Motivation: Memory is the Bottleneck

NUMA: Non-Uniform Memory Access

core \hspace{1cm} \text{QuickPath} \hspace{1cm} core

\hspace{1cm} cache

\hspace{1cm} memory

core \hspace{1cm} core

\hspace{1cm} cache

\hspace{1cm} memory

local \hspace{1cm} remote
access \hspace{1cm} access
State of the Arts

- **simulation methods**
  - deep insights
  - **weaknesses:**
    - 2-5x overhead
    - not real machines
  - low overhead with deep insights

- **measurement methods**
  - deep insights with low overhead
  - low overhead
Hardware Address Sampling

- **Features of address sampling**
  - sample memory-related events (memory accesses, NUMA events)
  - capture effective addresses
  - record precise IP of sampled instructions or events

- **Support in modern processors**
  - AMD Opteron 10h and above: instruction-based sampling (IBS)
  - IBM POWER 5 and above: marked event sampling (MRK)
  - Intel Itanium 2: data event address register sampling (DEAR)
  - Intel Pentium 4 and above: precise event based sampling (PEBS)
  - Intel Nehalem and above: PEBS with load latency (PEBS-LL)

- **Efficient memory measurement (SC’13)**
  - code-centric analysis
  - data-centric analysis
Code-centric vs. Data-centric

- **Code-centric attribution**
  - problematic code sections
    - instruction, loop, function

- **Data-centric attribution**
  - problematic variables
    - static/heap variables

```c
#pragma omp parallel for num_threads(4)
for (i = 0; i < n; i++) {
    for(j = 0; j < n; j++) {
        for(k = 0; k < n; k++) {
        }
    }
}
```

- **Combining code-centric and data-centric attribution provides additional insight**
Attributing Samples

heap allocated variables

allocation path

malloc

data-centric attribution

static variable range

variable name

0x0

0xff

code-centric attribution
Aggregating Profiles

heap allocated variables

allocation path

malloc

heap allocated variables

allocation path

malloc

heap allocated variables

allocation path

malloc

merge
LULESH on Platform of 8 NUMA Domains

heap data: 68% remote accesses

z accounts for 7.7% remote accesses

z is allocated in a NUMA domain but accessed by others

interleave pages of z across NUMA nodes 13% improvement in running time

allocation call path

call site of allocation

call paths for accesses

Remote accesses

13% improvement in running time

Existing Measurement is Inadequate

- Data collection + attribution ≠ optimal optimization
  - know problematic data objects but not know why
  - need more insights for optimization guidance
- Challenges for address sampling
  - very sparse memory access samples
  - not monitoring continuous memory accesses
- Opportunities for address sampling
  - effective addresses: analyze memory access patterns
  - data sources: understand where inefficiencies come from
  - latency: derive new latency metrics to quantify inefficiencies.
Published work

- analyzing NUMA bottlenecks (PPoPP’14)
- guiding array regrouping for better locality (PACT’14)
- identifying memory scaling issues (SC’15)
Interleaved Allocation is NOT Always Best

Goal: identify the best data distribution for a program
Memory Access Pattern Analysis

- **Online data collection**

- **Offline analysis**
  - merge \([\text{min}, \text{max}]\) intervals along call paths
  - plot \([\text{min}, \text{max}]\) for each thread

- can be for any context, any variable

---

**array A**

allocate A blockwise to different domains

---

\([\text{min}, \text{max}]\) per sampled memory access

---

balanced allocation + maximum locality
LULESH on Platform of 8 NUMA Domains

- Block-wise allocation: 25% faster running time
- Interleaved allocation: 13% faster running time

- Call path allocates z
- Call paths access z

- z accounts for 7.7% of remote accesses
Beyond Data Collection and Attribution

• Published work
  – analyzing NUMA bottlenecks (PPoPP’14)
  – guiding array regrouping for better locality (PACT’14)
  – identifying memory scaling issues (SC’15)
Array Regrouping

for (i = 0; i < N; i++)
    \[ B[i] = A_1[i] + A_2[i] + \ldots + A_n[i]; \]

for (i = 0; i < N; i++)
    \[ B[i] = Arr[i].A_1 + Arr[i].A_2 + \ldots + Arr[i].A_n; \]

multiple prefetching streams

contention: conflict misses

only 1 stream

no contention
Workflow

ArrayTool

- filter out insignificant arrays
- compute array affinity
- analyze access patterns

 executable binaries

 compile

 source code

 programmers

 regrouping decision

 regroup
Filter out Insignificant Arrays

latency

\[
\begin{align*}
I &= (\text{float } *)\text{malloc}(\text{size}_I \times \text{sizeof(float)}); \\
J &= (\text{float } *)\text{malloc}(\text{size}_I \times \text{sizeof(float)}); \\
c &= (\text{float } *)\text{malloc}(\text{sizeof(float)} \times \text{size}_I);
\end{align*}
\]

\[
\begin{align*}
iN &= (\text{int } *)\text{malloc}(\text{sizeof(unsigned int)} \times \text{rows}); \\
iS &= (\text{int } *)\text{malloc}(\text{sizeof(unsigned int)} \times \text{rows}); \\
jW &= (\text{int } *)\text{malloc}(\text{sizeof(unsigned int)} \times \text{cols}); \\
jE &= (\text{int } *)\text{malloc}(\text{sizeof(unsigned int)} \times \text{cols});
\end{align*}
\]

\[
\begin{align*}
dN &= (\text{float } *)\text{malloc}(\text{sizeof(float)} \times \text{size}_I); \\
dS &= (\text{float } *)\text{malloc}(\text{sizeof(float)} \times \text{size}_I); \\
dW &= (\text{float } *)\text{malloc}(\text{sizeof(float)} \times \text{size}_I); \\
dE &= (\text{float } *)\text{malloc}(\text{sizeof(float)} \times \text{size}_I);
\end{align*}
\]

SRAD from Rodinia
Latency-based Array Affinity

loops in their calling contexts

A mixed CCT

A simplified mixed CCT

Prune CCT leaf nodes: loops

aggregate latency of uncommon loops

aggregate latency of all loops

\[
R_{cw} = 1 - \frac{W_{uncommon}}{W_{total}}
\]
An Example
Access Pattern Analysis

arrays c’s access pattern

array interval touched by each thread

dN, dS, dE, dW have the same pattern
Regrouping Results of SRAD

```c
#pragma omp parallel for
for(int i=0; i<rows; i++) {
    for(int j=0; j<cols; j++) {
        k = i*cols+j;
        cN = c[k];
        cS = c[iS[i]*cols+j];
        cW = c[k];
        cE = c[i*cols+jE[i]];
        D = cN*dN[k]+cS*dS[k]+cW*dW[k]+cE*dE[k];
        J[k] = J[k]+0.25*lambda*D;
    }
}
```

only regroup dN, dS, dW, dE $\uparrow 1.47x$

regroup dN, dS, dW, dE and c $\uparrow 1.89x$
• Published work
  – analyzing NUMA bottlenecks (PPoPP’14)
  – guiding array regrouping for better locality (PACT’14)
  – identifying memory scaling issues (SC’15)
Scaling Losses in Memory Hierarchies

- Memory contentions hurt scalability: cache/bandwidth contention
  - which data objects contribute to the most scaling losses
  - which memory layers incur the most scaling losses

- Methods
  - decompose latency according to data objects and memory layers
    - data-centric analysis with data source information
    - differential analysis supported by HPCToolkit
      - compare profiles between different runs

more details in SC’15
Conclusions and Future Work

- **Hardware address sampling**
  - widely supported in modern architectures
  - powerful in monitoring memory behaviors
  - more analysis of the samples provides more performance insights

- **On-going work**
  - structure splitting
  - locality optimization between SMT threads
  - cache line false sharing
  - automatic page migration for NUMA architectures

- **Future directions of address sampling**
  - comparing different address sampling mechanisms
  - analyzing new performance issues
    - heterogeneous memory: 3D stack memory
UMT2013 on Quad-socket POWER7 Node

- Sample off-chip accesses
- self%STime
- 18.2% of remote accesses allocated in one domain accessed by everyone
Optimize self%STime for UMT2013

address-centric analysis for self%STime

self%STime’s address space

optimization: let each thread initialize its own data
result: all threads have data locally -- 7% faster