What it Takes to Assign Blame

Nick Rutar
Jeffrey K. Hollingsworth

University of Maryland
Parallel Framework Mapping

- **Traditional profiling represented as**
  - Functions, Basic Blocks, Statement

- **Frameworks have intuitive abstractions**
  - Direct ties with mathematical terms
  - PETSc, Cactus, POOMA, GrACE

- **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

```c
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);
    VecSet(u,one);
    MatMult(A,u,b);

    KSPCreate(PETSC_COMM_WORLD,&ksp);
    KSPGetPC(ksp,&pc);
    PCSetType(pc,PCJACOBI);

    ierr = KSPSolve(ksp,b,x); }
```
Variable “Blame”

- Record writes 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

Static Analysis

- Implicit/Explicit Data Flow Relationships

Runtime (Instances)

- Transfer Functions
  - Generation
  - Application

- Mem Containers
  - Stubs
  - Allocation/Free

- Container Resolution
  - Static
  - Dynamic

Variable Blame

- PAPI Hardware Counter Sampling
- DyninstAPI
- Symtab API
- StackWalker API
Preliminary Experimental Results

- Chose 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

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

- **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

University of Maryland
## HPL Results

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

Blame Points

**Dyn Inst**
Implementation Details

- Mixture of Static and Runtime Tools
- Static Analysis
  - LLVM
  - Boost
- Runtime Analysis
  - Dyninst API
  - Symtab API
  - Stackwalker API
  - PAPI
LLVM (Low Level Virtual Machine)

- **What is it?**
  - Compiler Infrastructure
  - Provides Intermediate Representation
    - Each instruction in SSA form

- **Why we use it?**
  - Need intermediate representation for static analysis
  - SSA form useful for creating dependency relationships
  - Intuitive API for accessing
    - Def-use chains
    - Dominator & CFG information
    - Language Independent representation of complex types
  - Integration with GCC
  - Multiple Language support
    - C, C++, Fortran

- **Limitations**
  - llvm-gcc versus gcc
Boost

● **What is it?**
  - Widely used portable C++ Libraries

● **Why we use it?**
  - Implicit/Explicit data flow relationships
    - Can create very large graphs
  - Boost provides graph libraries
    - Efficient representation of nodes/edges
      - Descriptors assigned to both
    - DFS, BFS, Uniform Cost Search
    - Dijkstra’s Shortest Path, Kruskal’s MST, ...

● **Limitations**
  - Trade efficiency for requiring one more library
StackWalker API

● What is it?
  - API for runtime traversing of stack

● Why we use it?
  - Instance Generation
    • Used in combination with PAPI
    • Each sample point we need full path information
    • Use full context given from PAPI
      - Walk up stack until we reach the top
  - Mem-Container Information
    • Used in combination with Dyninst
    • Wrapper functions mean we need full path
      - Every allocation we get full allocation path

● Limitations
  - Frame pointer removal decreases accuracy
DyninstAPI

- **What is it?**
  - Dynamic instrumentation tool

- **Why we use it?**
  - Need to instrument memory allocation sites
  - Integrated with StackWalkerAPI

- **Limitations**
  - Instrumentation overhead
SymtabAPI

● **What is it?**
  - API for accessing symbol information

● **Why we use it?**
  - General Module/Function Information
  - Line Number Mappings
    • Runtime information mapped back to source
    • Use line number mappings for this

● **Limitations**
  - Debugging Information needed
PAPI

- **What is it?**
  - API that provides interface to hardware counters

- **Why we use it?**
  - Instance (Sample Point) Generation
    - PAPI provides sampling interface
    - User chooses metric to trigger sample
      - Metrics can be any measurable event on system
      - PAPI hardware counters

- **Limitations**
  - Special kernel patch required on certain systems
## Advantage of Using Tools

<table>
<thead>
<tr>
<th>Application/API</th>
<th>LOC (w/comments)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blame</td>
<td>6K (8K)</td>
</tr>
<tr>
<td>Dyninst API 6.0</td>
<td>292K (360K)</td>
</tr>
<tr>
<td>Symtab API 6.0</td>
<td>51K (65K)</td>
</tr>
<tr>
<td>Stackwalker API 6.0</td>
<td>52K (66K)</td>
</tr>
<tr>
<td>LLVM 2.3</td>
<td>298K (375K)</td>
</tr>
<tr>
<td>PAPI 3.6</td>
<td>278K (320K)</td>
</tr>
<tr>
<td>Boost (Graph) 1.36</td>
<td>29K (33K)</td>
</tr>
</tbody>
</table>
Conclusion

- **Variable “blame” mapping**
  - Switch analysis from delimited regions to variables
  - Alternative to standard profiling techniques

- **Lessons Learned**
  - Standards are a good things
    - PAPI gives ucontext
    - Stackwalker uses information for context
  - Best not to reinvent the wheel ... BUT
  - Tool interoperability can be a problem
    - Compiler, OS compatibilities
    - Runtime tool interoperability
    - Target application/end-user requirements

- **Questions?**