Implementing Nonnumerical Algorithms on a Decoupled Architecture Supporting Software Pipelining

A. Moshovos
Department of Computer Science
University of Crete
July 1992
Outline

- Area of interest
- Access Decoupling
- A first example: \( \min(a_0, \ldots, a_n) \)
- QuickSort on Access Decoupled Architectures
- Implementing Elementary Sorting Algorithms
- Implementing the KMP pattern matching algorithm
- Simulator overview
- Measurements
- Conclusions
Area of interest

- Considering Algorithms that:
  - Operate on Large Data Sets
  - Are nonnumerical — usually have data fetch/store sequences that are data dependent.
Access Decoupling

Purpose: Overcome execution delays due to memory accesses;
Alternative to cache memory technique

Caches Fail For Certain Access Sequences

```c
int a[1000000], b[1000000];
for (i=0; i<1000000; i++)
    b[i] = a[i]+1;
```

A simple Access Decoupled Architecture

2 instruction streams which can slip relative to each other
Access memory to fetch operands and store the results
Function execution to produce results

Execution flow:

<table>
<thead>
<tr>
<th>Time</th>
<th>Convetional</th>
<th>Address Generator</th>
<th>Processing Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>request a[0]</td>
<td>request a[0]</td>
<td>wait</td>
</tr>
<tr>
<td>n</td>
<td>b[0]=a[0]+1</td>
<td>request a[n]</td>
<td>b[0]=a[0]+1</td>
</tr>
</tbody>
</table>
- Separate address generators are used for each data queue:

Consider the process of Merging two sorted arrays:

while there are elements
  if \( a[i] > b[i] \)
    then \( o[n++] = a[i++] \)
  else \( o[n++] = b[i++] \)
end
A first example: Finding min(a0,..,an)

Conventional Method:

\[
\text{min} = a_0 \\
\text{for } i = 1 \text{ to last_element} \\
\text{if } a[i] < \text{min} \text{ then } \text{min} = a[i]
\]

- Instead Find:
  \[\text{min}(\text{min}(\text{subset1}), \text{min}(\text{subset2}), ..., \text{min}(\text{subset}l))\]
- The minimum of each subset depends only on its contents so the execution of those processes can be interleaved.

Execution Flow:

```
subset 1
  a_{1,0} < min_1
    a_{1,1} < min_1
      a_{1,x} < min_1
        min_1 < min_2

subset 2
  a_{2,0} < min_2
    a_{2,1} < min_2
      a_{2,x} < min_2

subset y
  a_{y,0} < min_y
    a_{y,1} < min_y
      a_{y,x} < min_y
        min_{y-1} < min_y
```

\{initialization\} \{execution\}
QuickSort on Access Decoupled Architectures

- Modify the algorithm to produce regular data fetching/storing sequences

- A new iteration can be initiated as soon as the previous one read the Mark value and updated the Read-FromQueue flag.
- It's nonrecursive
QuickSort on Access Decoupled Architectures

An implementation of QuickSort on our Architecture

- Two source arrays are used
- Each step updates both of them
- The data fetching sequence is slightly modified.

- Selecting the first element of a partition as the key degrades the performance of the partition process.

- Solution: Change the direction of fetching/storing after each iteration.
Elementary Sorting Algorithms

Insertion Sort

| 9 | 8 | 5 | 3 | 2 | 7 |
---|---|---|---|---|---|
PU | key = 7 |

Execution Flow

- After each insertion we must wait until all data are actually stored.
- It can be used on small sets produced by the application of an advanced sorting method as it has a complexity of $N^2$.

Bubble Sort

- Not appropriate, the elements compared at each step are not known before the completion of the previous iteration.

Selection Sort

- Not appropriate, the sequence of stores is data depended.

find $\min(a_x, \ldots, a_n)$

exchange $(a_x \rightarrow \min, \ldots)$

A. Moshovos
July 20, 1992

9
Using fixed/variable length keys

Insertion Sort

Fixed Length Keys:

- Fetching the key value into the 2nd queue simplifies the process

Variable Length Keys:

- Instead break the process in two parts:
  1. Compare, and record the results
  2. Use the comparison results to create the output sequence
Using fixed/variable length keys

QuickSort

- A small data store, local to PU, must be used.
- Each time a new partition starts the key value is copied into that memory store and later compared with the other elements.

1st phase: Compare

2nd phase: Create the two partitions.
**The KMP pattern matching algorithm**

- The *next* array is defined differently:
  
  Instead of using \( n[i] \) defines the pattern position from which the searching process must continue when a mismatch is detected at the \( i \) pattern element.

  \( nNew[i] \) defines the number of elements to flush from the data queue in order to reach the \( n[i] \) element.

- For the pattern 10100111 the \( n[] \) array is 0,1,1,2,3,1,2,2 and the \( nNew[]\) array becomes 8,7,6,6,5,3,3,2.

- Not appropriate for software pipelining as a new iteration cannot be initiated until the previous one completes.
Simulator Overview

- Cycle by Cycle simulation of six, concurrently running, functional units:
  - Three address generators
  - Processing Unit
  - Queues Controller
  - Memory Subsystem

- Unified scalar processor model for both Address Generators and Processing Unit.

- Two memory models included: Interleaved and Constant Delay.

- Configurable parameters include:
  - Data/Sync Queues size, instruction pipeline delays, pipeline depth, memory model and parameters (interleaving, delay, queues).

- Main memory is simulated by a binary file accessed through a simple paging mechanism.

- An interactive execution-trace monitor included.

- Written in C and compiled successfully on SunOS, SCO 386ix and on MSDOS. The code is about 3500 lines long.

- A simple assembler was implemented in AWK and uses CPP for macro processing.

- ~1.15K cycles/sec simulation speed. 3.6 Mcycles simulated in about 52 minutes on a SunSPARCstation-2.
Measurements

- **Partitioning 100,000 elements**

  **Conventional Architecture:**
  - A superscalar architecture with 4 operations/instruction.
  - ~9 instructions, $1/9 = 11\%$ of them are loads.
  - **900,000** cycles, with a 100% cache hit ratio.
  - $(0.89 + 0.11 \times 100) \times 9 \times 10^5 = 10,701,000$ cycles, if a 100 times slower memory is used and the miss ratio is 100%.
  - The architecture with a 100% hit ratio is **12 times** faster.

  **Access Decoupled Architecture:**
  - 18 instructions/iteration, $2/18 = 11\%$ of them are loads.
  - **3.600.115**, if memory delay is 100 cycles.
  - The architecture with a 100% hit ratio is only **4 times** faster.
  - The decoupled architecture is **3 times** faster than the conventional architecture with a 100% miss ratio.
Adding two arrays of size 10,000.

**Conventional Architecture**

• 2 instructions/iteration, 1/2 = 50% of them are loads.

• 20,000 cycles (100% cache hit ratio). (.5 + .5*100) * 20,000 or 1,010,000 cycles with a 100 times slower memory.

• The architecture with a 100% hit ratio is 50.5 times faster.

**Access Decoupled Architecture**

• 2 instructions, 1/2 = 50% of them are loads.

• 30117 cycles with a 100 cycles memory access delay.

• The conventional architecture with 100% hit ratio is 1.56 times faster.

• The decoupled architecture is ~32.4 times faster than the conventional architecture with a 100% miss ratio.
Measurements

- Executing a partitioning process over 1000 elements.
Conclusions

- Access Decoupling can reduce and possibly hide long memory delays.
- Its performance improves with the size of the data set applied on.
- Not all algorithms can execute efficiently on an Access Decoupled Architecture.
- Few nonnumerical algorithms can be implemented efficiently without modifications.
- Algorithm rewriting needed, as automatic program transformation seems impossible.

Future Work

- Behavior of a Superscalar Access Decoupled Architecture.
- Other nonnumerical algorithms.
Pipeline overview 1

- Stage 1:
Pipeline overview 2

Stage 2:

[Diagram showing pipeline stages with labels such as 'st2.wait', 'st2.xfer', 'st2.lpxfer', 'st2.isespm', 'st2.dst.pc', 'st2.ir', 'st2.pc', 'lr.valid', and various logic blocks like Control Logic, Guard File, Rotating Register File, Static Register File, Port File, and others labeled as 'LC', 'MCP', 'ESC', 'SP control logic', 'st2.write', 'st2.mem.mode', 'st2.dst.enabled', 'st2.dst', 'st2.DinA', 'st2.DinB', 'st2.DinC', 'st2.alu.mode', 'ps.can/comm', and 'imm10/imm16'.]
Pipeline overview 3

- Stages 3 to last.