Automated Timer Generation for Empirical Tuning

Josh Magee
Qing Yi
R. Clint Whaley

University of Texas at San Antonio
Propositions

- How do we measure success for tuning?
  - The performance of the tuned code --- of course
  - But what about tuning time?
    - **How long are the users willing to wait?** Given 3 more hours, how much can we improve program efficiency?

- Auto-tuning libraries have been successful and widely used
  - ATLAS, PHiPAC, FFTW, SPIRAL...
  - Critical routines are tuned because they are invoked many many times

- What happens when tuning whole applications?
  - What the end users need and what compilers expect to see
  - But applications are often extremely large and time consuming to run
  - **Do not want to rerun entire applications to try out different optimization configurations**
Observations

- Performance of full applications critically depend on a few computation/data intensive routines
  - These routines are often small but invoked a large number of times
  - Performance analysis tools (e.g., HPC toolkit) can be used to identify these routines
- Tuning these routines can significantly improve overall performance of whole applications while reducing tuning time
  - In some SPEC benchmarks, running the whole application is about 175K times longer than running a single critical routine
- The problem: setting up execution environment of the routines
  - A driver is required to set up parameters and global variables properly and accurately measure the runtime of each routine invocation
  - The cache and memory states of the machine is very important (Whaley and Castaldo, SPE’08)
  - NOT a trivial problem as one may think

**Overall goal: reduce tuning time without sacrificing tuning accuracy**
Empirical tuning approach

- **Instrumentation library**
  - Collect details of routine execution within whole applications
  - Invoked after HPC toolkit is used to identify critical routines

- **POET timer generator**
  - Input: routine specification + cache config + output config
  - Output: timing driver with accurately replicated execution environment
    - Support a checkpointing approach for routines operating on irregular data

- **Empirical tuning system**
  - Apply optimizations to produce different routine implementations
  - Link routine implementation with the timing driver and collect performance feedback
Replicating Environment of Routine Invocations

- **Goal**: ensure proper input values and operand workspaces
  - Reflect common usage patterns of routine
  - Should not result in abnormal evaluation

- **Data insensitive routines**
  - Amount of computation determined by integer parameters controlling problem size
  - Performance not noticeably affected by values stored in input
  - Example: dense matrix multiplication

- **Data sensitive routines**
  - Amount of computation depends on values and positioning of data
  - Examples: sorting algorithms, complex pointer-chasing algorithms

- **Replicating routine invocation environment**
  - For data insensitive routines: replicate problem size and use randomly generated values
  - For data sensitive routines: use the check-pointing approach
The Default Timing Approach
(for data-insensitive routines)

Routine specification for a
Matrix Multiplication kernel

```c
routine=void ATL_USERMM(const int M,
const int N, const int K,
const double alpha,
const double* A, const int lda,
const double* B, const int ldb,
const double beta,
double* C, const int ldc);

init={
    M=Macro(MS,72);
    N=Macro(NS,72);
    K=Macro(KS,72);
    lda=MS; ldb=KS; ldc=MS; alpha=1; beta=1;
    A=Matrix(double,M,K,RANDOM,flush|align(16));
    B=Matrix(double,K,N,RANDOM,flush|align(16));
    C=Matrix(double,M,N,RANDOM,flush|align(16));
} ;
flop="2*M*N*K+M*N";
```

Template of auto-generated timing driver

```c
for each routine parameter s in R do
    if s is a pointer or array variable then
        allocate memory for s
    end for
for each repetition of timing do
    for each routine parameter s in R do
        if s needs to be initialized then
            initialize memory_s
        end for
        if Cache flushing = true then then Flush Cache
            time_start <- current time
            call R
            time_end <- current time
            time_spent <- time_end - time_start
        end for
        Calculate min, max, and average of
time_spent
        if flps is defined then
            Calculate Max and average MFLOPS
        end if
Print All timings
```
The Checkpointing Approach (for data-sensitive routines)

- Checkpoint image includes
  - All the data in memory before calling enter_checkpoint
  - All the instructions between enter_checkpoint and stop_checkpoint

- Checkpoint image is saved to a file
  - Auto-generated timers can invoke the checkpoint image via a call to restart_checkpoint

- Utilize the Berkeley Lab Checkpoint/Restart (BLCR) library

- Delayed checkpointing
  - Call enter_checkpoint several instructions/loop iterations ahead of time to restore the cache state

```c
enter_checkpoint(CHECKPOINTING_IMAGE_NAME);

startup=GetWallTime();
retval = mainGtU(i1, i2, block, quadrant, nblock, budget);
endtime=GetWallTime();

startup=GetWallTime();

stop_checkpoint();
```
The POET Language

- Language for expressing parameterized program transformations
  - Parameterized code transformations and configuration space
    - Transformations controlled by tuning parameters
    - Configuration space: parameters and constraints on their values
  - Interpreted by search engine and transformation engine

- Language capabilities:
  - Able to parse/transform/output arbitrary languages
    - Have tried subsets of C/C++, Cobol, Java; going to add Fortran
  - Able to express arbitrary program transformations
    - Support optimizations by compilers or developers
    - Have implemented a large collection of compiler optimizations
    - Have achieved comparable performance to ATLAS(LCSD07)
  - Able to easily compose different transformations
    - Allow transformations to be defined easily reordered
    - Empirical tuning of transformation ordering (LCPC08)
  - Parameterization is built-in and well supported
Experimental Evaluation

- **Goal:** verify that POET-generated timers can
  - Significantly reduce tuning time for large applications
  - Accurately reproduce performance of the tuned routines

- **Methodology**
  - Compare POET-generated timers with the ATLAS timers
    - Using differently optimized gemm kernels by POET
  - Compare POET-generated timers with profiling results from running whole applications
    - For both data-insensitive and data-sensitive routines
    - Verify both the default timing approach and the checkpointing approach

- **Evaluation platforms**
  - Two multicore platforms: a 3.0Ghz Dual-Core AMD Opteron 2222 and a 3.0Ghz Quad-Core Intel Xeon Mac Pro.
  - Timings obtained in serial mode using a single core of each machine.
Reduction in tuning time

<table>
<thead>
<tr>
<th></th>
<th>Full application</th>
<th>Delayed checkpoint</th>
<th>Immediate checkpoint</th>
<th>Default timing via POET</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult_su3_mat_vec</td>
<td>877,430ms</td>
<td>3,502ms</td>
<td>3,510ms</td>
<td>5ms</td>
</tr>
<tr>
<td>mainGtU</td>
<td>45,765ms</td>
<td>2,019ms</td>
<td>1,975ms</td>
<td>4ms</td>
</tr>
<tr>
<td>scan_for_patterns</td>
<td>90,460ms</td>
<td>6,218ms</td>
<td>5,930ms</td>
<td>n/a</td>
</tr>
</tbody>
</table>
Comparing to ATLAS
Tuning Data-Insensitive Routine
Tuning Data-Sensitive Routine
Summary and Ongoing work

- **Goal:** reduce the tuning time of large scientific applications
  - Independently measure and tune the performance of critical routines
  - Accurately replicate the execution environment of routines

- **Solutions**
  - Libraries to profile and collect execution environment of critical routines
  - Use POET to automatically generate timing drivers
  - Immediate and delayed checkpointing approach

- **Ongoing work**
  - Reduce tuning time through the right search strategies
  - Automate the tuning process by integrating POET with advanced compiler technologies