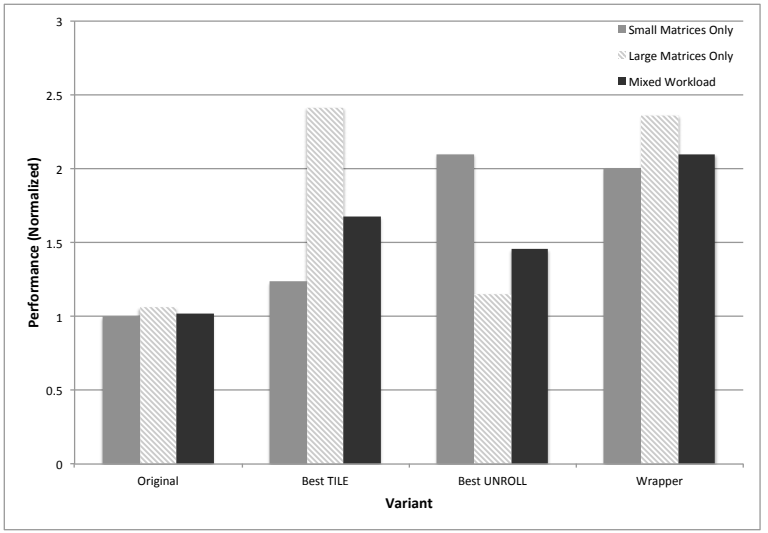


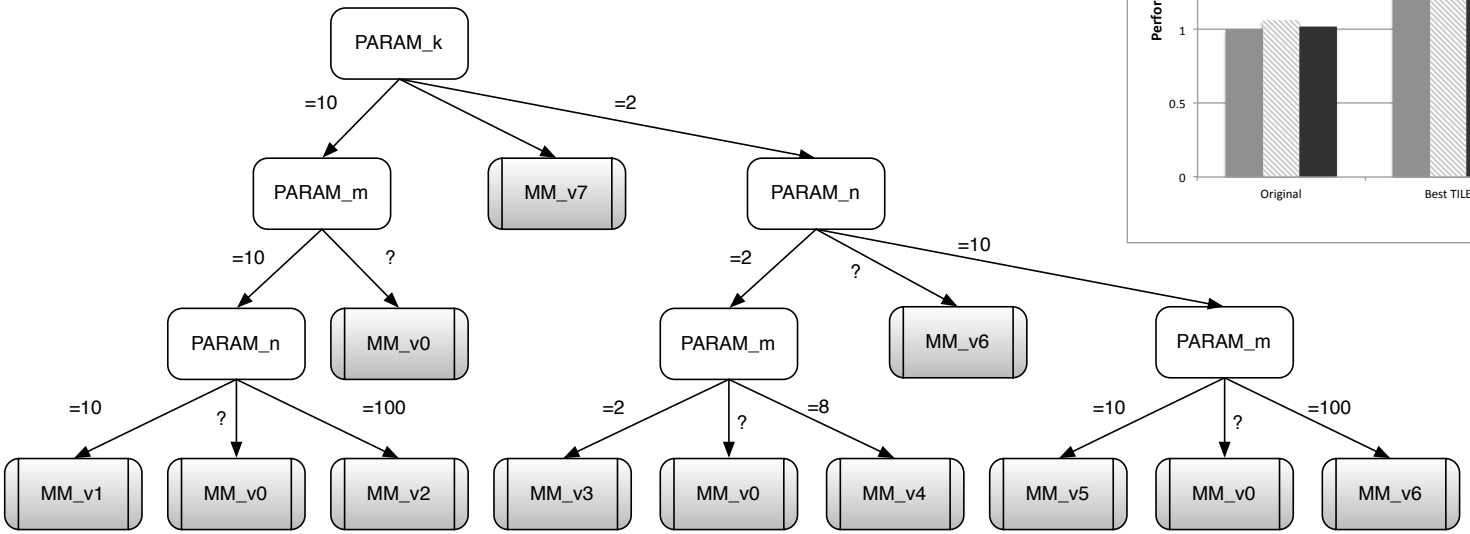
Autotuning with TAUdb Methodology

- Each time the program executes a code variant, we store metadata in the performance database indicating by what process the variant was produced:
  - Source function
  - Name of CHiLL recipe
  - Parameters to CHiLL recipe
- The database also contains metadata on what parameters were called and also on the execution environment:
  - OS name, version, release, native architecture
  - CPU vendor, ID, clock speed, cache sizes, # cores
  - Memory size
  - Any metadata specified by the end user

Specialization using Decision Tree Learning

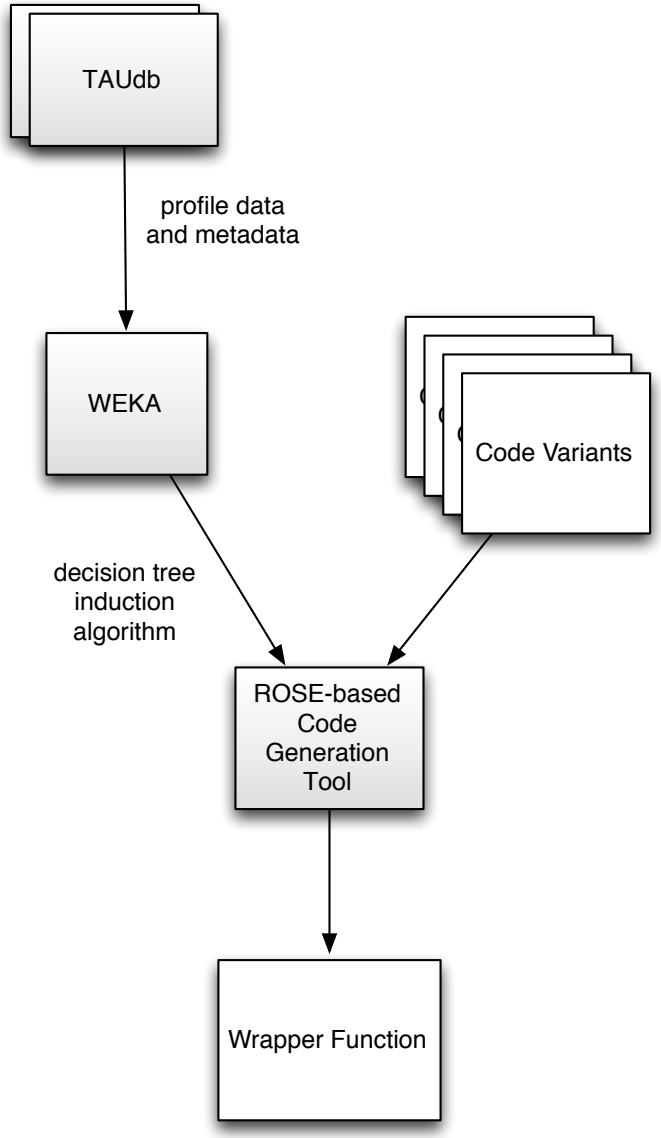
- Apply machine learning to data stored in TAUdb
  - Generate decision trees based upon code features
- Consider matrix multiply kernel
  - Matrices of different sizes with different performance
  - Automatically generate function to select specialized variants





Decision Code Generation for Specialization

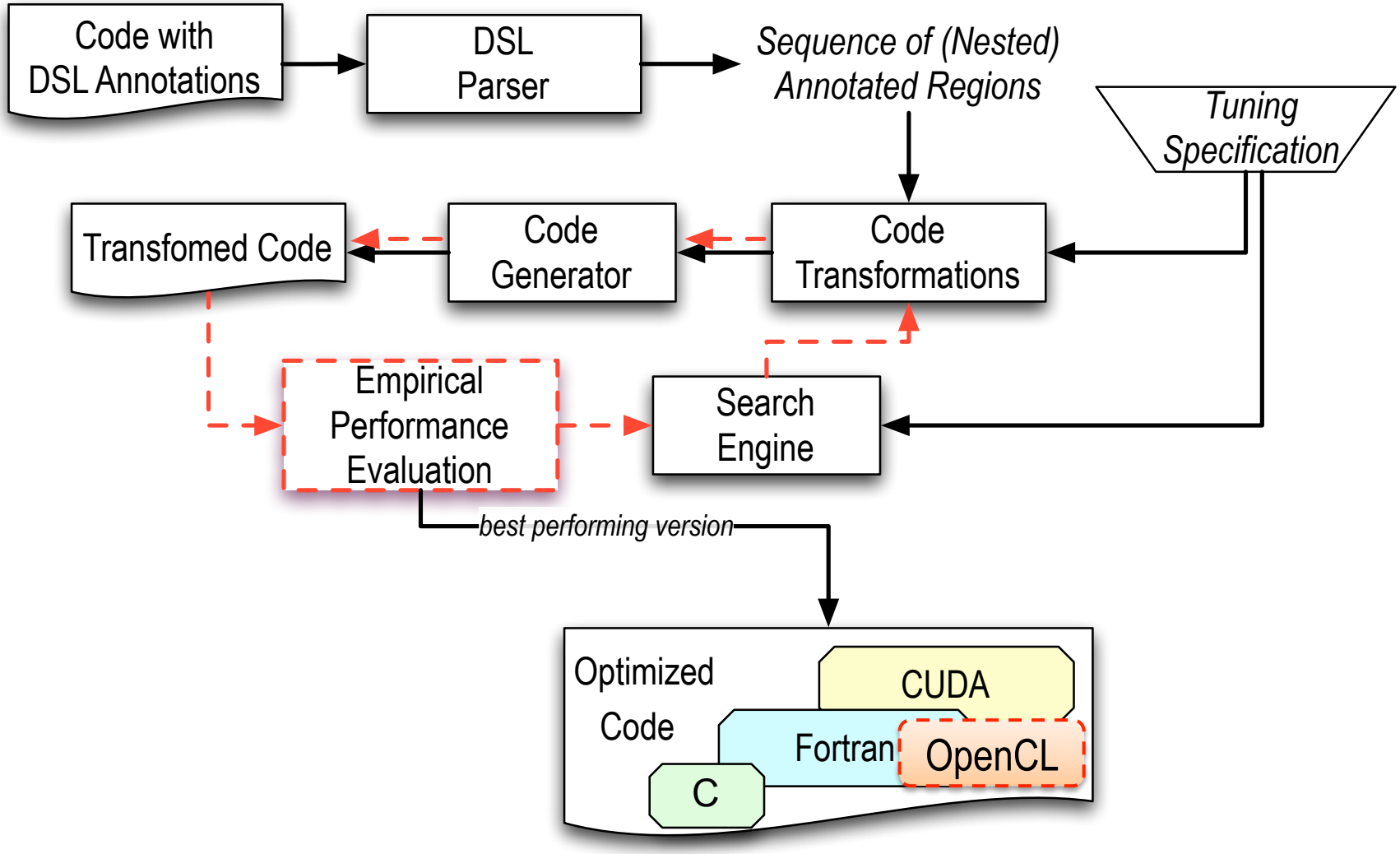
- Use a ROSE-based tool to generate a wrapper function
  - Carries out the decisions in the decision tree and executes the best code variant
- Decision tree code generation tool takes Weka-generated decision tree and a set of decision functions
  - If using custom metadata, user needs to provide a custom decision function
  - Provide decision functions for metadata to be automatically collected by TAU



Orio: DSL-based Transformation for Autotuning

- Express any properties of the computation that can possibly be exploited to optimize performance
  - Ones that a general purpose language (GPL) compiler cannot determine necessarily
    - dense representation of the sparse matrix from stencil
- Approach
  - Define new DSLs or embeddable restricted GPLs
  - Augmentable with optimization specifications
    - capture typical optimizations (e.g., tiling, unrolling, …)
    - specialized implementations (e.g., different input sizes)
  - Transform code based on knowledge in various forms
  - Evaluate performance of variants (different code output)
  - Search for best

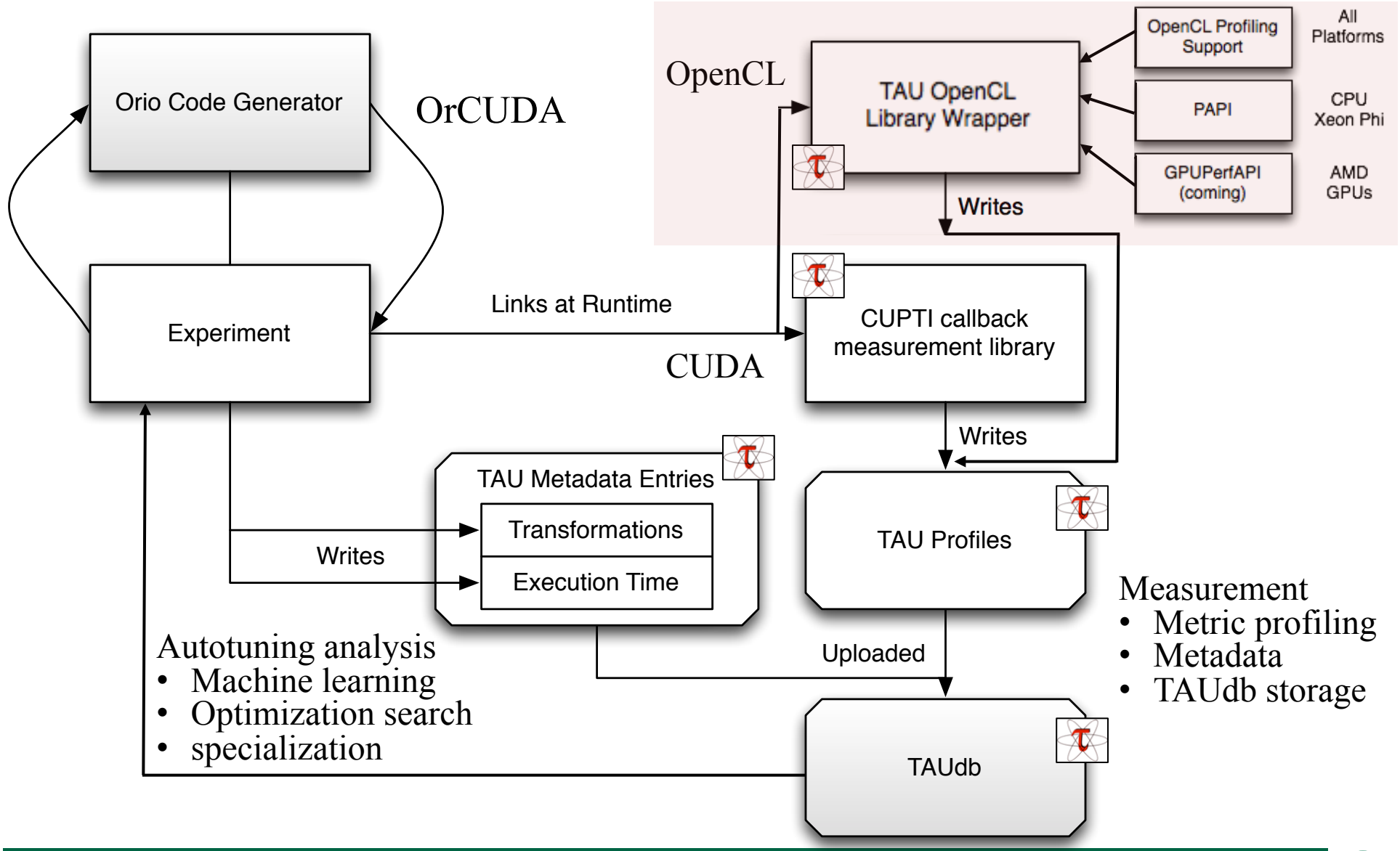
Orio Autotuning Framework



Orio and TAU Integration

- Source code annotations (pragmas) allow Orio to generate a set of low-level performance optimizations
  - Focus on code for accelerator cards
  - Run kernel after each optimization (transformation) applied
  - Set of optimizations is searched to find the best for a kernel
- Integrated TAU with Orio to collect performance data about each experiment that Orio runs
- Performance data collected by TAU are stored in TAUdb and are fed back to Orio
- Allows Orio to tune based on any TAU metric
- Performance data in TAUdb can be analyzed by TAU data mining routines

Orio and TAU Integration

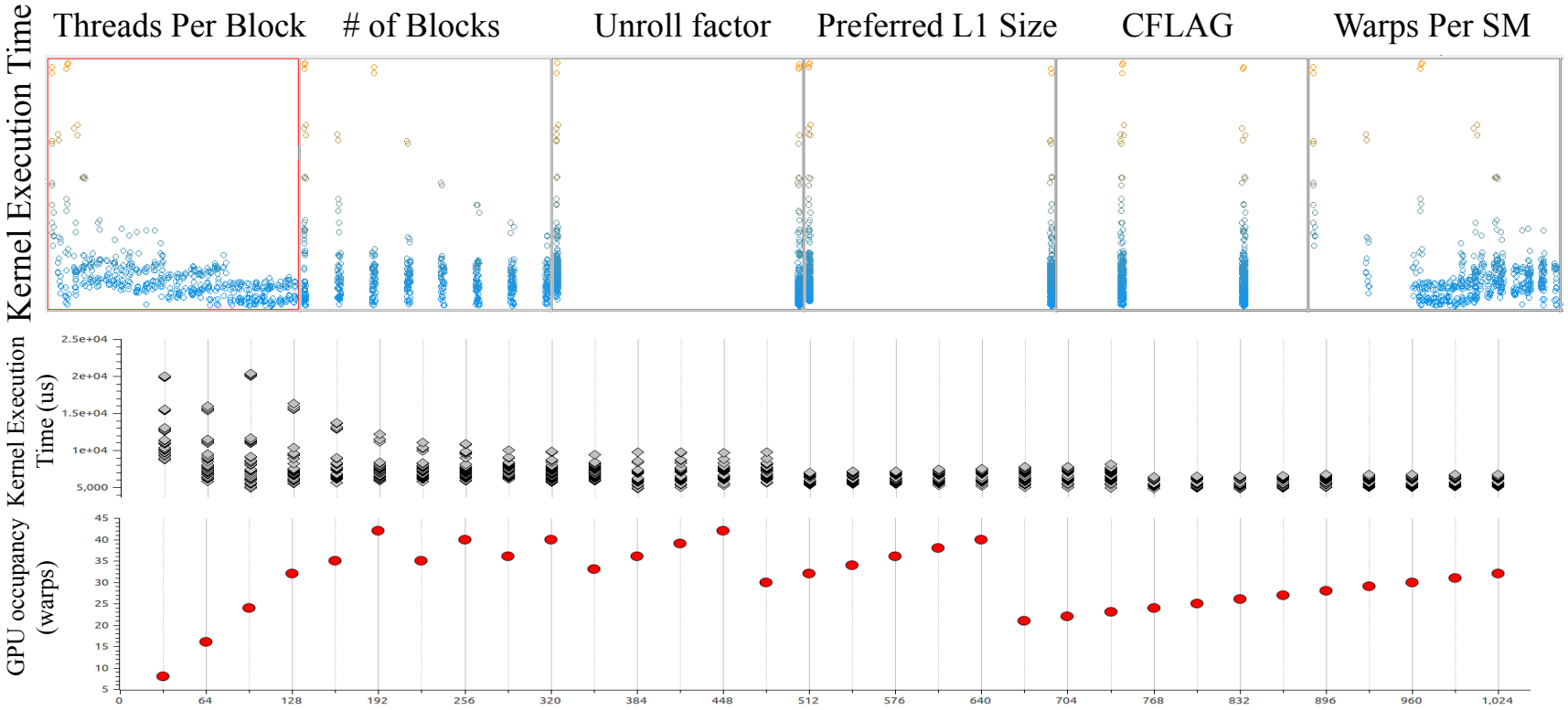


Orio Transformations

- Orio already supports transformation to for CUDA kernels
  - Search of a parameter space on thread count, block count, cache blocking, L1 size, loop unrolling
  - Use TAU's support for CUPTI to gather CUDA performance
- Recently we added support for OpenCL code generation
  - Analogous to the existing CUDA support
  - Allows generation of OpenCL kernels from the same original code as for CUDA
  - Allows different platforms to be targeted (CPUs, GPUs, Phi)
  - Use TAU's OpenCL library wrapper to gather performance
- TAU allows us to use the same measurement infrastructure to gather performance data for all these experiments
- Metadata describing the applied transformations and the execution environment (accelerator type, memory sizes, …) are stored with the profiles in TAUdb

Orio Tuning of Vector Multiplication

- Orio tuning of a simple 3D vector multiplication
- 2,048 experiments fed into TAUdb
  - TAU PerfExplorer used Weka for component analysis

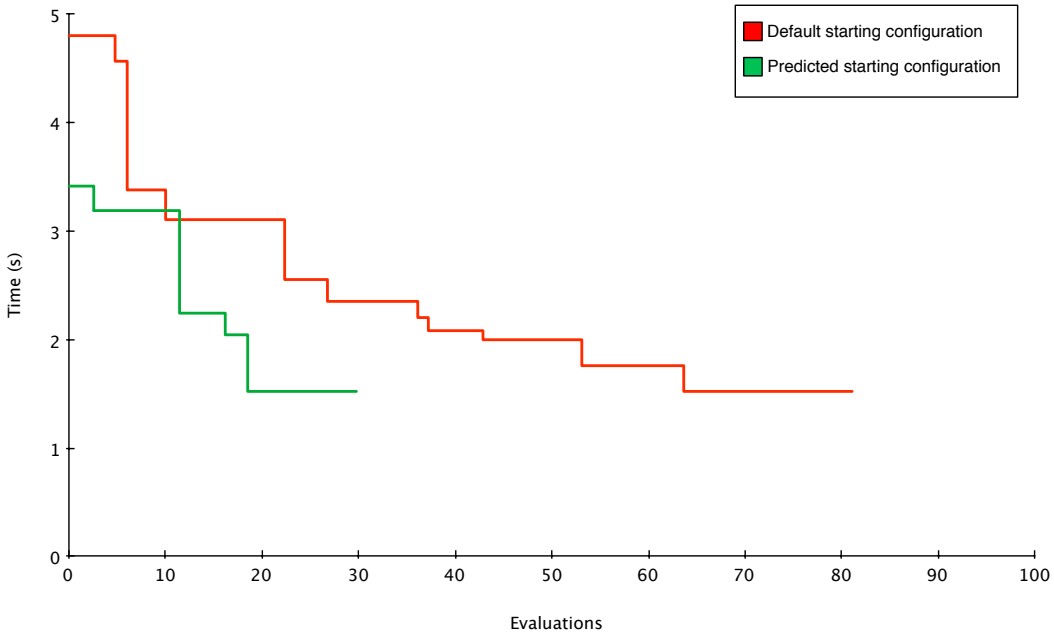


Initial Configuration Selection

- Speed autotuning search process by learning classifier to select an initial configuration.
- When starting out autotuning a new code:
  - Use default initial configuration
  - Capture performance data into TAUdb
- Once sufficient data is collected:
  - Generate classifier
- On subsequent autotuning runs:
  - Use classifier to propose an initial configuration for search
- Could also implement a guided search
  - With the Active Harmony plugin interface, we could provide input beyond the first step of the search

Initial Configuration Selection Example

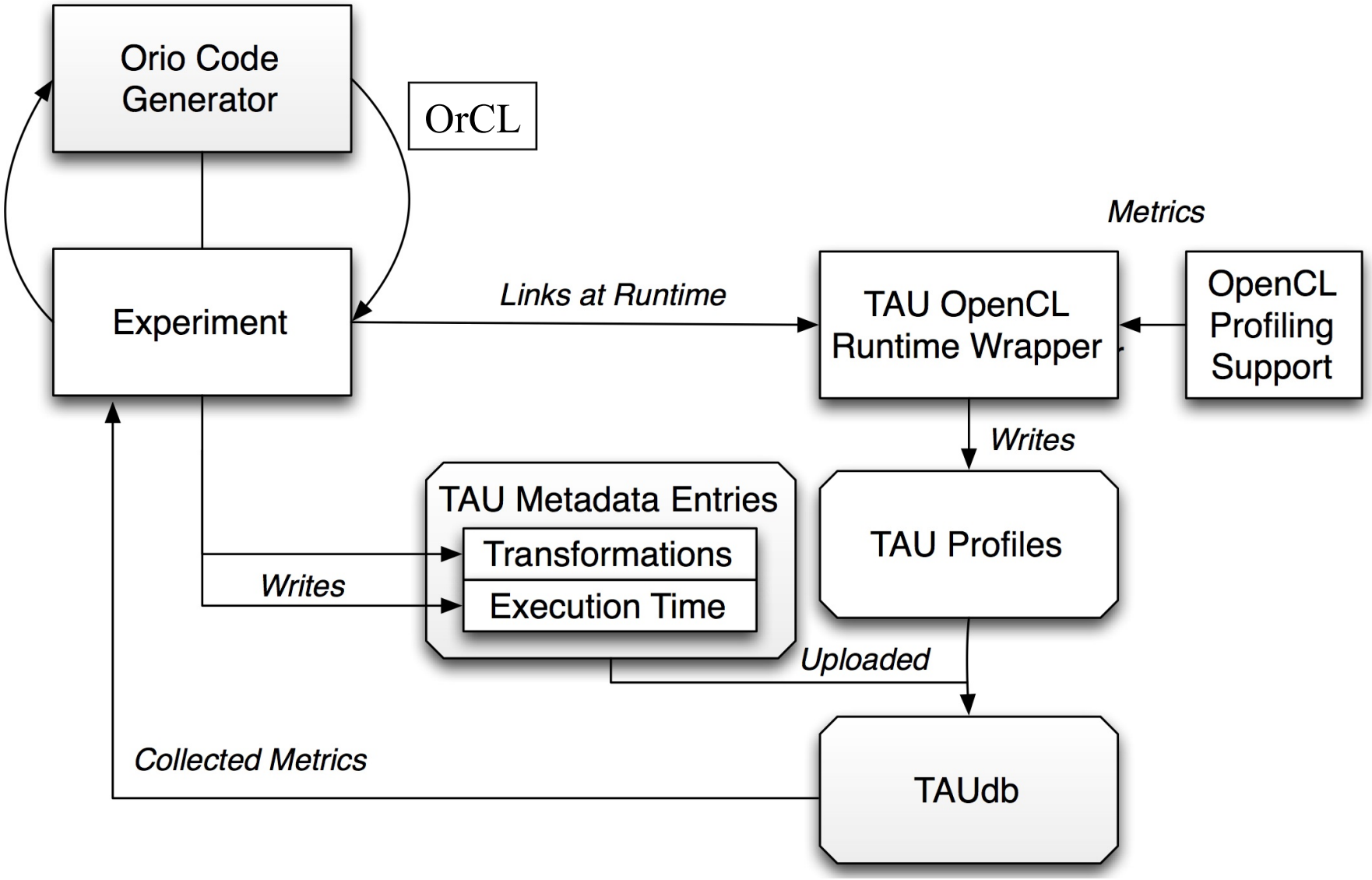
- Matrix multiplication kernel in C
- CUDA code generated using CUDA-CHiLL
- Tuned on several different NVIDIA GPUs
  - S1070, C2050, C2070, GTX480
- Learn on data from three GPUs, test on remaining one
- Results in reduction in evaluations required to converge



Multi-target Autotuning

- Extend Orio with support for OpenCL autotuning code generator (OrCL transformation module)
  - Enable autotuning across a broader set of architectures
    - CPU
    - GPU (NVIDIA, AMD)
    - Xeon Phi
  - Allows for comparison of different generated code
    - CUDA versus OpenCL
  - More portable (not necessarily performance portable)
- Leverage more TAU integration to utilize hardware counter data in the optimization process

TAU Integration in Orio for OpenCL



Orio OpenCL Tuning Specifications

- Sections
  - `performance_params`
    describing the parameter values which make up the search space for autotuning
  - `build`
    describes how generated variants can be compiled
  - `input_params`
    describes properties of the inputs against which generated variants are tested
  - `input_vars section`
    specifies the values of the inputs
  - `performance_counter`, `performance_test_code`
    describe how performance measurements of the generated variants are to be made

Orio OpenCL Transformation Statement

- Makes use of the parameter values described in the tuning specification
- *workGroups* – # of OpenCL work groups to use
  - workGroups * *workItemsPerGroup* gives the overall number of threads that make up a kernel invocation, the *global work size*
- *workItemsPerGroup* – # of work items (threads) that make up each work group
  - controls the *local work size*
  - each device has a maximum number of work items per group, which can be queried on the host
- *sizeHint* – OpenCL provides a pair of function attributes which provide hints to the compiler about the expected local work size (*work_group_size_hint*, *reqd_work_group_size)*

Orio OpenCL Transformation Statement

- *Other parameters:*
  - *vecHint*
  - *cacheBlocks*
  - *unrollInner*
  - *clFlags*
  - *device*
  - *Platform*
- OrCL is able to utilize all of these parameters in the code transformation decision analysis

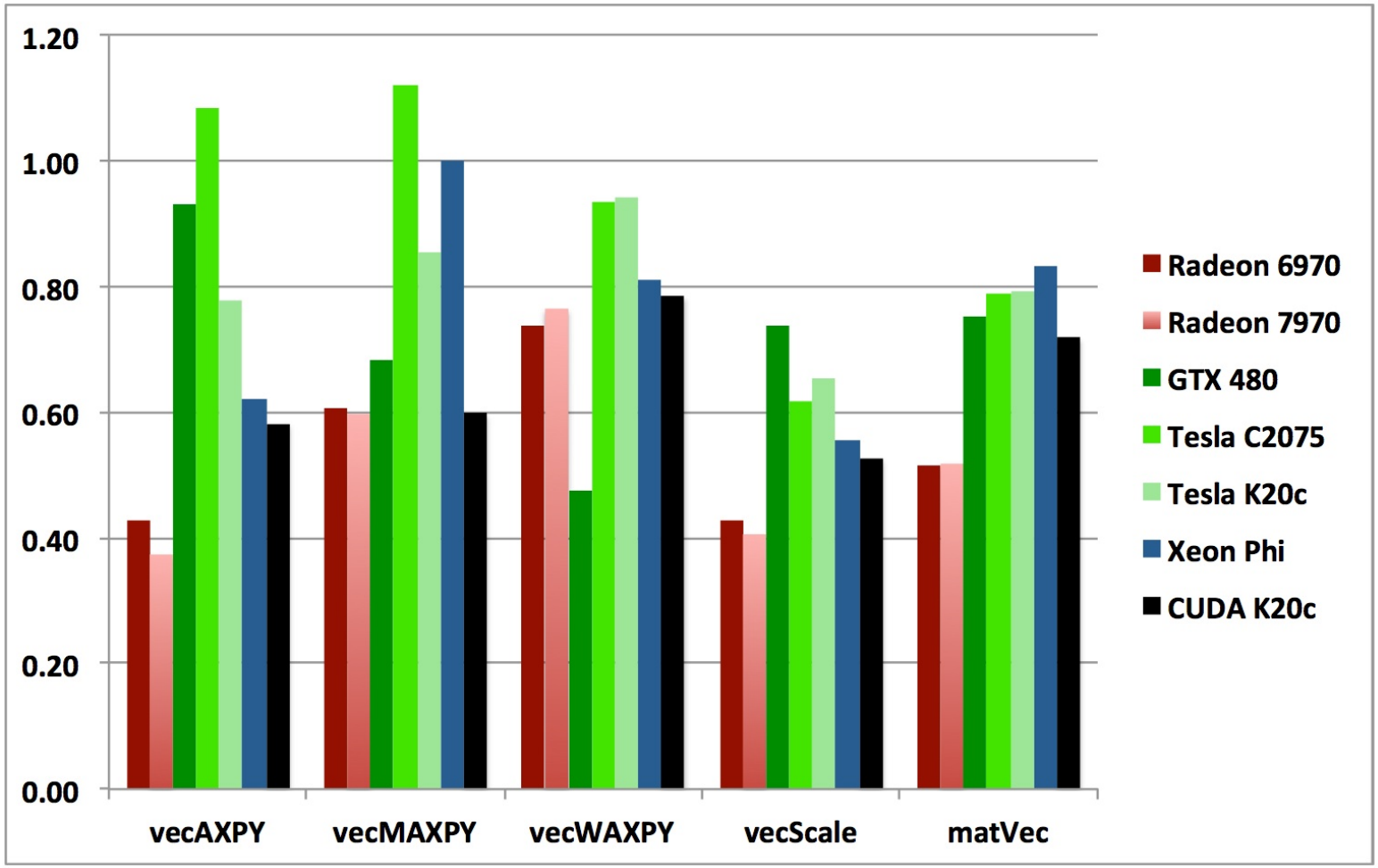
Experimental Evaluation

- Series of autotuning experiments on 6 devices
  - 2 AMD GPUs,
  - 3 NVIDIA GPUs
  - Intel Xeon Phi
- Verify OpenCL code generator on BLAS kernels previously used with the OrCUDA code generator
- Autotuned a subset of kernels that typically consume a significant portion of the solution time of Newton-Krylov nonlinear solvers
- Also autotuned the function and Jacobian computations of a PETSc-based application solving a 3-D solid fuel ignition (SFI) problem
  - Relies on the vectors kernels

Vector Kernel Specifications

Kernel	Operation
matVec	$y = Ax$
vecAXPY	$y = \alpha x + y$
vecMAXPY	$y = y + \alpha_1 x_1 + \alpha_2\, x_2 + \cdots + \alpha_n x_n$
vecScale	$w = \alpha w$
vecWAXPY	$w = y + \alpha x$

Kernel Performance



Normalized by ViennaCL times

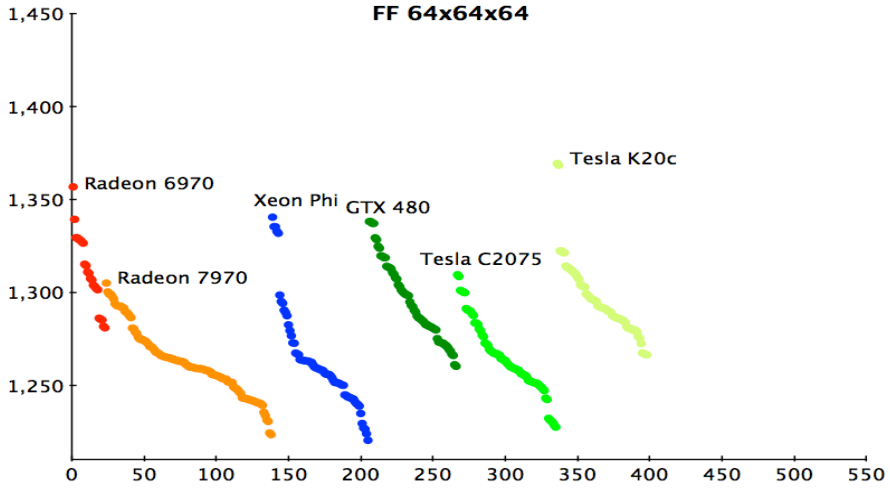
Tuning Platforms and Autotuned Parameters

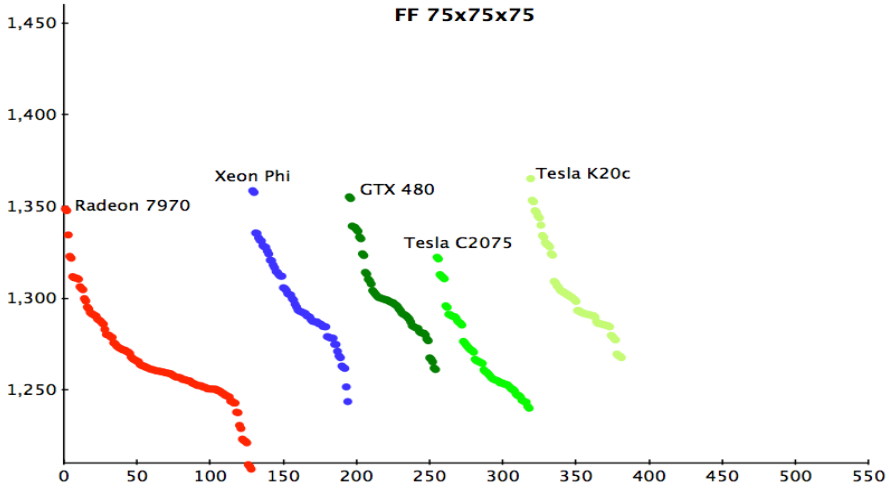
Accelerator Device	Radeon 6970	Radeon 7970	GTX 480	Tesla C2075	Tesla K20C	Xeon Phi
OpenCL Version	1.2	1.2	1.1	1.1	1.1	1.2
Max Compute Units	24	32	15	14	13	204
Max Work Items	(256,256,256)	(256,256,256)	(1024,1024,64)	(1024,1024,64)	(1024,1024,64)	(1024,1024,1024)
Max Workgroup Size	256	256	1024	1024	1024	1024
Clock Frequency	880 MHz	1000 MHz	1401 MHz	1147 MHz	705 MHz	2000 MHz
Cache Size	None	16 KB	24 KB	224 KB	208 KB	None
Global Memory Size	1024 MB	2048 MB	1535 MB	5375 MB	4800 MB	2835 MB
Constant Buffer Size	64 KB	64 KB	64 KB	64 KB	64 KB	128 KB
Local Memory Size	32 KB	32 KB	48 KB	48 KB	48 KB	32 KB
Preferred Workgroup Size Multiple	64	64	32	32	32	16

Accelerator Device	Radeon 6970	Radeon 7970	Xeon Phi
ex14FF 64^3	(64,64,",1,False,0)	(64,32,",2,False,0)	(64,64,",2,True,2)
ex14FF 75^3	N/A	(16,32,'cl-fast-relaxed-math',4,False,4)	(32,128,",2,False,2)
ex14FF 100^3	N/A	(32,32,",4,True,0)	(128,64,",1,True,0)
ex14FF 128^3	N/A	(32,64,",4,True,4)	(16,256,",1,False,0)
ex14FJ 64^3	N/A	(64,64,'cl-fast-relaxed-math',2,False,2)	(128,64,'cl-fast-relaxed-math',2,True,2)
ex14FJ 75^3	N/A	(64,256,'cl-fast-relaxed-math',1,False,0)	(64,256,",2,False,2)
ex14FJ 100^3	N/A	(128,128,",2,False,0)	(16,32,",2,True,2)
ex14FJ 128^3	N/A	(32,128,",2,False,2)	(32,64,'cl-fast-relaxed-math',4,False,2)

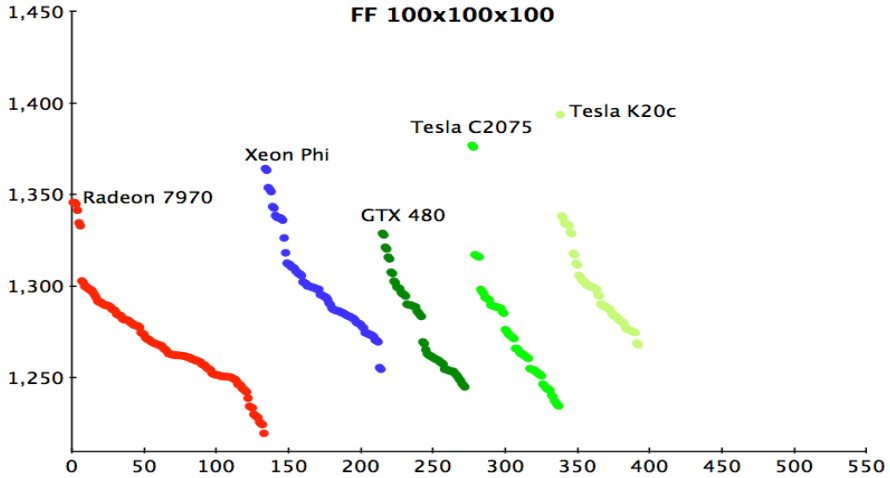
Accelerator Device	GTX 480	Tesla C2075	Tesla K20c
ex14FF 64^3	(16,64,",1,False,2)	(64,64,",2,True,0)	(64,128,",1,False,0)
ex14FF 75^3	(128,64,'cl-fast-relaxed-math',2,True,2)	(16,128,'cl-fast-relaxed-math',2,False,0)	(32,32,'cl-fast-relaxed-math',2,True,0)
ex14FF 100^3	(128,128,'cl-fast-relaxed-math',4,True,0)	(64,128,",1,True,2)	(64,64,'cl-fast-relaxed-math',4,False,2)
ex14FF 128^3	(32,32,",2,False,2)	(64,128,'cl-fast-relaxed-math',1,True,0)	(32,32,'cl-fast-relaxed-math',False,0)
ex14FJ 64^3	(16,64,",2,False,0)	(64,128,",2,True,2)	(32,64,",1,True,0)
ex14FJ 75^3	(64,256,",2,False,2)	(64,128,'cl-fast-relaxed-math',2,True,2)	(128,64,'cl-fast-relaxed-math',1,True,0)
ex14FJ 100^3	(32,128,",2,False,2)	(32,64,'cl-fast-relaxed-math',1,True)	(32,256,",1,False,4)
ex14FJ 128^3	(32,128,'cl-fast-relaxed-math',4,False,2)	(32,128,'cl-fast-relaxed-math',2,True,2)	(32,64,",4,False,0)

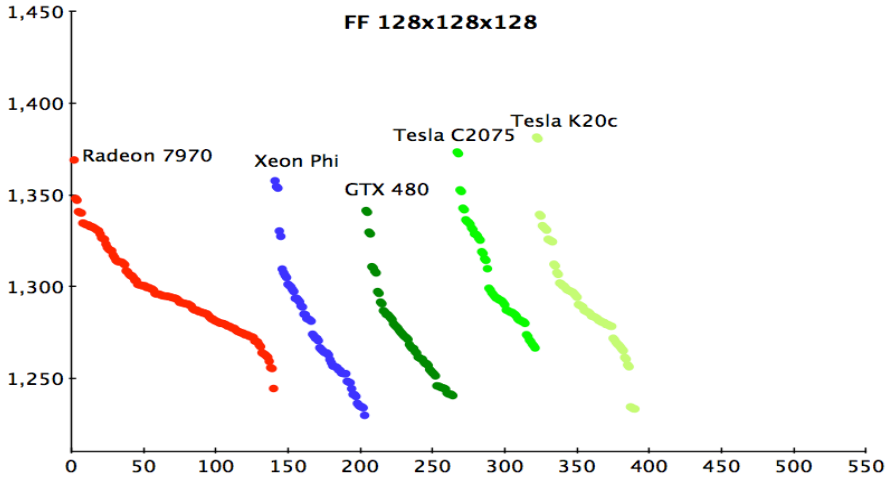
Autotuned Performance of Evaluated Variants



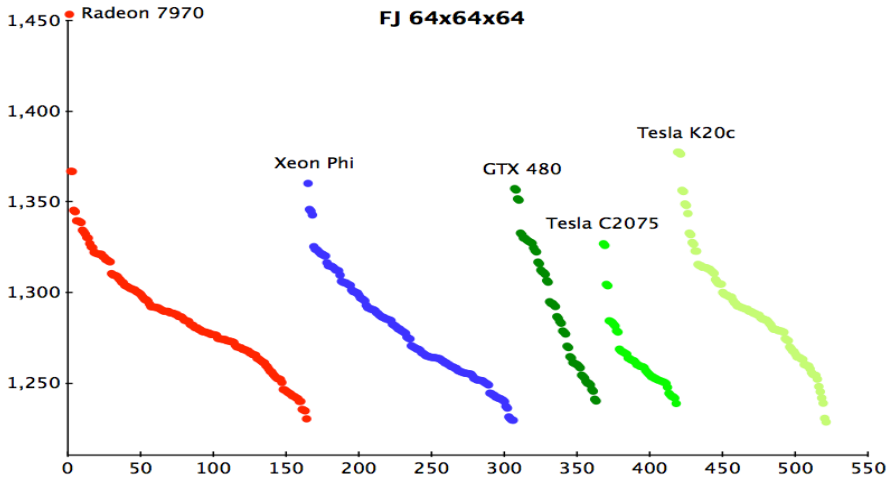


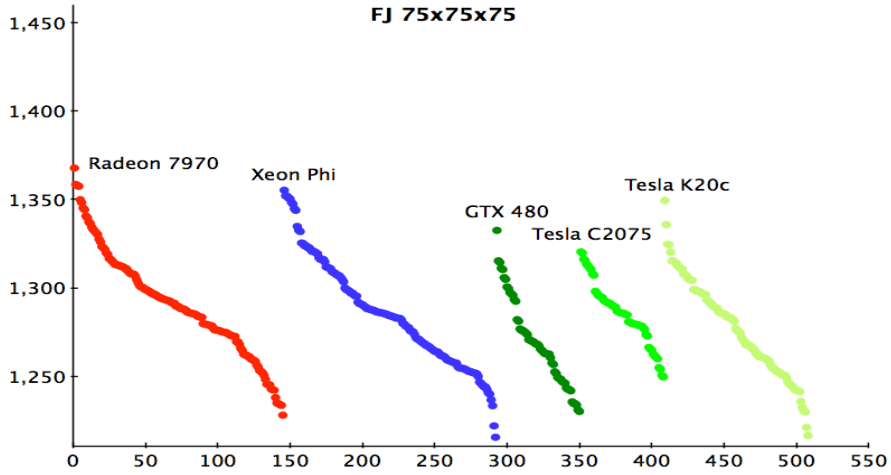
FormFunction (FF)



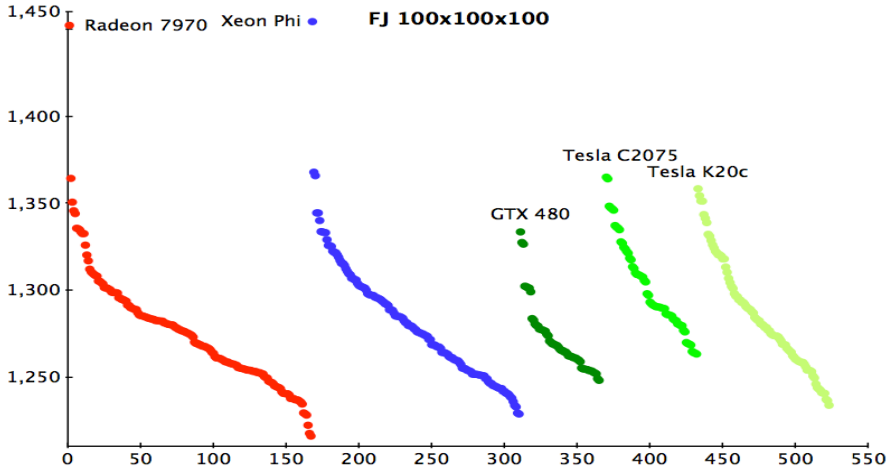


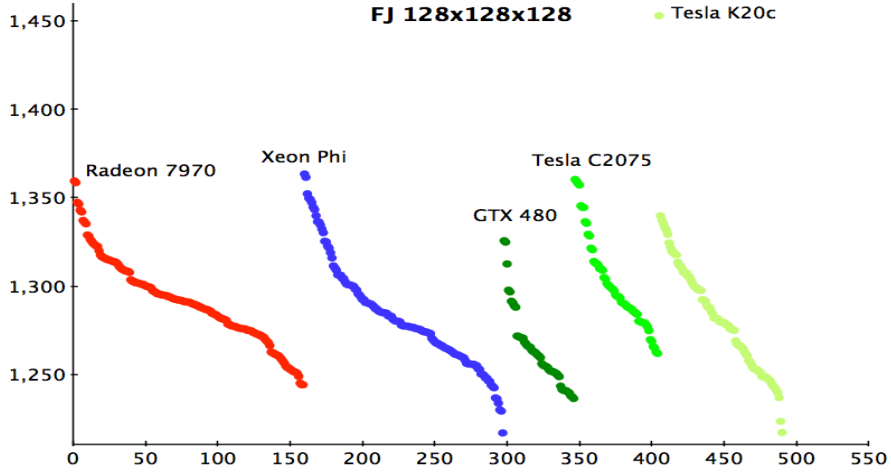
Autotuned Performance of Evaluated Variants



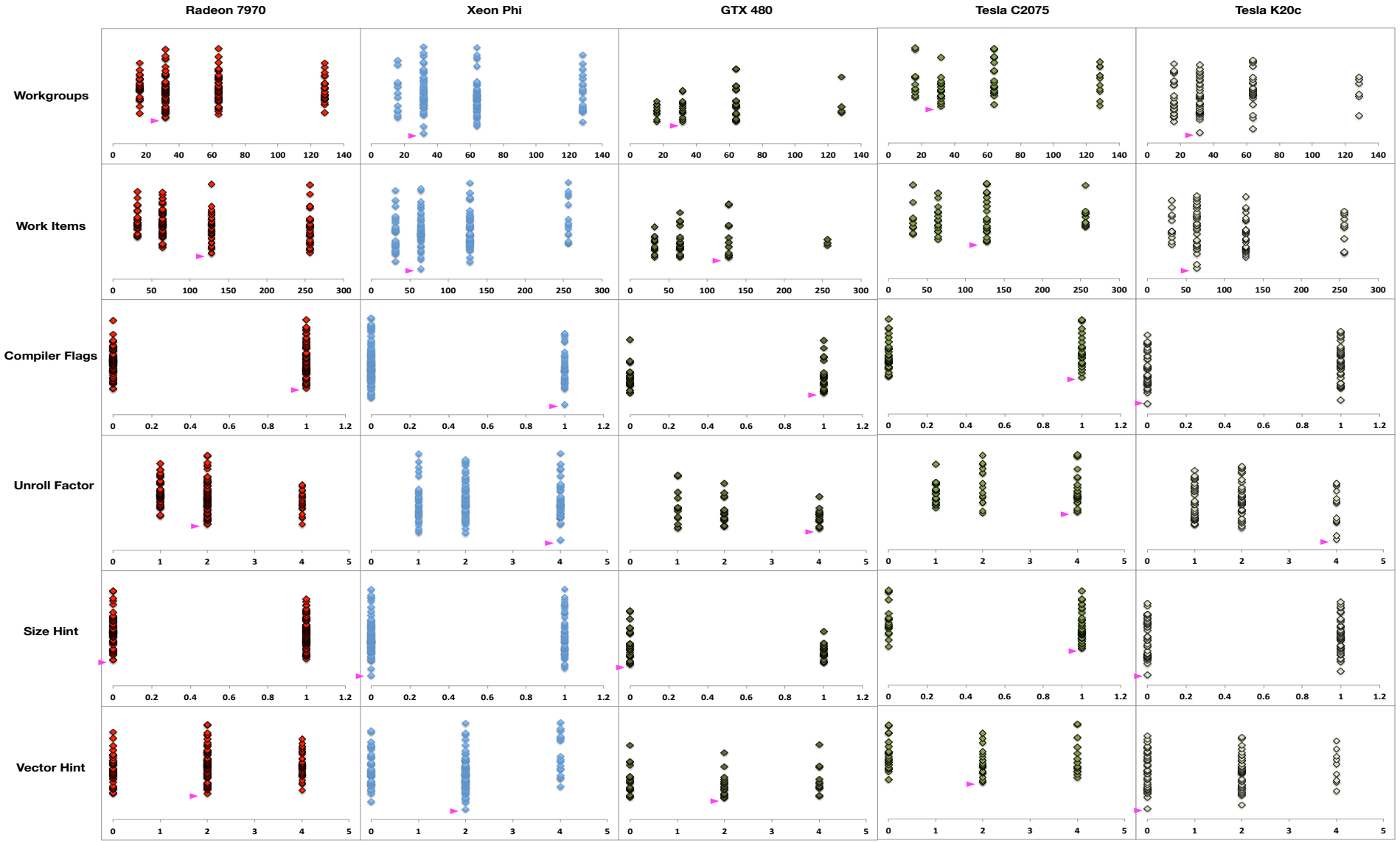


FormJacobian (FJ)





Autotuning Factors



Autotuning Autotuning

- Apply the autotuning methodology to autotuning to improve the search process.
- Use different search algorithms (or search parameters) and capture performance data and metadata as above
  - Also capture static code properties
- Learn a classifier to predict which search algorithm to use, using as features
  - Properties of code
  - Properties of input data
  - Properties of execution environment