Lightweight Dynamic Task Creation and Scheduling on the Intel Single Chip Cloud (SCC) Processor

Deepak Majeti

Department Of Computer Science
Rice University
Houston, Texas
Deepak.Majeti@rice.edu

April 7, 2011
Outline

1. Introduction
2. Implementation
3. Design
4. Experimental Results
5. Conclusion
Headlines

1. Introduction
2. Implementation
3. Design
4. Experimental Results
5. Conclusion
Introduction

Trend

- More and more processing units on a chip.
- Research Challenge: **High-performance + Power-efficiency**
Problem

- Programming these newer architectures is complicated.

**Figure:** Race conditions can put you in such a situation :)
More Problems

- Deadlocks, Livelocks

Why have these cars been abandoned?
Even More Problems

- Load Balancing
- Memory Consistency
- Core Utilization
- Debugging
Possible Approach

- Rice Habanero Multicore Software Project
  http://habanero.rice.edu/

**Figure: Habanero Approach**
Habanero Programming Model

- Lightweight asynchronous tasks and data transfer
  - async
  - finish
  - asyncMemcpy

- Locality control for task and data distribution
  - Hierarchical place tree

- Collective, point-to-point, stream synchronization
  - phasers
Habanero Build Model

HC Program (*.hc)

HC Compiler (HCC)

C Program (*.c)

HC Runtime

C Compiler (CC)

Executable
SCC System Overview

Figure: SCC System Overview (Figure Credit [SCC])
1. Introduction

2. Implementation
   - Async
   - Hierarchical Places
   - Work Stealing & Work Sharing

3. Design

4. Experimental Results

5. Conclusion
Async Example

```c
int fib(int n)
{
    if (n < 2) {
    }
    else {
        int x, y;
        finish {
            async IN(n) OUT(x){ x = fib(n - 1); }
            async IN(n) OUT(y){ y = fib(n - 2); }
        }
        n = x + y;
    }

    return n;
}
```

asynchronous lightweight task
Fib Task Graph...
Hand Coded Async Implementation

- The role of the async construct is to create an asynchronous lightweight task.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>int function</td>
<td>The function ID to be executed.</td>
</tr>
<tr>
<td>int label</td>
<td>Program point in the function where the execution must start.</td>
</tr>
<tr>
<td>void *input</td>
<td>The address location of the input data.</td>
</tr>
<tr>
<td>void *output</td>
<td>The address location where the values computed must be written to.</td>
</tr>
</tbody>
</table>

Table: Description of the async attributes

- An integer ID is used to represent a function instead of a function pointer. This is because in the SCC processor, each core has a different virtual address space.

- The address location for input or output could be an MPB or the off-chip DRAM.
Hierarchical Places

- Locality is important for performance.

- In many cases, parallel algorithms require locality [LCPC2009].

- The 'place' construct gives the programmer the ability to schedule tasks based on load balancing and locality.
Hand Coded Hierarchical Places

2 different memory locations.

```
for (int i = 0; i < 100; i++) {
    async_place(Work(A[i]), (i/50));
}
```
Hierarchical Places on SCC

Diagram showing a hierarchical structure of places and tiles connected by lines.

- Place P18 connects to P12, P19, and P20.
- Place P19 connects to P21 and P22.
- Place P20 connects to P21 and P22.
- Place P22 connects to P23.
- Place P0 connects to P1, P2, and P3.
- Place P1 connects to P2 and P3.
- Place P2 connects to P3.
- Place P3 connects to P4 and P5.
- Place P4 connects to P5.
- Place P5 connects to P11, P10, and P9.
- Place P6 connects to P7 and P8.
- Place P7 connects to P8 and P9.
- Place P8 connects to P9 and P10.
- Place P9 connects to P10 and P11.
- Place P11 connects to P12 and P13.
- Place P12 connects to P13 and P14.
- Place P13 connects to P14 and P15.
- Place P14 connects to P15 and P16.
- Place P15 connects to P16 and P17.
- Place P16 connects to P17 and P18.
- Place P17 connects to P18 and P19.
- Place P18 connects to P19 and P20.
- Place P19 connects to P20 and P21.
- Place P20 connects to P21 and P22.
- Place P21 connects to P22 and P23.
- Place P22 connects to P23.

Diagram indicates a structured network of places and tiles.
Work Stealing & Work Sharing

- **Work Stealing:**
  - The runtime supports a *help-first* work stealing scheduling policy.
  - "The help-first scheduling policy dictates that a worker executes the continuation and leaves the spawned task to be stolen" [IPDPS2010].

- **Work Sharing:**
  - The runtime also supports work sharing among the processors.
  - A single MPB queue on one of the tiles can now be used to share work among the worker threads.

- Concurrent queues have been implemented on the shared memory region available on the SCC. These are used to support the scheduling policies.
Current Habanero Build Model on SCC

HC Program (*.hc) → HC Compiler (HCC) → C Program (*.c) → C Compiler (CC) → HC Runtime → Executable

Hand coded async implementation
Headlines

1. Introduction
2. Implementation
3. Design
   - Locks
   - Queue
4. Experimental Results
5. Conclusion
There are no atomic operations and one can achieve synchronization only via test-and-set registers available on each core.

Pseudo code

```c
int lock(int tile, int core) {
    return Test_Set[tile][core];
}
```

```c
void unlock(int tile, int core) {
    Test_Set[tile][core] = 0;
}
```

Reading a '1' is a success

Write a value to reset
Queue

**Design Queue**

**Operation**

- `enqueue( place, async_frame )`
  - Inserts the asynchronous ’task’ at the **top** of the queue on the specified ’place’.

- `dequeue( place, async_frame )`
  - Removes the asynchronous ’task’ at the **bottom** of the queue on the specified ’place’.

**Table:** Description of queue operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue( place, async_frame )</td>
<td>Inserts the asynchronous ’task’ at the <strong>top</strong> of the queue on the specified ’place’.</td>
</tr>
<tr>
<td>dequeue( place, async_frame )</td>
<td>Removes the asynchronous ’task’ at the <strong>bottom</strong> of the queue on the specified ’place’.</td>
</tr>
</tbody>
</table>
A queue has been designed on top of the MPB and on the Off-Chip DRAM.

Synchronization of the queue located on a tile is achieved by using the locks on the same tile.

Synchronization of the queue located on the off-chip DRAM is achieved by using a lock on one of the tile.

The enqueue and dequeue methods use different test-and-set registers for their operations. This gives way to some parallelism as the enqueue and dequeue operations can now occur concurrently.
Headlines

1. Introduction
2. Implementation
3. Design
4. Experimental Results
   - Work Stealing Vs. Work Sharing
5. Conclusion
Setup 1

- Under the work-stealing approach, a master thread (PID 16) enqueues a task onto each queue (MPB of all the tiles) in a round-robin fashion via the hand coded async and place constructs.

- Under the work-sharing approach, a master thread (PID 16) enqueues tasks onto its own queue (MPB on tile 8).

- The worker threads compete for the tasks enqueued. Under the work stealing policy, if the worker’s local queue becomes empty, it tries to dequeue a single task from a random place(queue).
Performance Evaluation

We measure the performance of work stealing and work sharing in our implementation. A simple micro benchmark has been written using the async and place constructs in a for loop. Table 3 shows the pseudo code for the master and worker.

```
// Master code
// enqueues the "task" onto the
// queue at the given "place"

for(int i=0; i < ITERATIONS; i++){
    async_place(place,1);
    // place = i%24 in work stealing
    // = 8 in work sharing
}
// implicit finish

void async_place(int place,int function){
    async_node work;
    ...............;
    work.func = function;
    ...............;
    enqueue(place,work);
}

// Worker code
// Under work stealing;
// Dequeues a "task" from its own queue.
// if its queue is empty, it tries to
// grab a "single task" from a random queue.
// Under work sharing;
// Dequeues "work" from the master’s queue.

while(true){
    work = dequeue();
    if(work.func == 1){
        func_1();
    }
    void func_1(){
        for(int i=0; i<rand()%RANGE;i++){
            for(int j=0; j < WORK; j++){
                a=a+j;
            }
        }
    }
}
```

**Table:** Master and Worker pseudo code
## Experimental Results

### Work Stealing Vs. Work Sharing

**Results**

<table>
<thead>
<tr>
<th>Iterations</th>
<th>High variation in work per task</th>
<th>Total Time (sec)</th>
<th>Low variation in work per task</th>
<th>Total Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANGE = 100,000</td>
<td></td>
<td></td>
<td>RANGE = 1,000</td>
<td></td>
</tr>
<tr>
<td>ITERATIONS</td>
<td>Work Sharing</td>
<td>Work Stealing</td>
<td>ITERATIONS</td>
<td>Work Sharing</td>
</tr>
<tr>
<td>100</td>
<td>3.4</td>
<td>4.4</td>
<td>10,000</td>
<td>2.4</td>
</tr>
<tr>
<td>200</td>
<td>5.12</td>
<td>7.9</td>
<td>20,000</td>
<td>4.88</td>
</tr>
<tr>
<td>400</td>
<td>9.17</td>
<td>15.3</td>
<td>40,000</td>
<td>9.8</td>
</tr>
<tr>
<td>800</td>
<td>17.2</td>
<td>20</td>
<td>80,000</td>
<td>19.2</td>
</tr>
<tr>
<td>1,600</td>
<td>33</td>
<td>36</td>
<td>160,000</td>
<td>39</td>
</tr>
<tr>
<td>3,200</td>
<td>64.2</td>
<td>69.7</td>
<td>320,000</td>
<td>76.9</td>
</tr>
</tbody>
</table>

**Table:** Work stealing and Work sharing under varied work per task
Related Work

Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal ”A simple load balancing scheme for task allocation in parallel machines”[SPAA ’91]

- Load balancing strategy: ”At time t, before scheduling the next task from its local workpile, processor $i$ flips a coin and executes the load balancing task with probability $\frac{1}{L_{i,t}}$. $L_{i,t}$ is the length of the work pile at time $t$.

- The load-balancing task simply chooses some other PE at random and tries to equalize the load between the two workpiles.

- Show that work stealing (local workpile) with load balancing is ideal for shared-memory systems.
Setup 2

- The master thread (PID 16) enqueues tasks on a single central queue (Queue on Tile 8).

- The work-stealing policy mentioned in Rudolph et al. has been implemented.

- The work-sharing policy is same as in setup 1.
Results

- Work stealing scheduling strategy works better than work-sharing in most of the cases.

- In the case when the task load is highly varied and only few tasks are available, work sharing performs slightly better.

<table>
<thead>
<tr>
<th>Low Task Count</th>
<th>RANGE = 10,000,000</th>
<th>Total Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITERATIONS</td>
<td>Work Sharing</td>
<td>Work Stealing</td>
</tr>
<tr>
<td>50</td>
<td>15.2</td>
<td>19.8</td>
</tr>
<tr>
<td>100</td>
<td>23.6</td>
<td>27.5</td>
</tr>
<tr>
<td>200</td>
<td>40.1</td>
<td>43.3</td>
</tr>
<tr>
<td>400</td>
<td>74.2</td>
<td>76.6</td>
</tr>
<tr>
<td>500</td>
<td>90</td>
<td>90.1</td>
</tr>
</tbody>
</table>

Table: Work stealing and Work sharing under low task count
Headlines

1. Introduction
2. Implementation
3. Design
4. Experimental Results
5. Conclusion
Programming models like Habanero make parallel programming easier. It is important to support these models on new architectures like the SCC.

The uniqueness of SCC is that, it is a hybrid of a distributed architecture and a shared memory system. One can now gain the advantages of both the architectures. SCC also offers fine-grain power management. These features make SCC an interesting platform for dynamic task parallelism.

Work-stealing scheduler performs better than a work sharing scheduler in most of the cases.
X10: an object-oriented approach to non-uniform cluster computing.

http://habanero.rice.edu/hj.html.

Yi Guo, Barik Rajkishore, Raman Raghavan, and Vivek Sarkar.
Work-first and help-first scheduling policies for async-finish task parallelism.

Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar.
Hierarchical place trees: A portable abstraction for task parallelism and data movement.
Proceedings of the 22nd Workshop on Languages and Compilers for Parallel Computing (LCPC), 2009.

Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar.
Slaw: a scalable locality-aware adaptive work-stealing scheduler.
24th IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2010.

David Chase and Yossi Lev.
Dynamic circular work-stealing deque.

Single-chip Cloud Computer: An experimental many-core processor from Intel Labs.