Compilers and Compiler-based Tools for HPC

Recent Achievements

John Mellor-Crummey
Department of Computer Science
Rice University

http://lacsi.rice.edu/review/slides_2006/

LACSI

Participants

- Graduate Students
  - —Yuan Zhao, Nathan Froyd, Apan Qasem, Yuri Dotsenko, Cristian Coarfa
- Research Staff
  - —Nathan Tallent, Fengmei Zhao
- Research Scientists
  - —Robert Fowler, Guohua Jin
- Faculty
  - —Ken Kennedy, John Mellor-Crummey

- LANL Interactions
  - —David Montoya, Chip Kent, Hank Alme, Jeff Brown, Olaf Lubeck, Craig Rasmussen, Matt Sottile, Greg Watson

Overview of Ongoing and Future Impact

Long Term Research Affecting Future HPC Systems

High-level data-parallel programming systems

Source-to-source transformation tools

Open source compilers

Compilation techniques for SPMD languages (UPC, CAF)

Performance & memory diagnosis tools

Fundamental compiler algorithms

Compiling for computational accelerators

Detailed performance modeling of applications

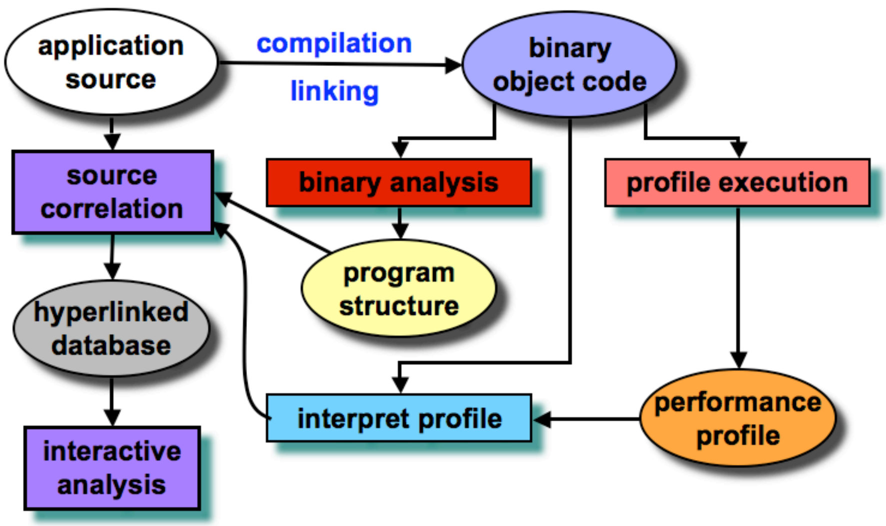
Hand application of aggressive transformations to important codes

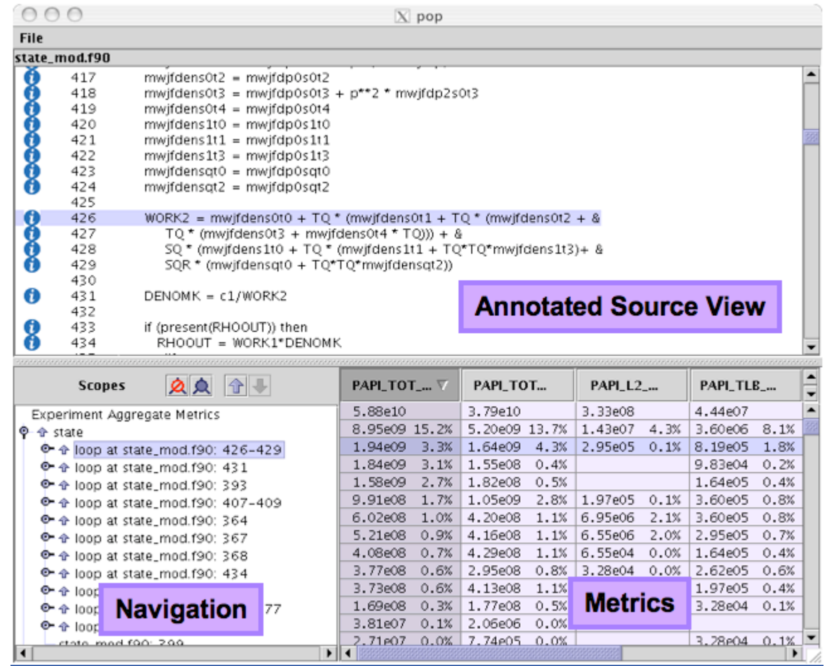
Immediate Impact in Support of ASC Mission Goals

Outline

- Tools for analyzing program performance and correctness
  - —call stack profiling
  - —memory analysis
- Performance modeling
  - —predicting memory hierarchy response for scientific applications
- Compiler technology
  - —computational accelerators
  - —programming models for scalable parallel systems
    - – Co-array Fortran
    - – compiler technology for global view languages

Code Performance and the ASC Mission





- **Performance of ASC codes is an area of ongoing concern**
  - **—LANL performance team: Peery, Graham, Alme[2], Brown, Koch**
- **Today: LANL using Rice's HPCToolkit to assess code performance**
- **Issue: HPCToolkit doesn't address the whole problem**

Background: HPCToolkit's Flat Profiles

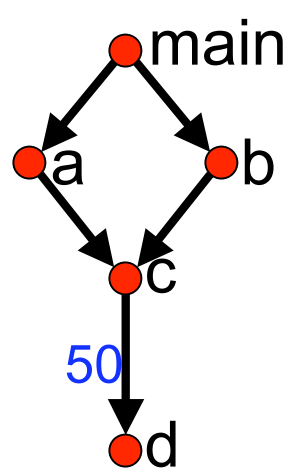
- What: measure resources consumed by an application
  - —time
  - —memory accesses
  - —cache misses
- How: statistical sampling
  - —time
  - —hardware performance counter events
- Where: attribute resource consumption back to source code
  - —procedures
  - —source lines
  - —loops
- What's missing: calling context

Understanding Costs In Context

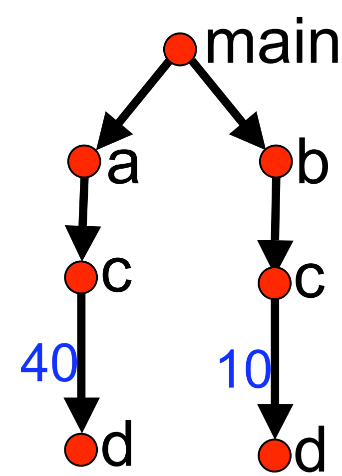
Call Path Profiling

- **Measure resource consumption in each procedure**
- **Attribute upward along call chain**
- **Report average consumption per call per calling context**

Call Graph



Calling Context Tree



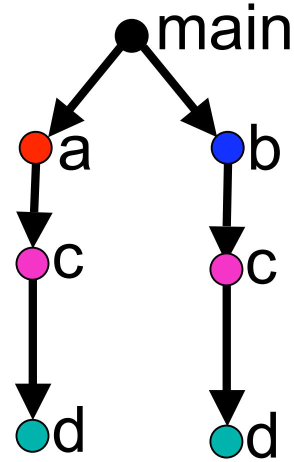
Why Calling Context Matters

- Modern program development strategies
  - —layered design
    - – communication libraries in parallel codes
    - – math libraries
  - —generic programming, e.g. C++ templates
    - – both data structures and algorithms
- Resource consumption is extremely context dependent
  - —which call to MPI in an application blocks the longest?
  - —which matrix solve consumes the most time?
  - —which instance of the C++ map template is costly?

Improving code performance requires knowing how code is used

A Tiny Motivating Example

```
#define HUGE (1<<28)

void d() {}

void c(long n) {
  for(int j=0; j<HUGE/n; j++) d();
}

void a(void (*f)(long)) { f(1); f(1); }

void b(void (*f)(long)) { f(2); f(2); f(2); f(2); }

void main() { a(c); b(c); }
```



Results with Existing Tools

(for our motivating example)

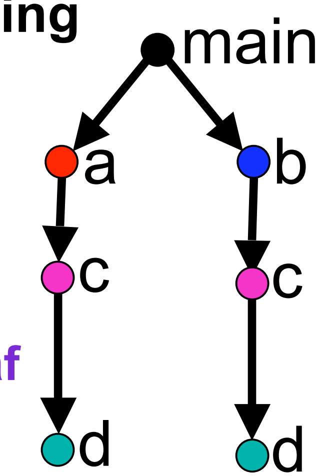
- **Instrumentation-based profilers**
  - —Vtune
    - – increases execution time by a <u>factor of 31</u> (P4+Linux)
  - —gprof
    - – cannot distinguish different costs per call for calling contexts
      - average time assumption
    - – increases execution time by a <u>factor of 3 - 14</u> (P4, PowerPC, Alpha)
- **Pure callstack sampling profilers**
  - —Shark, scgprof, qprof
    - – cannot distinguish different costs per call for calling contexts
      - know full contexts in which costs were incurred
      - no knowledge of how many calls per context

csprof: 1.5% overhead; accurate context-based attribution

LACSI

Our Approach

- **Attribute events to calling context with call stack sampling**
  - **—at each sample event**
    - **walk the call stack to discover calling context**
      - **chain of callsite PCs + current PC**
    - **record the calling context in a tree**
      - **insert calling context as a path from tree root to leaf**
    - **increment sample count for path leaf**



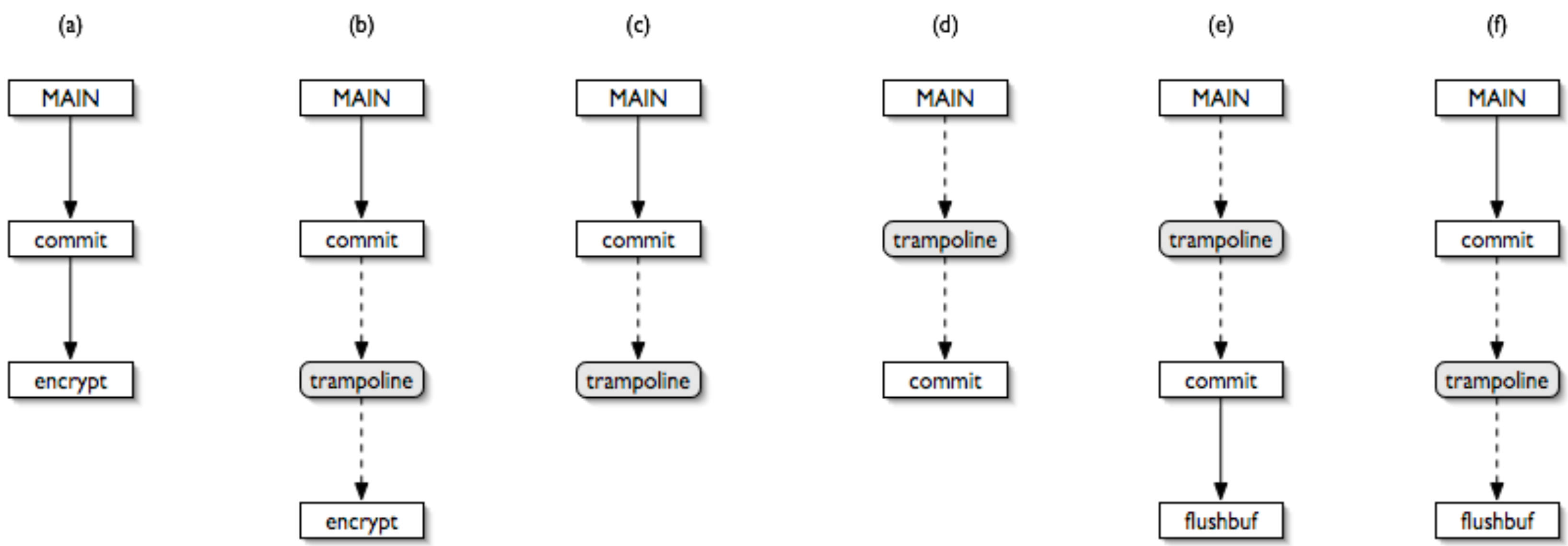
- **Associate a frequency count with each edge in the context tree**

Nathan Froyd. Efficient Call Graph Profiles on Unmodified Optimized Code. Masters Thesis, Dept. of Computer Science, Rice University, April 2005.

Nathan Froyd, John Mellor-Crummey, and Rob Fowler. “Low-Overhead Call Path Profiling of Unmodified, Optimized Code.” ICS 05, Cambridge, MA, June 2005.

Edge Counting with a Trampoline

- At each sample
  - —remove inserted trampoline (if any)
  - —interpose a trampoline between leaf and caller
- When a trampoline is triggered
  - —increment count for associated call edge
  - —move trampoline up one level in the call stack



Benefits of Our Approach

- **Supports profiling of fully-optimized code**
  - —doesn’t disrupt optimization with instrumentation
  - —permits optimized procedure linkage
    - – no frame pointers, register frame procedures, tail calls
- **Operates with low, controllable overhead**
  - —overhead proportional to sampling frequency <u>not</u> calling frequency
- **Minimizes distortion of application performance**
  - —no instrumentation of function entries: minimizes call dilation
- **Requires no changes to build process**
  - —no special compilation (e.g. gprof’s compile-time instrumentation)
  - —initiates monitoring at program launch using preloaded library

Alpha Experiments: CINT2000 Benchmarks

Benchmark	Base time (seconds)	gprof overhead (%)	gprof number of calls	csprof overhead (%)
164.gzip	479	53	1.960E+09	4.2
175.vpr	399	53	1.558E+09	2.0
176.gcc	250	78	9.751E+08	N/A
181.mcf	475	19	8.455E+08	8.0
186.crafty	196	141	1.908E+09	5.1
197.parser	700	167	7.009E+09	4.6
252.eon	245	263	1.927E+09	3.4
253.perlbmk	470	165	2.546E+09	2.5
254.gap	369	39	9.980E+08	4.1
255.vortex	423	230	6.707E+09	5.4
256.bzip2	373	112	3.205E+09	1.1
300.twolf	568	59	2.098E+09	3.0
Average overhead		**115**		**3.9**

Alpha Experiments: CFP2000 Benchmarks

Benchmark	Base time (seconds)	gprof overhead (%)	gprof number of calls	csprof overhead (%)
168.wupwise	353	85	2.233E+09	2.5
171.swim	563	0.17	2.401E+03	2.0
172.mgrid	502	0.12	5.918E+04	2.0
173.applu	331	0.21	2.192E+05	1.9
177.mesa	264	67	1.658E+09	3.0
178.galgel	249	5.5	1.490E+07	3.2
179.art	196	2.1	1.110E+07	1.5
183.equake	549	0.75	1.047E+09	7.0
187.facerec	267	9.4	2.555E+08	1.5
188.ammp	547	2.8	1.006E+08	2.7
189.lucas	304	0.3	1.950E+02	1.9
191.fma3d	428	18	5.280E+08	2.3
200.sixtrack	436	0.99	1.030E+07	1.7
301.apsi	550	12	2.375E+08	1.6
Average overhead		**14.6**		**2.5**

LACSI

Assessing Profiler Distortion

- **How accurately does profiler assign costs to individual functions?**

$$distortion(p, X) = \sum_{f \in functions(p)} P_X(f) - P_{DCPI}(f)$$

—**measure "baseline" with DCPI: close to real program behavior**

- **Results**

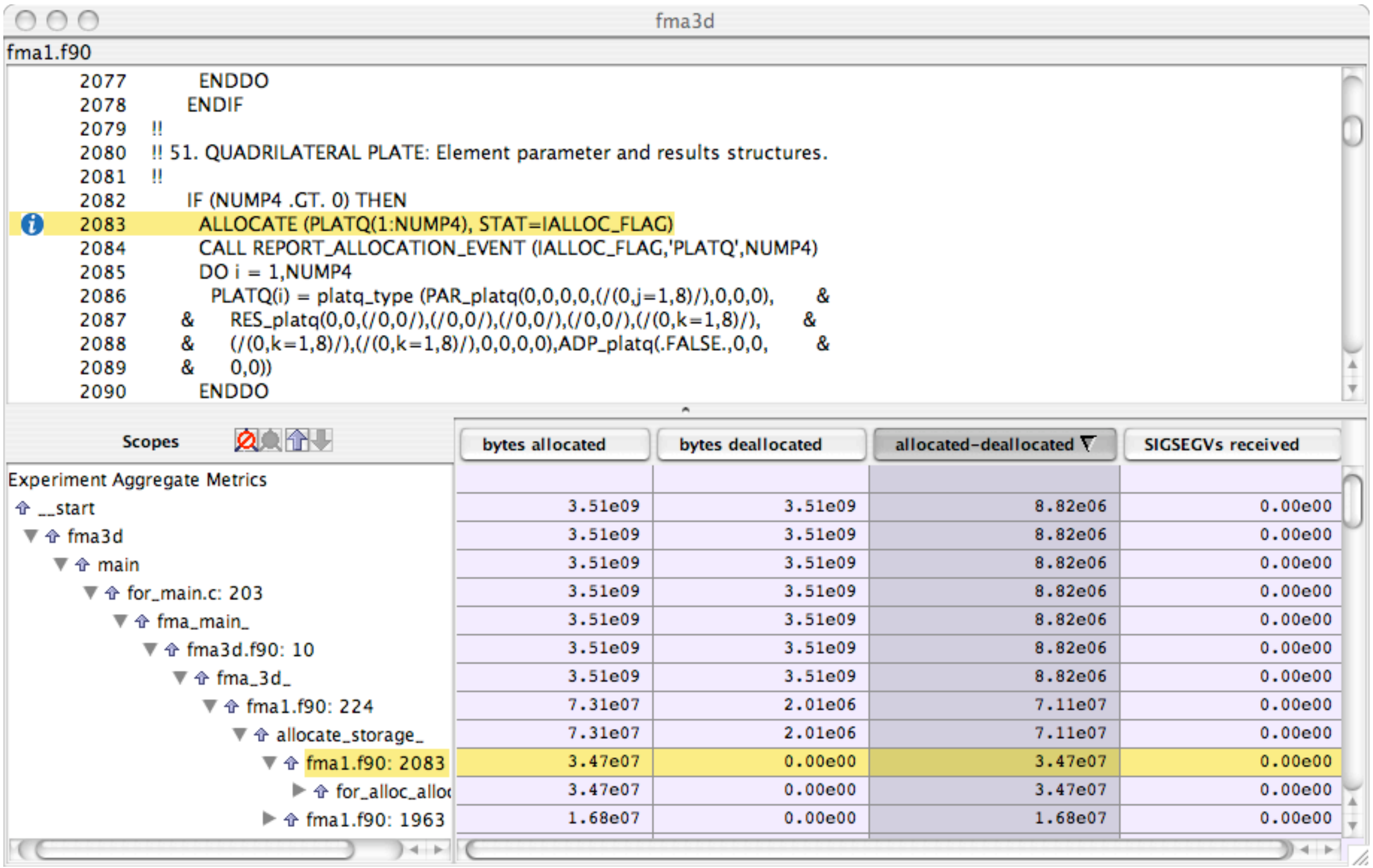
	Integer programs csprof	Integer programs gprof	Floating-point programs csprof	Floating-point programs gprof
Minimum	0.7	7.6	0.4	0.3
Median	2.9	15.0	3.6	2.4
Mean	8.0	23.0	5.0	4.1
Maximum	51.0	120.0	18.0	15.0

- **csprof accurately attributes to calling context with low distortion**
  —**less distortion for integer benchmarks**
  —**competitive for the FP benchmarks**

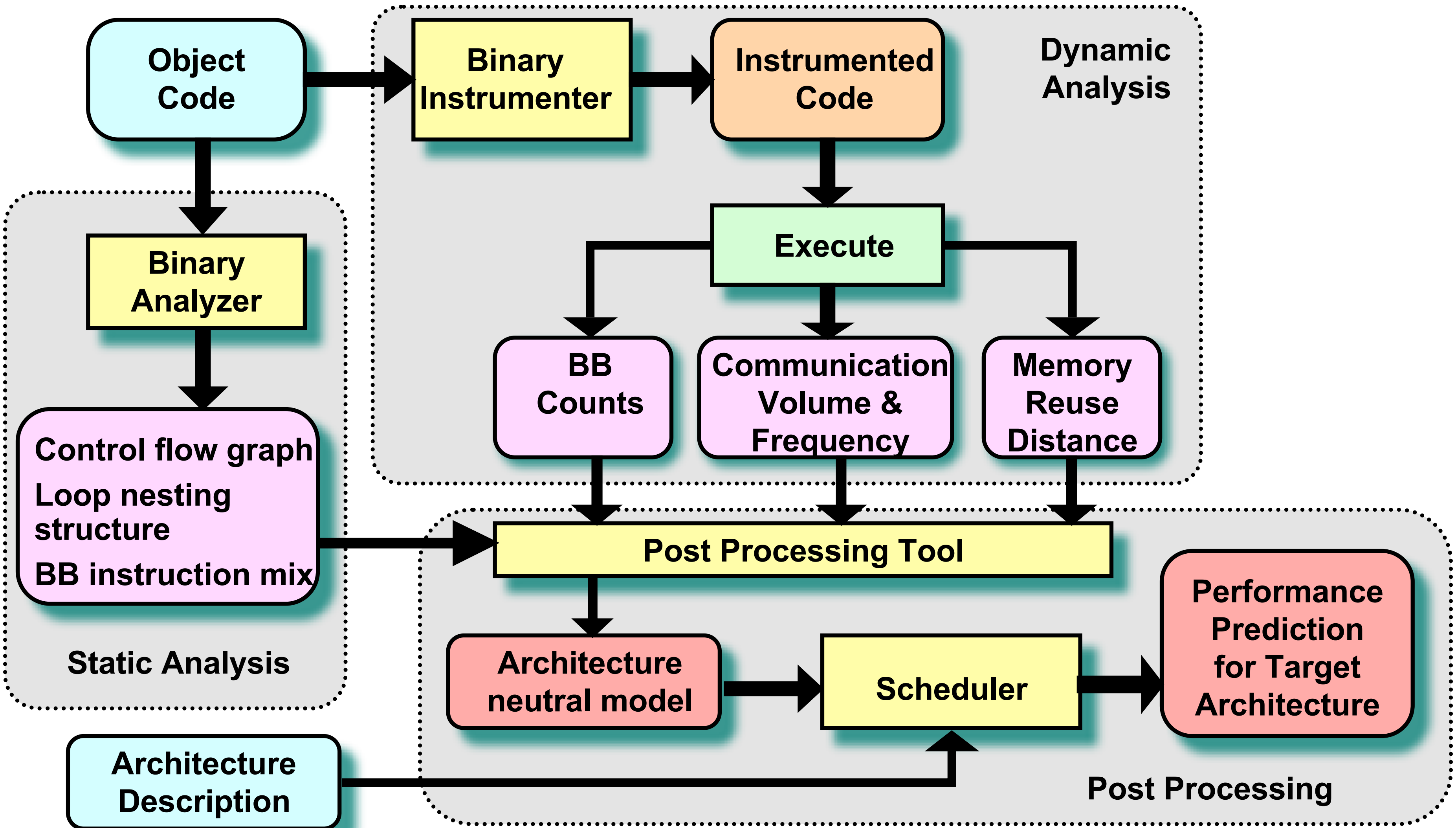
From Profiling to Memory Leak Diagnosis

- **LANL believes that memory leaks cause a classified code to fail**
  - —discussed in June DRC meeting and during November visit by Rice
- **Finishing a memory leak diagnosis tool for production codes**
  - —use call stack profiling infrastructure
  - —use a preloaded library to synchronously profile malloc and free
    - – malloc: bytes allocated in each calling context
    - – free: bytes deallocated in each calling context
  - —catch and report asynchronous segmentation fault (if any)

Memprof


fma3d
fma1.f90
2077 ENDDO
2078 ENDIF
2079 !!
2080 !! 51. QUADRILATERAL PLATE: Element parameter and results structures.
2081 !!
2082 IF (NUMP4 .GT. 0) THEN
2083 ALLOCATE (PLATQ(1:NUMP4), STAT=IALLOC_FLAG)
2084 CALL REPORT_ALLOCATION_EVENT (IALLOC_FLAG,'PLATQ',NUMP4)
2085 DO i = 1,NUMP4
2086 PLATQ(i) = platq_type (PAR_platq(0,0,0,0,(/(0,j=1,8)/),0,0,0), &
2087 & RES_platq(0,0,(/0,0/),(/0,0/),(/0,0/),(/0,0/),(/(0,k=1,8)/), &
2088 & (/(0,k=1,8)/),(/(0,k=1,8)/),0,0,0,0),ADP_platq(.FALSE.,0,0, &
2089 & 0,0))
2090 ENDDO
Scopes
bytes allocated
bytes deallocated
allocated-deallocated
SIGSEGVs received
Experiment Aggregate Metrics
__start 3.51e09 3.51e09 8.82e06 0.00e00
fma3d 3.51e09 3.51e09 8.82e06 0.00e00
main 3.51e09 3.51e09 8.82e06 0.00e00
for_main.c: 203 3.51e09 3.51e09 8.82e06 0.00e00
fma_main_ 3.51e09 3.51e09 8.82e06 0.00e00
fma3d.f90: 10 3.51e09 3.51e09 8.82e06 0.00e00
fma_3d_ 3.51e09 3.51e09 8.82e06 0.00e00
fma1.f90: 224 7.31e07 2.01e06 7.11e07 0.00e00
allocate_storage_ 7.31e07 2.01e06 7.11e07 0.00e00
fma1.f90: 2083 3.47e07 0.00e00 3.47e07 0.00e00
for_alloc_alloc 3.47e07 0.00e00 3.47e07 0.00e00
fma1.f90: 1963 1.68e07 0.00e00 1.68e07 0.00e00

Performance Modeling Toolkit



LACSI

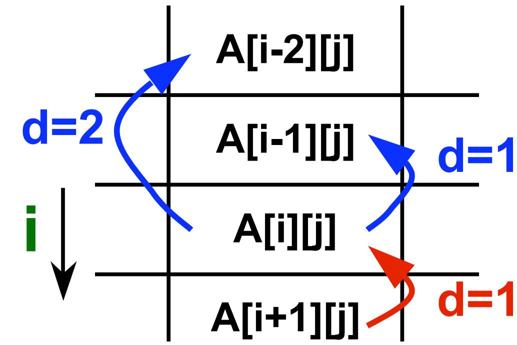
Predicting Schedule Latency for an Architecture

- **Input:**
  - —basic block and edge execution frequencies
- **Methodology:**
  - —infer executed paths from BB and edge frequencies
  - —native instructions → generic RISC instructions
  - —instantiate scheduler with architecture description
  - —construct instruction schedule for executed paths
    - – consider instruction latencies and dependencies

Gabriel Marin and John Mellor-Crummey. "Scalable Cross-Architecture Predictions of Memory Hierarchy Response for Scientific Applications." Proceedings of LACSI 2005 (October 2005).

Scheduling Graph Example

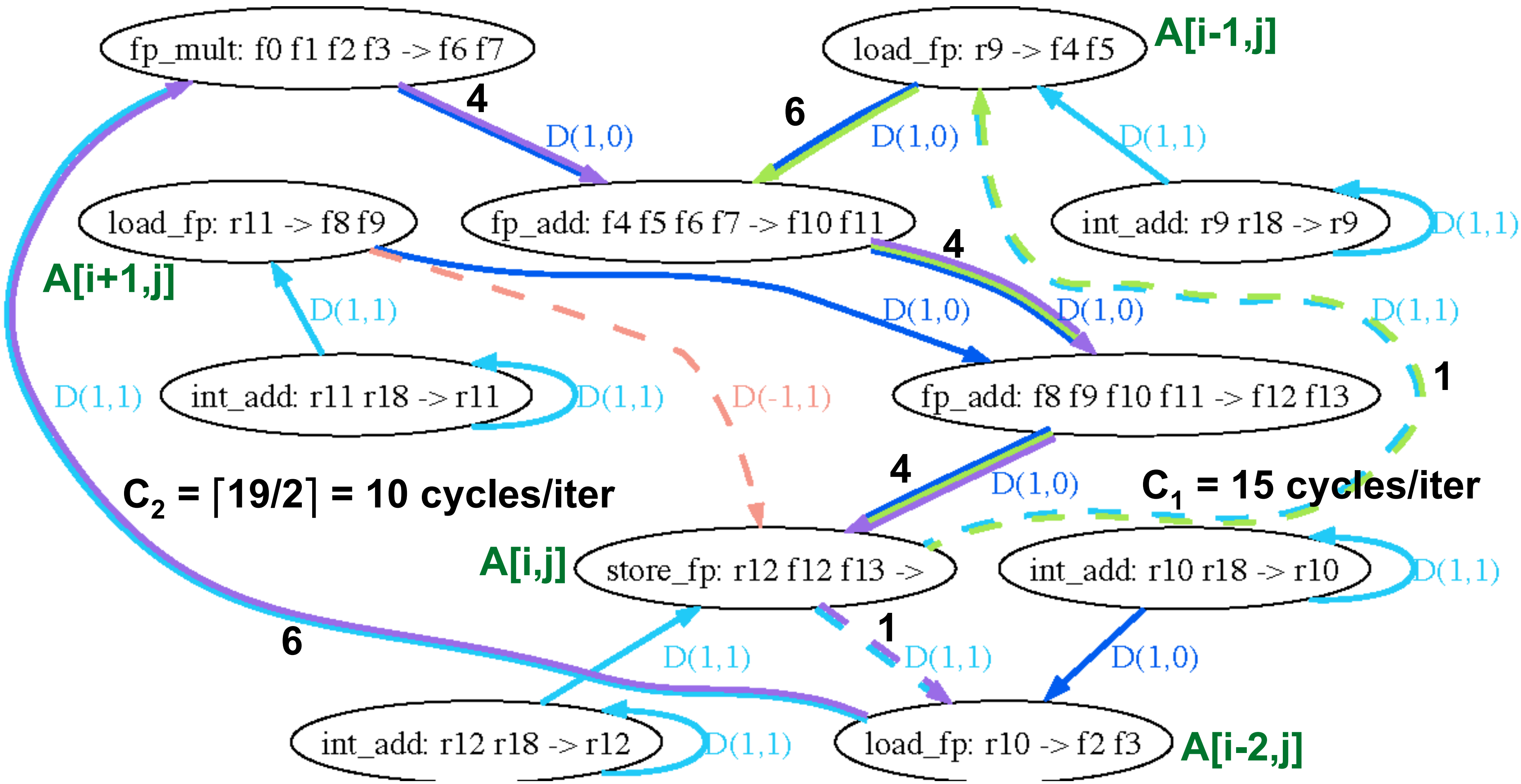
```
void compute(int size, double* A, double c1) {
  for( int j=0 ; j<size ; ++j )
    for( int i=2 ; i<size-1 ; ++i )
      A[i*size+j] =
          c1*A[(i-2)*size+j] +
          A[(i-1)*size+j] + A[(i+1)*size+j];
}
```



			First location	*i*-stride	*j*-stride
	.L2: fmuld	%f0, %f2, %f6			
A[i-1,j]	ldd	[%o1], %f4	8*%i0+%i1	8*%i0	8
	add	%o5, 0x1, %o5			
A[i+1,j]	ldd	[%o3], %f8	24*%i0+%i1	8*%i0	8
	add	%o2, %l2, %o2			
	add	%o1, %l2, %o1			
	add	%o3, %l2, %o3			
	cmp	%o5, %l4			
	faddd	%f6, %f4, %f10			
	faddd	%f10, %f8, %f12			
	std	%f12, [%o4]			
A[i,j]	add	%o4, %l2, %o4	16*%i0+%i1	8*%i0	8
	bl,a,pt	%icc,.L2			
	ldd	[%o2], %f2			
A[i-2,j]			8*%i0+%i1	8*%i0	8

LACSI

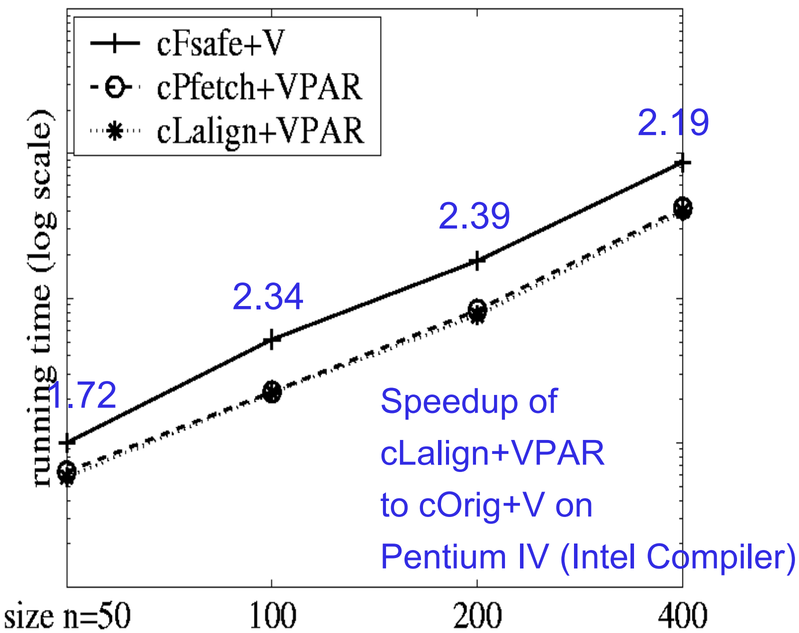
Scheduling Graph Example

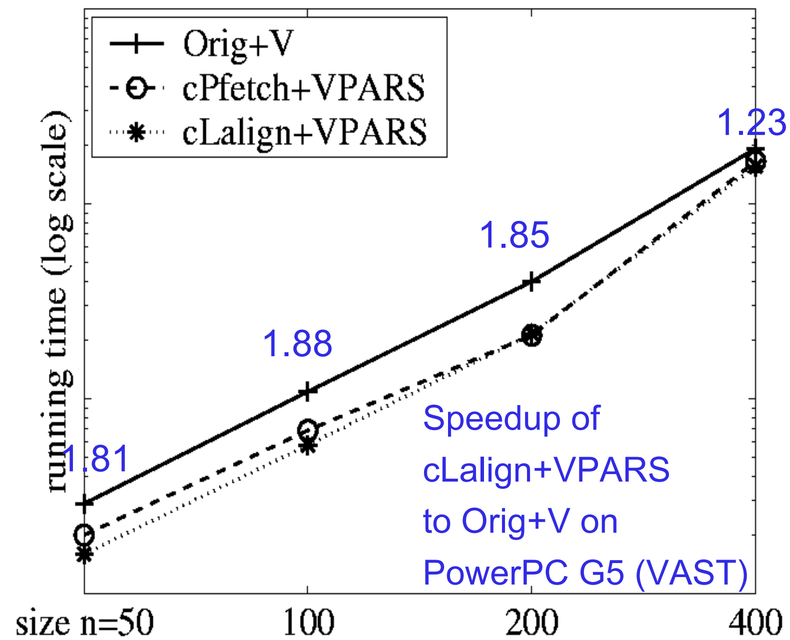


Compiling for Computational Accelerators

- **Employ integrated techniques for vectorization, padding, alignment, scalar replacement to compile for short vector machines**

DO J = 2, N-1; A(2:N-1, J) = (A(2:N-1, J-1) + A(2:N-1, J+1) + A(1:N-2, J) + A(3:N, J))/4; ENDDO





- **Extending source-to-source vectorizer to support CELL**

Y. Zhao and K. Kennedy. Scalarization on Short Vector Machines." IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Austin, Texas, March 2005.

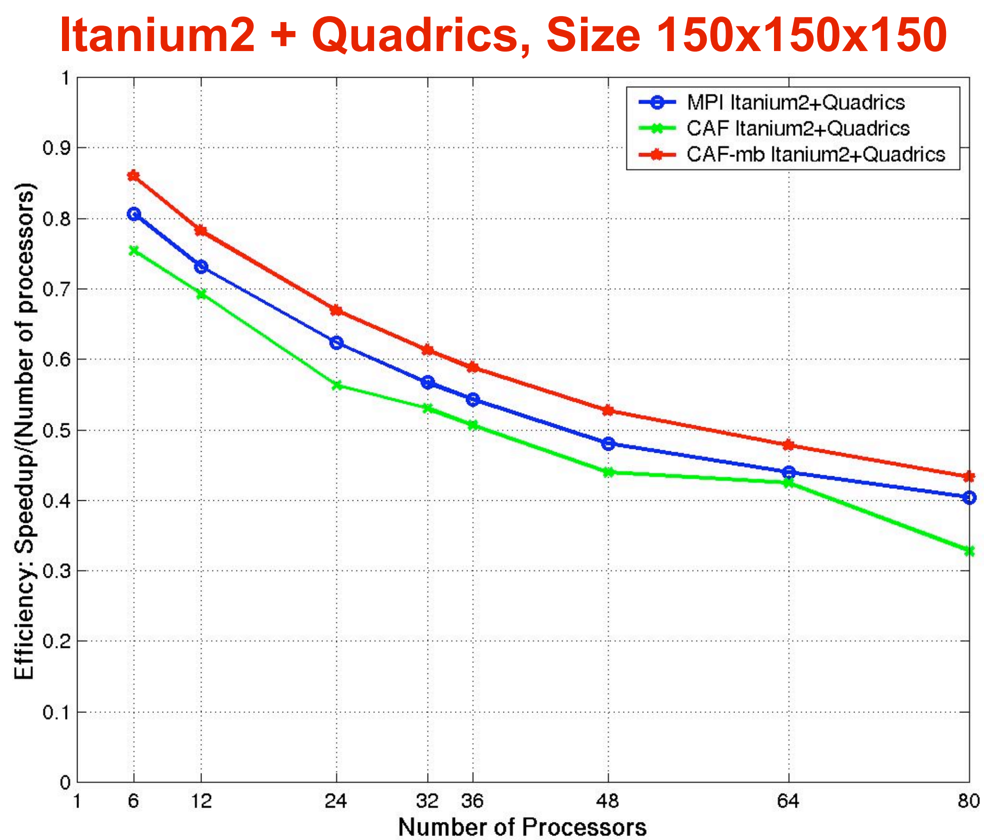
LACSI

Compilers for Scalable Parallel Systems

Parallel Programming Models

- MPI: de facto standard
  - —difficult to program
- OpenMP: inefficient to map on distributed memory platforms
  - —lack of locality control
- SPMD global address space languages
  - —CAF, Titanium, UPC
- Global view languages (e.g. HPCS languages, HPF)
  - —extremely sophisticated compilers needed for high-performance

CAF Sweep3D



Cristian Coarfa, Yuri Dotsenko, and John Mellor-Crummey. "Experiences with Sweep3D Implementations in Co-array Fortran." *Journal of Supercomputing*. (In Press)

Global View Programming = Productivity

Delegate difficult tasks to the compiler and runtime

- Managing local address space computations
  - —partitioning data
  - —partitioning computation
- Managing communication
  - —where communication is needed
  - —what must be communicated
- Managing and indexing storage for non-local data

Compiling Global View Languages

- Partition data
  - —follow user directives
- Select mapping of computation to processors
  - —co-locate computation with data
- Analyze communication requirements
  - —identify references that access off-processor data
- Partition computation by reducing loop bounds
  - —schedule each processor to compute on its own data
- Insert communication
  - —exchange values as needed by the computation
- Manage storage for non-local data

D. Chavarria-Miranda and J. Mellor-Crummey. "Effective communication coalescing for data-parallel applications." Symposium on Principles and Practice of Parallel Programming (June 15-17, 2005).

D. Chavarria-Miranda, G. Jin, and J. Mellor-Crummey. "COTS Clusters vs. the Earth Simulator: An Application Study Using IMPACT-3D." IPDPS 2005 (April 4-8, 2005).

Overview of Ongoing and Future Impact

Long Term Research Affecting Future HPC Systems

High-level data-parallel programming systems

Source-to-source transformation tools

Open source compilers

Compilation techniques for SPMD languages (UPC, CAF)

Performance & memory diagnosis tools

Fundamental compiler algorithms

Compiling for computational accelerators

Detailed performance modeling of applications

Hand application of aggressive transformations to important codes

Immediate Impact in Support of ASC Mission Goals

LACSI