Modeling UPC Fine-grain Access Performance

MTU

Zhang Zhang, Steve Seidel

Performance Model & Programming Model

- Modeling UPC performance is made difficult by the UPC programming model.
  - Implicit communication: weak correspondence between references and messages.
  - Fine-grain accesses: communication intermixes with computation.
  - Memory consistency model: aggressive code motion allowed, what you see isn't what you get.

Approach

- Application-level analytical performance model.
- Model UPC fine-grain access performance through platform benchmarking and code analysis.
- Platform benchmarking: determine compiler/RTS optimization abilities.
- Code analysis: dependence-based analysis, determine if a UPC code can benefit from optimizations supported by the platform.

Platform Abstraction

- Envision a common set of optimizations performed by a quality UPC platform:
  - Access aggregation
  - Access vectorization
  - Access pipelining
  - Local shared access optimization
  - Communication-computation overlapping

Platform Abstraction

- Design four microbenchmarks to determine a platform's potentials of conducting above optimizations.
  - *Baseline*: random remote shared accesses.
  - *Vector*: accesses to consecutive locations on a remote thread.
  - *Pipeline*: small but random-stride accesses to locations on a remote thread.
  - *Local*: local shared accesses using definite and indefinite block-sized pointers-to-shared.

Code Analysis

- Observation: high performance achievable by exploiting concurrency in shared references.
- Constraints for concurrent scheduling of shared accesses:
  - Dependences among references
  - Sequence points: fences, barriers, strict operations, library function calls, etc.
- Dependence analysis to identify these constraints.

Code Analysis

- Preliminaries:
  - Sequence points slice code into *intervals*.
  - Analysis is restricted to each interval.
  - Assume a flattened code structure (user defined functions inlined).
- Reference partitioning:
  - References are partitioned into groups. References in a group may be scheduled concurrently, i.e., they are subject to one type of optimization.

Code Analysis

- Reference partitioning: (cont'd)
  - Definition:

    A partition is a quadruple: *(C, pattern, name, affinity)*
    - *C* is the set of references in a partition.
    - *Pattern* is one of four patterns: baseline, vector, pipeline, local.
    - *Name* is the name of the shared object referenced by *C*.
    - *Affinity* is the thread with which references in C have affinity.
  - Construct a dependence graph for references within an interval.

Code Analysis

- Reference partitioning: (cont'd)
  - Use loop-independent dependence as a guide to find groups for vectorizable references.
  - Use the *Typed Fusion* algorithm to partition other shared references.
  - Typed Fusion: Fusing vertices of the same type that are not joined by bad edges into a group. In this case,
    - Vertices: references
    - Edges: dependences
    - Type: the affinity related to the references
    - Bad edges: true dependence and antidependence edges

Examples

```
shared [ ] TYPE *pSH;
// *pSH points to a remote
// location.
for (i = 0; i < M; i++)
{
  ..... = pSH[i];
  pSH[i] = ......;
}
```

Vectorizable, can be transformed into a vector read and a vector write.

```
#define BLOCK   (N*N/THREADS)
shared [BLOCK] TYPE A[N][N];
```

Pipelined remote accesses.

```
d1 = A[i-1][j-1] + A[i-1][j+1];

d2 = A[i][j-1]     + A[i][j+1];
d3 = A[i+1][j-1] + A[i+1][j+1];
```

Local shared accesses.

Performance Prediction

- Cost of shared accesses in an interval:

$$T_{shared}^{\text{interval}} = \sum_{i}^{\text{Re } fGroups} \left\{ \frac{N_i}{r(N_i, pattern)} \right\}$$

  - r is the effective data transfer rate of a pattern.
  - N_i is the total # of words accessed in a group.
- Define parallelism:

$$\frac{N_s}{N_p} = \frac{\#\text{ of mem ops in seq code}}{\#\text{ of mem ops per thread}}$$

Performance Prediction

- Speedup:

  N_c = # of private memory ops per thread

  N_r = # of remote shared memory ops per thread

  N_l = # of local shared memory ops per thread

  g_r = avg. gap between private and remote memory access latencies.

  g_l = avg. gap between private and local memory access latencies.

$$speedup = \frac{N_s}{N_c + N_r * g_r + N_l * g_l} \approx \frac{T_s}{T_{shared}} \leq \frac{N_s}{N_p}$$

Example: Impact of Compiler Optimization

- Berkeley UPC's experimental optimizations
  - Aggregation of remote accesses
  - Elimination of redundant accesses
  - Optimized shared pointer arithmetic
- Sobel edge detection
  - Local shared accesses are majority
  - Communication occurs only at border lines
  - Balanced workload across threads
- Try to model performance improvement due to compiler optimizations.

Example: Impact of Compiler Optimization

Threads	Measured Improvement (%)	Predicted improvement (%)
2	41	34
4	39	36
6	42	37
8	42	38
10	33	39
12	41	40