Assigning Blame

Nick Rutar
Jeffrey K. Hollingsworth

University of Maryland

Motivation

- Parallel languages becoming more mature
- Parallel frameworks becoming more common
  - PETSc, Cactus, POOMA, GrACE
- Frameworks provide
  - High level abstractions for mathematics
    - Matrices, Vectors, (Non)Linear Systems, PDEs
  - Masking of low level parallel constructs
- More levels of abstraction complicates
  - Profiling
  - Debugging

Parallel Framework Mapping

- Traditional profiling represented as
  - Functions, Basic Blocks, Statement
- Frameworks have intuitive abstractions
  - Direct ties with mathematical terms
- Map profiling information to variables
  - Maps to abstractions in case of frameworks
  - Also can be used for standard programs
    - Map Structs, Classes, Arrays, Scalars

Example PETSC Program*

* - $PETSC_DIR/src/ksp/ksp/examples/tutorials/ex23.c

50% cache misses →

30% MPI operations →

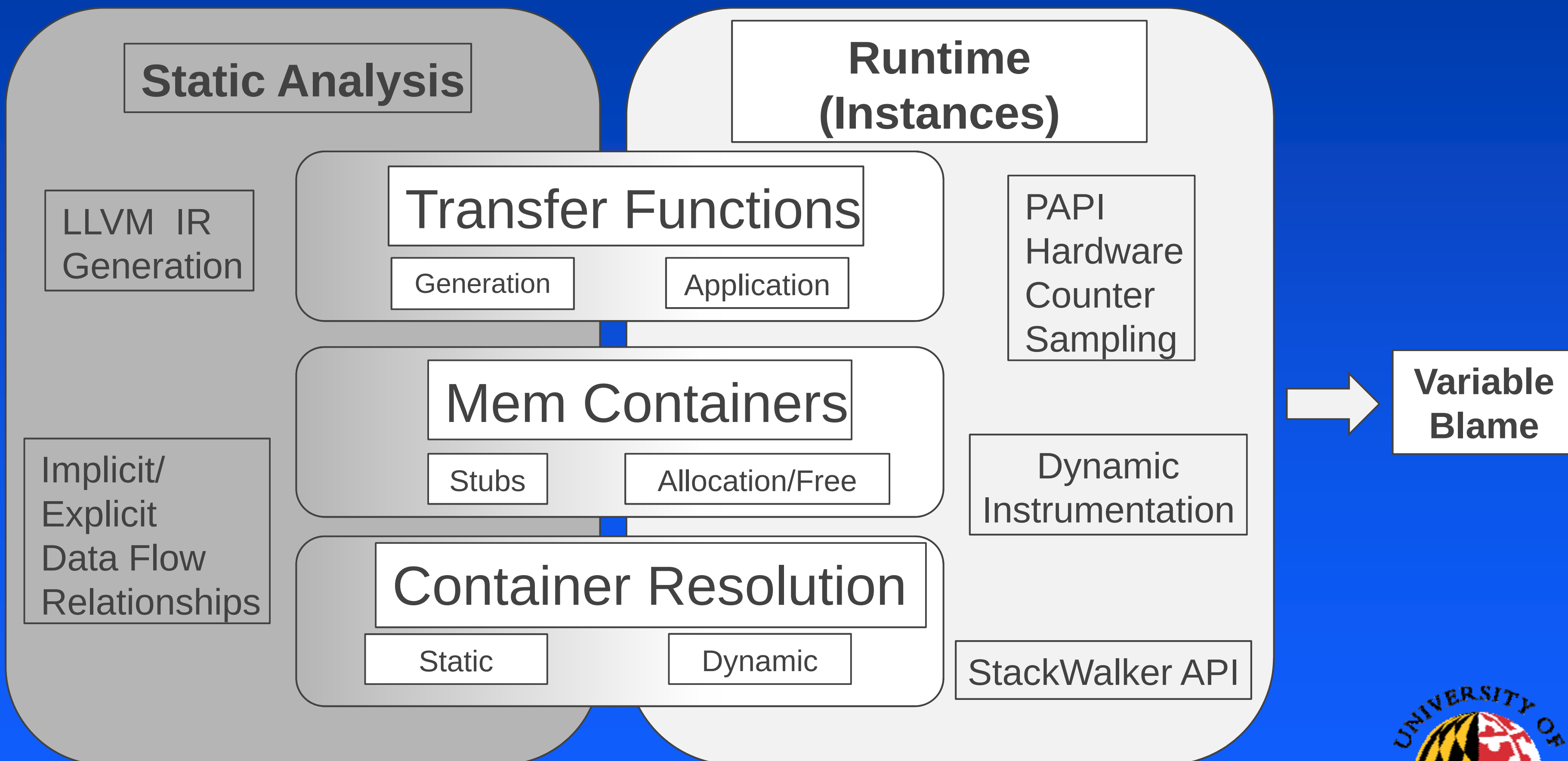
40% run time →

```
int main(int argc,char **args) {
  Vec     x, /* approx solution */
           b, /* right hand side */
           u;  /* exact solution*/
  Mat     A;          /* linear system matrix */
  KSP     ksp;        /* linear solver context */
  PC      pc;         /* preconditioner context */
 VecCreate(PETSC_COMM_WORLD,&x);
 VecDuplicate(x,&b);
 VecDuplicate(x,&u);
 MatCreate(PETSC_COMM_WORLD,&A);
 MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
 MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
 /* Set exact solution */
  VecSet(u,one);
  MatMult(A,u,b);
  /* Create linear solver context */
  KSPCreate(PETSC_COMM_WORLD,&ksp);
  KSPGetPC(ksp,&pc);
  PCSetType(pc,PCJACOBI);
  /* Solve linear system */
  ierr = KSPSolve(ksp,b,x); }
```

Variable "Blame"

- Record activity in a function
- Build association tree of writes from ground up
- Use transfer function to filter information up
  - Up the call stack
  - Aggregate over distributed nodes
- Eventually reach high level abstractions
  - Example: Matrix abstraction
    - Allocated storage for actual data
      - Sparse or Dense
    - Storage for bookkeeping
- Augments traditional profiling approaches

Blame Calculation Components



Data Flow Relationships for Blame

- Two kinds of relationships
- Explicit
  - Small sample snippet

```
int a, b, c;
a = 7;
b = 8;
c = a + b;
```

  - Blame goes to c
- Implicit
  - Control Flow Operations
    - Loop indices, conditional statements

Calculating Data Flow Relationships

- Use LLVM for intermediate representation
  - Allows same approach for all supported languages
    - C, C++, Fortran
- Calculate explicit blame
  - Create dependency graph based on data flow
  - Focus on nodes with no incoming edges
- Calculate implicit blame
  - Generate control flow graph & dominator tree
  - Calculate basic blocks affected by control flow

Transfer Functions

- Establish "Exit Variables" for each function
  - Those variables that are live after function ends
    - Parameters
    - Return Value
    - Modified Global/Static Variables
    - Generated Heap Variables
- Create transfer function in terms of exit variables
- Special transfer functions
  - Source is not available
    - Series of Heuristics used to calculate blame
      - Return value with no params, all blame to return
  - Well defined APIs
    - math.h
      - Know that all blame for sqrt(double) goes to return

Mem-Containers

- Representation of memory operations
  - Stack and Heap based
- Represents unique contiguous memory region
- Mappings handled with container resolution
  - Ultimately map up to program variables
  - Mem-containers can map to other mem-containers
- Discovered through static/dynamic analysis
  - Static determines allocation points
    - Stubs created at these points
  - Dynamic happens for each instantiation
    - Full path of allocation calculated

Instance Generation (Sampling)

- Represents operations at each sample
- Can use different metrics for sampling
  - L2 Cache misses, floating point ops, cycles
- Mapped to either mem-container or variable
- Information recorded per thread and node
- Gather context (stack) at each sample
- Implementation
  - PAPI to generate samples
  - Stackwalk API for context sensitive information

Final Variable Blame

- Given to user at various "blame points"
- Blame point can be
  - "main" function
  - Where blame cannot be propagated up
    - Set of exit variables is null
    - Function with void params & void return
  - Any function deemed interesting by user
  - Any function that matches defined criteria
    - Contains a variable that has threshold of blame

Experimental Results

- Chose three programs with similar properties to those found in parallel frameworks
- Blame metric is number of cycles
- For each sampling point (instance)
  - Instance gets blamed for set number of cycles
  - Variable that instance maps up to gets blame

FFP_SPARSE

- C++ program that solves Poisson's Equation
  - Approximately 6,700 lines of code & 63 Functions
- Non-parallel program
- Uses Sparse Matrices
  - No specific data structure for representation
  - Composite of primitive pointers declared in 'main'
- Recorded 101 samples from program run

FFP_SPARSE Results

Name	Type	Description	Direct	Blame (%)
node_u	double *	Solution vector	0	35 (34.7)
a	double *	Coefficient matrix	0	24.5 (24.3)
ia	int *	Non-zero row indices of a	1	5 (5.0)
ja	int *	Non-zero column indices of a	1	5 (5.0)
element_neighbor	int *	Estimate of non-zeroes	0	10 (9.9)
node_boundary	bool *	Bool vector for boundary	0	9 (8.9)
f	double *	Right hand side of vector	0	3.5 (3.5)
Other	-		99	9 (8.9)
Total	-		101	101 (100)

QUAD_MPI

- C++ MPI program
  - Approximates multidimensional integral
  - Approximately 2000 lines of code & 18 functions
- Program interesting to look at handling MPI
- Ran on 4 Red Hat Linux nodes
  - OpenMPI 1.28
  - Range of 94-108 samples per node

QUAD_MPI Results

		Blame (per Node)					
Name	**Type**	**N1 (%)**	**N2 (%)**	**N3(%)**	**N4(%)**	**Total (%)**	**Dominant MPI Call**
dim_num	int	27(27.2)	90(95.7)	97(84.3)	102(94.4)	316 (76.0)	MPI_Bcast
quad	double	19(19.2)	1(1.1)	5(4.3)	5(4.6)	30 (7.2)	MPI_Reduce
task_proc	int	15(15.2)	-	-	-	15 (3.6)	MPI_Send
w	double *	9(9.1)	-	-	-	9 (2.1)	-
point_num_proc	int	-	1(1.1)	7(6.1)	-	8 (1.9)	MPI_Recv
x_proc	double *	-	2(2.1)	5(4.3)	-	6 (1.4)	MPI_Recv
Other	-	3(3.0)	-	-	-	3 (0.7)	-
Output	-	6(6.1)	-	1(0.9)	1(0.9)	8 (1.9)	-
Total	-	99(100)	94(100)	115(100)	108(100)	416 (100)	-

HPL (Linpack)

- C program that solves a linear system
  - Utilizes MPI and BLAS
  - Has wrappers for functions from both libraries
  - Operations done on dense matrices
  - Approximately 18,000 lines of code
  - 149 source files
- 32 Red Hat nodes connected via Myrinet
  - OpenMPI 1.2.8
  - Range of 149-159 samples over the nodes

HPL Results

Blame Points

		Blame over 32 Nodes		
Name	**Type**	**Mean**	**(%)**	**Node St. Dev.**
All Instances	-	154.7	(100)	2.7
main				
grid	HPL_T_grid	2.2	(1.4)	0.4
main→HPL_pdtest				
mat	HPL_T_pmat	139.3	(90.0)	2.8
Anorm1	double	1.4	(0.9)	0.8
AnormI	double	1.1	(0.7)	1.0
XnormI	double	0.5	(0.3)	0.7
Xnorm1	double	0.2	(0.1)	0.4
main→HPL_pdtest→HPL_pdgesv				
A	HPL_T_pmat *	136.6	(88.3)	2.9
main→HPL_pdtest→HPL_pdgesv→HPL_pdgesv0				
PANEL→L2	HPL_T_pmat	112.8	(72.9)	8.5
PANEL→A	double	12.8	(8.3)	3.8
PANEL→U	double	10.2	(6.6)	5.2

Conclusion

- Variable “blame” mapping
  - Switch analysis from delimited regions to variables
  - Used to represent abstractions in parallel frameworks, standard programs as well
  - Application Programmer Centric Analysis
- Target applications are large and parallel
  - Many levels of abstraction
  - Data structures map to mathematical constructs
- Future work
  - Create corpus of shared library transfer functions
  - Create GUI for data presentation
  - Evaluate system on larger programs

Further Information

- Europar 2009 paper
  - www.dyninst.org/papers/euroPar09.pdf
- SC'09 Demo
  - Booth 2449 (Dyninst and Paradyn Projects)