ECE7995
(6) Process Scheduling
The Position of Process Scheduling in the I/O Scheduler

- A process is the producer and consumer of I/O requests.
- The process scheduling has a control of which requests to be issued and which data to be used.
  - Effect on working set size;
  - Effect on I/O efficiency.
Outline

• Alleviate thrashing for virtual memory by reducing working set size.
  ➢ Explicitly prioritizing process executions
  ➢ Using token to implicitly serialize process execution

• Coordinating execution of a process with the service of its I/O requests for I/O efficiency
What Thrashing in Linux We Target?

- Multiprogramming environment
- Memory shortage spread over processes
- Each process has lots of page faults
- Very low CPU utilization
Existing Schemes and Our Approach

- Local replacement;
- Kill some processes (e.g. Linux);
- Load control (e.g. BSD);

We address the problem by adjusting page replacement.
TPF: Prioritize Process Execution for Elimination of Thrashing

How thrashing develops in the kernel?
Analysis of page replacement variations in Linux;
How our Thrashing Protection Facility (TPF) works?
Performance evaluation
Thrashing

Definition

• In virtual memory systems, thrashing may be caused by programs that present insufficient locality of reference: if the working set of a program cannot be effectively held within physical memory, then constant data swapping may cause thrashing.

Questions?

How thrashing develops in the kernel?
How to deal with thrashing?
How thrashing develops in the kernel?

Global LRU replacement policy is usually used to allocate memory and coordinate use of memory among processes.

- No process is able to establish its working set
- Each process has large number of page faults
- Low CPU utilization
- All processes make little progress.

Memory demand

Proc1

Proc2

Physical memory

CPU IDLE

paging

paging
The performance degradation under thrashing

- Memory shortage 42%.

**Dedicated Executions**

- The time of first spike is extended by 70 times.

**Concurrent Executions**

- The time of a start of vortex is extended by 15 times.
Factors Related to Thrashing

The size of memory space in the system;
The number of processes;
The dynamic memory demands;
The page replacement scheme.
Outline

How thrashing develops in the kernel?
Analysis of page replacement variations in Linux;
How our Thrashing Protection Facility (TPF) works?
Performance evaluation
Conclusion
Framework of Linux Page Replacement

Searching of an NRU (Not Recently Used) page starts from where it was done last time in a process by process, and page by page fashion.

(1) Select a swappable process to find NRU pages;
(2) Check through the virtual memory pages in the selected process; if not find NRU pages, go to (1) for next process.
Two Aspects Related to Thrashing

How many NRU pages in a selected process are allowed to replace continuously?
How easily NRU pages can be generated?

- Allow a large amount of pages to be replaced from a specific process once a time
- Prepare enough NRU pages for eviction
- Help others to build up their working set, and reduce thrashing possibility
- Memory shortage concentrate on one or a few specific processes
Replacement in Kernel 2.0

NRU page contributions are distributed;
NRU pages are generated by aging (when age is 0).

During examining the reference bit of each page, a set bit (1) cause age increased by 3 with a limit of 20, an unset bit (0) cause age decreased by 3.

Encourages spreading the memory shortage burden over processes, so that no one can build up its working set.
Let p be current swappable process

\[ p->swap\_cnt == 0? \]

\[ Y \]

\[ p->swap\_cnt = RSS/MB \]

\[ p->swap\_cnt -- \]

\[ p->swap\_cnt == 0? \]

\[ N \]

Ready to go to next proc p

Try to find an NRU page in p

NRU page found?

\[ Y \]

\[ Y \]

succeed

Ready to go to next proc

--count>0?

\[ Y \]

fail

\[ N \]
A selected process continuously contribute its NRU pages;

No page aging any more (only using reference bit, easier to produce an NRU page)

Penalize the memory usage of one process at a time. Thus others have more chances to build up their working sets.
For each process p:
\[ p->\text{swap\_cnt} = p->\text{RSS} \]

Find the process \( p_{\text{best}} \) with maximum \( p->\text{swap\_cnt} \)

Try to find an NRU page in \( p_{\text{best}} \)

NRU page found?

\[ \text{NRU page found?} \]

\[ \text{Y} \]

\[ \text{succeed} \]

\[ \text{N} \]

\[ p_{\text{best}}->\text{swap\_cnt} = 0 \]

\[ \text{For all process P} \]
\[ p->\text{swap\_cnt} = 0 ? \]

\[ \text{Y} \]

\[ \text{fail} \]

\[ \text{N} \]

--count>0?

\[ \text{N} \]

\[ \text{succeed} \]

\[ \text{Y} \]

Kernel 2.2
Replacement in Kernel 2.4

Addressing concerns on memory performance of Kernel 2.2 by re-introducing age

proportional NRU page distribution;
Let P be the next process

Walk about 6% of the address space of p

--count>0?

Y

N
done

Kernel 2.4
### Summary of the Replacement Behavior during Thrashing

<table>
<thead>
<tr>
<th>Kernels</th>
<th>2.0</th>
<th>2.2</th>
<th>2.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keep CPU busy</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Keep memory well utilized</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Why an adaptive policy is needed for thrashing protection?

Conflicting interests in the design:

- Regarding CPU utilization, keep at least one process active
- Regarding memory utilization, apply the LRU principle consistently to all the processes.
The Goal of the Adaptive Solution

- When CPU utilization is not a concern, make memory resource be efficiently used.
- When CPU utilization is low due to thrashing, change replacement behavior adaptively.
Outline

How thrashing develops in the kernel?
Analysis of page replacement variations in Linux;
How the Thrashing Protection Facility (TPF) works?
Performance evaluation
Conclusion
Basic Idea of Thrashing Protection Facility (TPF)

- Multiple “CPU cycle hungry” processes but with high page fault rates
- Low CPU utilization

Temporal changing page replacement policy to help specific process build up its working set

CPU utilization increased

Return to normal page replacement
Parameters of TPF

**CPU_Low**: the lowest CPU utilization the system can tolerate.

**CPU_High**: the targeted CPU utilization for TPF to achieve.

**PF_High**: the page fault rate threshold for a process to potentially cause thrashing.

**PF_Low**: the targeted page fault rate of the identified process for TPF to achieve.

In addition, the list “**high_PF_proc**” records processes with high page fault rates.
Three States in TPF

※ **Normal state**: Keep track of the page fault rate for each process and place the processes with rates higher than `PF_High` into list `high_PF_proc`.

※ **Monitoring state**: Monitor the CPU utilization and the page fault rates of processes in the list `high_PF_proc`. Select the “least memory hungry” process in the `high_PF_proc` for protection when CPU utilization is low.

※ **Protection state**: Mark the selected process and let its `swap_cnt` reset to 0 no matter whether a replaced page has been successfully found (in Kernel 2.2). This lets the process contribute at most one page continuously and help it quickly establish its working set.
TPF state transition

Normal State
- Length(high_PF_proc)>1

Monitoring State
- Length(high_PF_proc)<=1
- Length(high_PF_proc)<=1
- Page fault rate of protected proc<PF_Low
- CPU utilization>CPU_High

Protection State
- CPU utilization>CPU_Low and
- length(high_PF_proc)>=2
Outline

How thrashing develops in the kernel?
Analysis of page replacement variations in Linux;
How our Thrashing Protection Facility (TPF) works?
Performance evaluation
Conclusion
# Characterizations of Workloads

<table>
<thead>
<tr>
<th>Programs</th>
<th>Description</th>
<th>Memory Req. (MB)</th>
<th>Exec. Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit-r</td>
<td>Bit-reversal</td>
<td>130.0</td>
<td>326.1</td>
</tr>
<tr>
<td>LU</td>
<td>LU decomposition</td>
<td>162.0</td>
<td>99.2</td>
</tr>
<tr>
<td>gcc</td>
<td>Optimized C compiler from SPEC2000</td>
<td>145.0</td>
<td>218.7</td>
</tr>
<tr>
<td>vortex</td>
<td>Database application from SPEC2000</td>
<td>131.2</td>
<td>398.0</td>
</tr>
</tbody>
</table>
**Experiment settings**

Pentium II of 400 MHz

Red Hat Linux release 6.1 with Kernel 2.2.14.

The predetermined threshold values are set as follows: \texttt{CPU\_Low} = 40\%, \texttt{CPU\_High} = 80\%, \texttt{PF\_High} = 10 faults/second, \texttt{PF\_Low} = 1 fault/second.

We instrumented the kernel to adjust the available user memory so that different memory constraints can be formed to facilitate our experiments.
Time-space figures of dedicated execution

**X axis**: execution time

**Y axis**: number of pages

**MAD**: the number of pages demanded

**RSS**: the number of resident pages

**bit-r**

**LU**
Time-space figures of dedicated execution

gcc
vortex
Comparison for gcc+vortex (42% memory shortage)

W/o TPF

W/ TPF
Comparison for gcc+bit-r (31% memory shortage)
Comparison for LU1+LU2 (35% memory shortage)

W/O TPF

W/ TPF
Comparison of Execution Time
Comparisons of Numbers of Page Faults

The bar chart compares the number of page faults without and with TPF for various benchmarks.

- gcc
- vortex
- LU1
- LU2
- bit-t

The chart shows a significant reduction in page faults with TPF for each benchmark.
Comparison of Total Execution Time

<table>
<thead>
<tr>
<th></th>
<th>Exec. Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc + vortex</td>
<td>1500 (without TPF)</td>
</tr>
<tr>
<td></td>
<td>600 (with TPF)</td>
</tr>
<tr>
<td></td>
<td>300 (ideal)</td>
</tr>
<tr>
<td>gcc + bit-r</td>
<td>900 (without TPF)</td>
</tr>
<tr>
<td></td>
<td>600 (with TPF)</td>
</tr>
<tr>
<td></td>
<td>300 (ideal)</td>
</tr>
<tr>
<td>LU1 + LU2</td>
<td>300 (without TPF)</td>
</tr>
<tr>
<td></td>
<td>200 (with TPF)</td>
</tr>
<tr>
<td></td>
<td>100 (ideal)</td>
</tr>
</tbody>
</table>
Summary of TPF

Thrashing can be easily triggered by:

(1) Dynamical memory usage,

(2) Common memory reference patterns, and

(3) Serious memory shortage.

TPF is highly responsive to stop thrashing triggered by (1) and (2);
TPF is not intrusive in the multiprogramming environment;
Load control will be used only when it is truly necessary.
Using token to implicitly serialize process execution

• A page frame of a process becomes a replacement candidate in the LRU algorithm if the page has not been used for a certain period of time.

• There are two conditions under which a page is not accessed by its owner process:
  1) the process does not need to access the page;
  2) the process is conducting page faults (sleeping) so that it is not able to access the page although it might have done so without the page faults.

• We call the LRU pages generated on the first condition true LRU pages, and those on the second condition false LRU pages.

• These false LRU pages are produced by the time delay of page faults, not by the access delay of the program.
  ➢ The LRU principle is not maintained. (Temporal locality is not applicable!)

• False LRU pages can cause thrashing. However, global LRU page replacement implementations do not discriminate between these two types of LRU pages, and treats them equally!
Challenges

• How to distinguish two kinds of page faults?

• How to design and implement a lightweight thrashing prevention policy?
Algorithm Design

Why token?

Processes → Taken by one process → False LRU Page Faults for Token process

Other Processes → True/False LRU Page Faults → I/O
Algorithm Design

• The swap-token algorithm
  - Set a token in the system.
  - The token is taken by one of the processes when page faults occur.
  - The system eliminates the false LRU pages from the process holding the token to allow it to quickly establish its working set.
  - The token process is expected to complete its execution and release its allocated memory as well as its token.
  - Other processes then compete for the token and complete their runs in turn.

  • By transferring privilege among processes in thrashing from one to another, the system can reduce the total number of false LRU pages and to transform the chaotic order of page usages to an arranged order.

  • The policy can be designed to allow token transferred more intelligently among processes to address issues such as fairness and starvation.
Considerations on Swap-Token

• Which process to receive the token?
  ➢ A process whose memory allocation is the closest to its working set.

• How long does a process hold the token?
  ➢ It should be adjustable based on the seriousness of the thrashing.
  ➢ Currently a taken is relinquished only when the token-owner process is done.
  ➢ Can multiple tokens be used effectively for thrashing?

The swap token algorithm has been adopted in the Linux kernels.
## Performance Evaluation

### Benchmark programs

Table 1
Execution performance and memory related data of the 10 benchmark programs, where the program names with * are SPEC 2000 benchmarks

<table>
<thead>
<tr>
<th>Programs</th>
<th>Description</th>
<th>Input file/size</th>
<th>Max MAD (MB)</th>
<th>Lifetime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>*apsi</td>
<td>Climate modeling</td>
<td>apsi.in</td>
<td>196.0</td>
<td>2628.3</td>
</tr>
<tr>
<td>*gcc</td>
<td>Optimized C compiler</td>
<td>166.i</td>
<td>145.0</td>
<td>218.7</td>
</tr>
<tr>
<td>*gzip</td>
<td>Data compression</td>
<td>input.graphic</td>
<td>197.4</td>
<td>248.7</td>
</tr>
<tr>
<td>*mcf</td>
<td>Combinatorial optimization</td>
<td>inp.in</td>
<td>79.2</td>
<td>975.9</td>
</tr>
<tr>
<td>*vortex</td>
<td>Database</td>
<td>lendian1.raw</td>
<td>115.0</td>
<td>342.3</td>
</tr>
<tr>
<td>*vortex</td>
<td>Database</td>
<td>lendian3.raw</td>
<td>131.2</td>
<td>398.0</td>
</tr>
<tr>
<td>bit-r</td>
<td>Data reordering</td>
<td>$2^{25}$</td>
<td>131.3</td>
<td>326.1</td>
</tr>
<tr>
<td>m-m</td>
<td>Matrix multiplication</td>
<td>$1800^2$</td>
<td>76.2</td>
<td>1430.3</td>
</tr>
<tr>
<td>m-sort</td>
<td>Merge sort</td>
<td>$2^{24}$</td>
<td>131.4</td>
<td>58.3</td>
</tr>
<tr>
<td>LU</td>
<td>LU decomposition</td>
<td>$2000^2$</td>
<td>161.2</td>
<td>98.0</td>
</tr>
<tr>
<td>r-wing</td>
<td>Volume rendering</td>
<td>500,000</td>
<td>48.9</td>
<td>60.8</td>
</tr>
</tbody>
</table>
Experiment Results

w/o token

At the execution time of 397th second, the token was taken by gcc.

w/ Token

Fig. 8. The memory performance of gcc (left) and vortex3 (right) during the interactions.

Fig. 14. The memory performance of gcc (left) and vortex3 (right) during the interactions managed by the token-ordered LRU.

<table>
<thead>
<tr>
<th>Program</th>
<th>gcc</th>
<th>vortex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow Down Not Token</td>
<td>5.61</td>
<td>3.37</td>
</tr>
<tr>
<td>Slow Down Token</td>
<td>1.85</td>
<td>1.54</td>
</tr>
<tr>
<td>Slowdown reduction</td>
<td>67%</td>
<td>54%</td>
</tr>
<tr>
<td>Page fault reduction</td>
<td>95%</td>
<td>79%</td>
</tr>
</tbody>
</table>
iHarmonizer: Coordinating execution of a process with the service of its I/O requests for I/O efficiency

- Background of Multithreaded Programming
- Motivation Example
- Design and Implementation
- Performance Evaluation
- Conclusion
Background

A thread is a small unit of processing that can be scheduled by the operating system.

Threads allow multiple streams of program control flow to coexist within a process. They share resources like memory and file handles, but each thread has its own program counter, stack and local variables.

Why multithreading?

- Take advantage of multicore processors.
- Context switch between threads is less expensive than processes.
- The program is easier to write.
## Background

<table>
<thead>
<tr>
<th>Sequential Program</th>
<th>Multithreaded Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can only utilize a single processor</td>
<td>Can take advantage of multiprocessor hardware</td>
</tr>
<tr>
<td>Behavior is usually deterministic</td>
<td>Behavior is usually nondeterministic</td>
</tr>
<tr>
<td>Program is easier to understand</td>
<td>Program is tricky, threads racing for shared resources causes the most problematic bugs</td>
</tr>
<tr>
<td>Testing that covers all codes find most of bugs</td>
<td>Hard to find bugs or even reproduce the defect</td>
</tr>
</tbody>
</table>
Existing Efforts on Multithreading

Coordinate the execution of threads for correctness and processor performance

- semaphore, mutex, barrier, (synchronization, monitor)
- conditional variables (spin lock)
- Thread scheduling for load balance

Thread coordination for I/O can be critical!

- what if multiple threads make I/O access pattern worse?
Moltivation

```c
int main() {
    ......
    for (i = 0; i < num_of_cols; i++) {
        sum[i] = 0;
        for (j = 0; j < num_of_rows; ++j) {
            off_t offset = (j * num_of_cols + i) * BLOCK_SIZE;
            sum[i] += array[off_t].data;
        }
    }
}
```

```c
int main() {
    ......
    for (i = 0; i < num_of_cols; i++) {
        sum[i] = 0;
        ret = pthread_create(&t[i], NULL, calc_sum, (void*) i);
    }
    for (i = 0; i < num_of_cols; i++) {
        ret = pthread_join(t[i], NULL);
    }
}
```

```c
void* calc_sum(void* column) {
    int col = (int) column;
    for (i = 0; i < num_of_rows; ++i) {
        off_t offset = (i * num_of_cols + col) * BLOCK_SIZE;
        sum[col] += array[offset].data;
    }
}
```

8000x16 elements, 2GB
calc the sum of
each column
(a) A large range of req are accessed within short period of time indicating disk heads are moving back and forth. (544 seconds)

(b) Threads are making progress in almost the same speed, but data accessed by different threads are still in a relatively small range. (139.6 seconds)

(c) Pure sequential access. (45.7 seconds)
Where is the problem?

- When sequential program is multithreaded, computations as well as I/O operations are distributed into different threads.

- As the execution of threads are nondeterministic at runtime, the order to issue I/O requests is also nondeterministic, or random.
Why not just use the sequential program?
  • might need more computational resources
  • cannot always be sequential

Predict the I/O requests from each thread and efficiently retrieve these data.
  • requirement: understand each thread's data usage

Coordinate running threads so that they are driven by the already retrieved data.
  • requirement: able to suspend/resume threads according to their data usage.
Design

Use pre-execution to generate *Iteration Map*

- records the mapping of loop iterations to their requested data in terms of page number
- we only record the shared files by threads
- instrument compiler to insert codes for generating the iteration map
  - copy the original loop nest
  - strip out unrelated statements

<iteration number -> \{page numbers\}>

*Hash Map*
Design

A helper thread called *calculation thread* is spawned

- examine the loop iterations distributed for each thread
- add page numbers to a prefetch queue

Calculate *Frontline* to coordinate thread execution

- what pages to prefetch
- how far each thread should run before suspended

Each thread can proceed to the frontline without sending I/O requests to the disk
Design

Temporary frontline after step 1

Frontline after free ride

Thread 0
A: 0, 1
B: 11
A: 2, 3
B: 12
A: 3, 5
B: 17
A: 2, 3
B: 0, 3
A: 3, 5
B: 0, 2
A: 4, 6
B: 0, 1

Thread 1
A: 0, 2
B: 2, 3
A: 4, 8
B: 12
A: 8, 12
B: 5
A: 1, 12
B: 3
A: 11, 13
B: 0, 1
A: 10, 12
B: 0, 2

Thread 2
A: 0, 3
B: 0
A: 1, 4
B: 11
A: 2, 5
B: 3
A: 3, 12
B: 0
A: 14, 16
B: 0, 1
A: 9, 11
B: 0, 3

A: 0, 1 : iteration with its requested page numbers in file-mapped array A

prefetch queue size: 15,
A: 0, 1, 2, 3, 4, 5, 8, 12; B: 0, 2, 3, 5, 11, 12, 17
Design

Prefetching data in the prefetch queue

- Another helping thread *prefetch thread* is spawned
- when the prefetching queue is filled, this thread sorts the page numbers for efficient disk access
- Sorted and merged I/O requests are sent to kernel to read data
- *prefetch thread* and *calculation thread* work in parallel
Design

1. threads are suspended
2. calculate current frontline
3. prefetch data
4. calculate next frontline
5. threads computation
6. prefetch data
7. calculate next frontline
8. threads computation

- OpenMP Threads
- Calculation Thread
- Prefetching Thread
OpenMP Programming Model

Shared address space programming model

Easily insert directives to parallelize sequential program

Scheduling could be chosen from static or dynamic
Compiler Code Insertion

```c
int main() {
    #pragma omp parallel shared(matrix_a, matrix_b, matrix_c) private(tid, i, j, k)
    #pragma omp for
    for (i = 0; i < size; i+=1) {
        for (j = 0; j < size; j+=1) {
            for (k = 0; k < size; k+=1) {
                matrix_c[i*size+j].data += matrix_a[i*size+k].data * matrix_b[k*size+j].data;
            }
        }
    }
}
```

(a)
Compiler Code Insertion

```c
if(__ompv_ok_to_fork)
    __ompc_fork(0, &__ompregion_main1, &reg3);
}

static void __ompregion_main1(__INT32 __ompv_gtid_a, __UINT32 __ompv_slink_a) {
    ...
    __ompc_static_init_4(__ompv_temp_gtid, 1, &__ompv_temp_do_lower,
    &__ompv_temp_do_upper, &__ompv_temp_do_stride, 1, 1);
    while(__ompv_temp_do_lower <= temp_limit) {
        for(__mplocal_i = __ompv_temp_do_lower; __mplocal_i <= __ompv_temp_do_upper; 
            __mplocal_i = __mplocal_i + (__INT32)(1U)) {
            while(size > j) {
                while(size > k) {
                    ((matrix_c + (__UINT32)(j + (__mplocal_i * size))) -> data = 
                        (__UINT32)( *(matrix_c + (__UINT32)(j +(__mplocal_i * size))) + 
                        (__UINT32)( *(matrix_a + (__UINT32)(k + 
                        (__mplocal_i * size))) * (__UINT32)( *(matrix_b + (__UINT32)(j + (size * k)))))
                }
            ...
        }
    }
}
```
Compiler Code Insertion

```c
if (__ompv_ok_to_fork)
{
    ...
    for(i = 0U; i < size; i = i + (n_INT32)(1U))
    {
        for(j = 0U; j < size; j = j + (n_INT32)(1U))
        {
            for(k = 0U; k < size; k = k + (n_INT32)(1U))
            {
                __iharmonizer_io_generate_iteration(3, i, j, k);
                __iharmonizer_io_generate_offset((n_UINT32)matrix_a, k1 + (i * size), 4096U, 3, i, j, k1);
                __iharmonizer_io_generate_offset((n_UINT32)matrix_b, j1 + (k1 * size), 4096U, 3, i, j, k1);
            }
        }
    }
    __ompc_fence(0, &__ompregion_main1, &reg3);
}

static void __ompregion_main1(n_INT32 __ompv_tid_a, n_UINT32 __ompv_slink_a) {
    ...
    __omp_static_init_4(__ompv_temp_gtid, 1, &__ompv_temp_do_lower, ..);
    while(__ompv_temp_do_lower <= temp_limit) {
        for(__mplocal_i = __ompv_temp_do_lower; __mplocal_i <= __ompv_temp_do_upper; __mplocal_i =
            __mplocal_i + (n_INT32)(1U))
        {
            while(size > j) {
                while(size > k) {
                    __iharmonizer_io_break(__ompv_temp_gtid, 3, __mplocal_i, __mplocal_i, __mplocal_i, __mplocal_k);
                    ((matrix_c + (__INT32)(j + (__mplocal_i * size))))->data =
                        ((matrix_c + (__INT32)((j + (__mplocal_i * size))))) +
                        ((matrix_a + (__INT32)((i + (__mplocal_i * size))))) *
                        ((matrix_b + (__INT32)((i + (k + (__mplocal_i * size))))));
                }
            }
        }
    }
    ...
}
```
Evaluation

Experiment Setting:

- Linux 2.6.18
- I/O scheduler: CFQ
- Open64 4.2.1
- OpenMP 2.5
- Dell PowerEdge 1950 Server
  - Two Quad-Core Intel 3GHz Xeon 5100 CPU
  - 8 GB memory
  - 7200 RPM SATA hard disk
# Evaluation

## Benchmarks:

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Input File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM</td>
<td>Perform multiplication of two matrices</td>
<td>2 x 1.3GB</td>
</tr>
<tr>
<td>Hist</td>
<td>Reduce a 3D array to 2D (graphics application)</td>
<td>2.5GB</td>
</tr>
<tr>
<td>Sparse</td>
<td>Sparse matrix-vector multiplication</td>
<td>2 x 780MB</td>
</tr>
<tr>
<td>LU</td>
<td>Dense LU matrix factorization</td>
<td>800MB</td>
</tr>
<tr>
<td>MD</td>
<td>Module dynamic simulation</td>
<td>3 x 376MB</td>
</tr>
</tbody>
</table>
Matrix Multiplication

(a) Accesses to matrix A w/o iHarmonizer
(b) Accesses to matrix B w/o iHarmonizer
(c) Accesses to matrix A with iHarmonizer
(d) Accesses to matrix B with iHarmonizer
Matrix Multiplication

w/o iHarmonizer, during the first 105 seconds, a row of A and whole B is read, B is read column by column

With iHarmonizer, the prefetch thread loads part of A and B into memory and then performs calculation using the loaded data

Execution time reduced 132%
## Matrix Multiplication

The performance improvement increases with the decreasing memory size, because iHarmonizer makes I/O access more sequential. The more swappings happen, the more I/O time to save.

<table>
<thead>
<tr>
<th>Memory Size</th>
<th>Execution Time (seconds) \textit{w/o} iHarmonizer</th>
<th>Execution time (seconds) \textit{with} iHarmonizer</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>2GB</td>
<td>246.8</td>
<td>95.7</td>
<td>158%</td>
</tr>
<tr>
<td>1.5GB</td>
<td>387.5</td>
<td>123.8</td>
<td>213%</td>
</tr>
<tr>
<td>1GB</td>
<td>755.6</td>
<td>222.7</td>
<td>239%</td>
</tr>
<tr>
<td>512MB</td>
<td>1803.5</td>
<td>553.5</td>
<td>226%</td>
</tr>
</tbody>
</table>
Sparse Matrix

```c
for (r = 0; r < M; r++) {
    double sum = 0.0;
    int rowR = row[r];
    int rowRp1 = row[r+1];
    for (i = rowR; i < rowRp1; i++)
        sum += x[col[i]] * val[i];
    y[r] = sum;
}
```
Sparse Matrix

(a) Accesses to A without iHarmonizer

(b) Accesses to B without iHarmonizer
3D Histogram Reduction

(a) without iHarmonizer

(b) With iHarmonizer
Dense LU factorization

(a) Access to array file without iharmonizer  (b) Access to array file with iharmonizer
Molecular Dynamics

(a) Access of disk blocks without iharmonizer  (b) Access of disk blocks with iharmonizer
Evaluation Overall

![Graph showing execution time for different benchmarks with and without iharmonizer]
Evaluation by threads number
## Sensitivity by Queue Size

<table>
<thead>
<tr>
<th>Queue Size</th>
<th>Execution Time</th>
<th>Num of Phases</th>
</tr>
</thead>
<tbody>
<tr>
<td>64MB</td>
<td>221.2</td>
<td>1090</td>
</tr>
<tr>
<td>128MB</td>
<td>145.8</td>
<td>660</td>
</tr>
<tr>
<td>256MB</td>
<td>99.8</td>
<td>57</td>
</tr>
<tr>
<td>512MB</td>
<td>95.1</td>
<td>26</td>
</tr>
<tr>
<td>1GB</td>
<td>89.0</td>
<td>11</td>
</tr>
<tr>
<td>1.5GB</td>
<td>79.5</td>
<td>6</td>
</tr>
<tr>
<td>2GB</td>
<td>79.8</td>
<td>2</td>
</tr>
<tr>
<td>2.5GB</td>
<td>81.2</td>
<td>1</td>
</tr>
</tbody>
</table>
Related Work

I/O Prefetching

- TIP (Patterson et al)
- Speculative execution (F.Chang)
- Reorder buffer (Fast'10)

Align requests with disk data layout

- FS2 (SOSP'05)
- BORG (Fast'09)

Coordinate the execution of threads/processes

- SEDA (Apache MINA, FTP server)
Conclusion

- Demonstrate the non-deterministic behavior of multithreaded programs can seriously degrade I/O performance.

- Propose iHarmonizer, a user-level scheme that integrates compiler and runtime library to coordinate thread execution and reorder I/O requests.

- Implement iHarmonizer in the context of OpenMP and extensively evaluate it.