Data Distribution in Practice: A Case for User-Level Page Migration

- Targets NUMA machines
  - SGI Origin
  - To find a data element:
    - First, look in local cache
    - Then, look in local main memory
    - Finally, locate data and fetch remotely
  - Ratio 1:3:5 common

Why do I care?

- "The Origin was 15 years ago!"
  - True, but with the explosion of core counts, the only way to scale on multicore machines is to provide a NUMA programming model
  - AMD, Intel build NUMA multicore nodes
    - Opteron: three different memory access times: local memory, one hop away, two hops away (memory modules are not fully connected)

Key Question in this Paper

- Can we program NUMA machines without regard to locality, and still obtain efficiency
- Many people believe the answer is *no*

Problem Statement

- Given a program, consider a single data element
  - Suppose that this data element is accessed by core *i* a number of times equal to C_i
  - The goal is to place this data element in the memory module that will minimize the number of remote references; i.e., choose *i* such that C_i is largest
    - Note: page migration might be allowed, which means that elements can move between cores

Dynamic Page Migration

- Basic idea:
  - Keep a reference counter per virtual page
  - Move page when it is being referenced more remotely than locally
    - Either via interrupts or sampling
      - Interrupts notifies when counters are "interesting"; sampling takes measurements every X time units
    - Not based on program semantics
    - Ends up with lots of heuristics (**read: hacks**): damping thresholds, freezing thresholds, unfreezing thresholds

In general: kernel has issues with effectiveness, timeliness, and flexibility

Paper by Nikolopoulos et al.

- Basic idea:
  - Combine compiler and run-time system
  - Compiler identifies “hot spots”
  - Run-time system monitors reference counts
  - Migrate only at end of *phase*

Hot memory areas

```
for t = 1 to timesteps {
   #pragma omp parallel for
  for i = 1 to n
    A[i] = B[i+1] + C[i]
  #pragma omp parallel for
  for i = 1 to n
    B[i] = A[i]
}
```

Decisions made here only

A, B considered “hot areas”

Competitive Criterion

- Should a page be migrated?
- Quantities we care about include:
  - Total latency of remote accesses from each node
  - Total latency of local access
  - Ratio of remote access latency to local access latency
  - Cost of migrating

$$ral_{tot}(i,h) > (r_u(i,h)/l_u) \cdot lal_{tot} + ml \qquad (1)$$

(This is done for each node *i*; home node [current location] is *h*)

Problems with this? (Think: migration latency)

Competitive Criterion, continued

$$ral_{tot}(i,h) > (r_u(i,h)/l_u) \cdot lal_{tot} + ml \qquad (1)$$

- How do we obtain these quantities?
- Obtain memory access time from chip spec
  - Add in contention;
- Obtain number of accesses from page counters
- Obtain migration latency from microbenchmarks

Problems with this? (Think: contention)

Predictive Criterion

- OS might migrate threads to different cores (that have different local memory modules)
- Goal: eagerly move pages to threads
- Idea: migration results in (assuming the page is properly placed) an inversion in local/remote accesses between iterations

$$\exists i, racc(i,t) > racc(i,t-1) \wedge lacc(t) < lacc(t-1) \quad (3)$$

Ping-Pong and other Hacks

- Ping-pong occurs if a page bounces back and forth between nodes
- Detect and freeze pages
- Also can mark pages as cold

Note: this is moving towards general dynamic page migration!

Problems with this? (Think: slippery slope)

Overall Algorithm

```
(1) for each hot memory area {
(2)   identify candidate pages for migration with either the competitive
(3)   or the predictive criterion;
(4)   if the predictive criterion is used {
(5)     apply ping-pong prevention algorithm;
(6)   }
(7)   for each candidate page {
(8)     if the page must not be frozen and is not likely to ping-pong {
(9)       migrate it;
(10)      update page bookkeeping data;
(11)    }
(12)  }
(13)  estimate the maximum memory latency due to remote accesses in this area;
(14)  if this latency increases compared to previous invocations {
(15)    tune the selectiveness of the migration algorithm;
(16)  }
(17)  if the area had no candidate pages for migration  {
(18)    if the area had no candidates in several previous invocations  {
(19)      mark the area as cold;
(20)    }
(21)  }
(22)}
```

Figure 1: Pseudocode for the page migration algorithm.

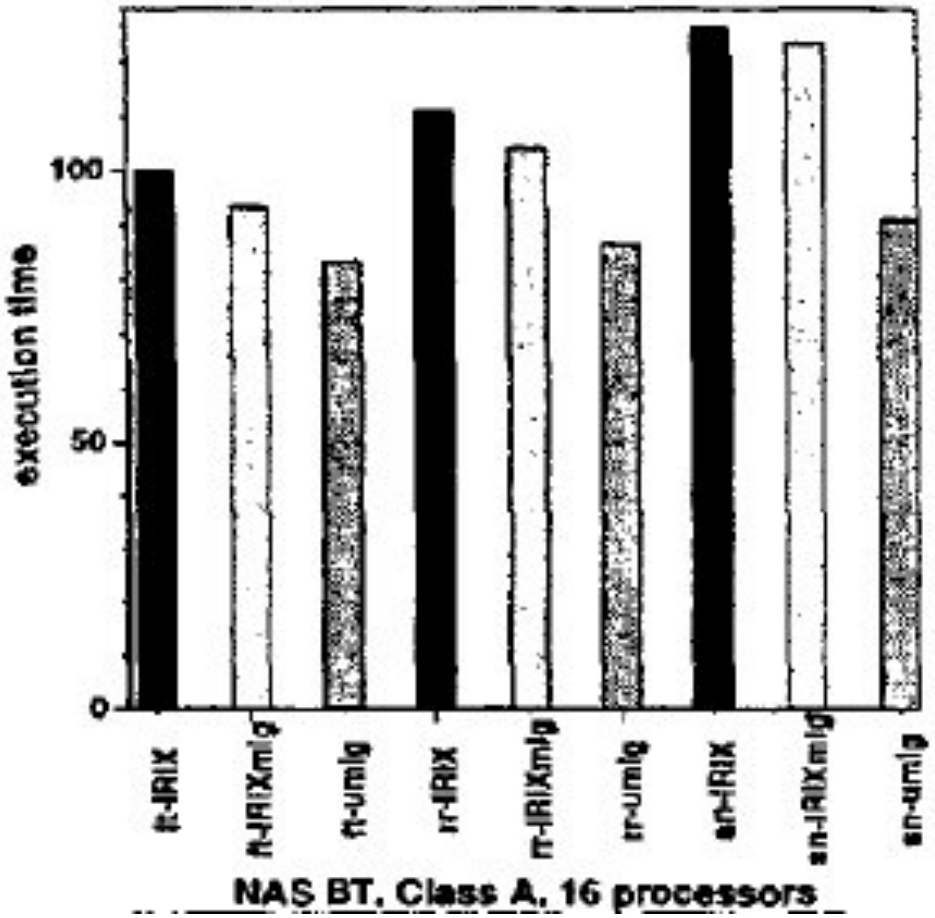
Problems with this? (Think: relationship to specified algorithm?)

Implementation

- Get reference counters through “/proc” filesystem
- Use *mmci* to migrate pages
- Use *schedcntl* to figure out on which processors threads are executing

Important: does not require OS modification

Sample Results



Problems with this? (Think: what is "good"?)

Implications

- Can the user program without regard to locality?
  - Sort of (in some cases)
- Recent operating systems allow explicit control of memory
  - *numactl* allows memory to be allocated from any given memory module
  - Without this, some Linux versions allocate arbitrarily
  - Doesn't handle the problem of thread migration
    - But user calls also allow binding of threads

Broader Issues (with this line of work/thinking)

- Overemphasis on placing pages where fewest remote accesses occur
  - While important, this paper ignores the possibility of a per-phase optimal distribution, as well as choosing suboptimal distributions in each phase
    - See the data distribution slide deck for reference