Sports Arbitrage Bettor

CSEE 4840: Embedded Systems

Team: Brennan McManus, Shivan Mukherjee, Jonathan Nalikka, Chelsea Soemitro, Shreya Somayajula
**What is Arbitrage?**

- Bettors place bets provided by bookmakers (bookies).
- Bettor can place multiple bets on *same* event from different bookies to guarantee profit, no matter the outcome of the event.
- Determined using simple comparison calculation:

  \[
  \text{is } A + B < AB? \]

- **Arbitrage** if \( A + B < AB \) is true.
- **Not Arbitrage** if \( A + B < AB \) is false.
Our Project

Detect combinations of bets on NBA games that result in guaranteed profit – i.e. are arbitrage opportunities.
Workflow

1. **SCRAPE BOOKIES**
   - Use Python API to scrape various bookie websites

2. **PARSE RAW DATA**
   - Transform data from API (assign IDs, convert to fixed-point)

3. **ARBITRAGE DETECTION**
   - Send data to hardware to perform arbitrage detection

4. **USER INTERFACE**
   - Retrieve arbitrage results and calculate profit in software
Avalon Bus

Software

- Data Scraping & Parsing
- Main Program
- Kernel Driver

Hardware

- Grouping Module
- Calculation Modules
- Results BRAM

Functions:
- iowrite32()
- ioread32()
## Software Parsing: Bookie Mapping

<table>
<thead>
<tr>
<th>Bookie Name</th>
<th>Bookie ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>DraftKings</td>
<td>0</td>
</tr>
<tr>
<td>FanDuel</td>
<td>1</td>
</tr>
<tr>
<td>BetOnline.ag</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>SuperBook</td>
<td>12</td>
</tr>
</tbody>
</table>

the most we've seen is 13—can be represented in 4 bits!
Software Parsing: Fixed-Point Conversion

Floating point: x.xxx

Fixed point: $2^{10}, 2^9, \ldots, 2^2, 2^1, 2^0, 2^{-1}, 2^{-2}, 2^{-3}, 2^{-4}, 2^{-5}, 2^{-6}, 2^{-7}, 2^{-8}, 2^{-9}$
Software-Hardware Interface: Representation

Event struct (32-bit):

typedef struct {
    uint32_t odds: 20;
    uint32_t game_id: 4;
    uint32_t bookie_id: 4;
    uint32_t outcome: 1;
    uint32_t unused: 3;
} arb_event_t;

Result struct (32-bit):

typedef struct {
    uint32_t arb_prob: 20;
    uint32_t game_id: 4;
    uint32_t bookie_id_a: 4;
    uint32_t bookie_id_b: 4;
} arb_result_t;

Done struct (32-bit):

typedef struct {
    uint32_t done: 1;
    uint32_t result_count: 8;
    uint32_t padding: 23;
} arb_read_regs_t;
Software-Hardware Interface: Registers

Software

9-bit address

010100110

Hardware

write = 1

<table>
<thead>
<tr>
<th>Address 0</th>
<th>writedata ignored, raise arb_reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address 1</td>
<td>writedata ignored, raise arb_start</td>
</tr>
<tr>
<td>Address 2</td>
<td>write an event, raise arb_write</td>
</tr>
</tbody>
</table>

read = 1

<table>
<thead>
<tr>
<th>Address 0</th>
<th>done struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address 1</td>
<td>result struct</td>
</tr>
<tr>
<td>Address 2</td>
<td>result struct</td>
</tr>
<tr>
<td>Address 3 - 255</td>
<td>...</td>
</tr>
<tr>
<td>Address 256</td>
<td>result struct</td>
</tr>
</tbody>
</table>
Software-Hardware Interface: ioctls

CALC_ARB_WRITE_EVENTS

```c
static void write_events(struct event_buf *buf)
{
    int i;

    // (1) send reset signal
    iowrite32(0, ((uint32_t *)dev.virtbase) + ARB_RESET_ADDR);

    // (2) write events
    for(i = 0; i < buf->len; i++) {
        iowrite32(*((buf->events_vec) + i),
                    ((uint32_t *)dev.virtbase) + ARB_EVENT_WRITE_ADDR);
    }

    // (3) raise start
    iowrite32(0, ((uint32_t *)dev.virtbase) + ARB_START_ADDR);
}
```

CALC_ARB_READ_EVENTS

```c
static struct result_buf *read_result(void)
{
    arb_read_regs_t read_regs;
    int i;
    struct result_buf *results_buf;
    uint32_t readdata;

    // (1) poll for done
    while(1) {
        readdata = ioread32(((uint32_t *)dev.virtbase) + ARB_REGS_ADDR);
        read_regs = *((arb_read_regs_t *) &readdata);

        if (read_regs.done)
            break;
    }

    results_buf = kmalloc(sizeof(int) + read_regs.result_count * sizeof(arb_result_t), GFP_KERNEL);
    results_buf->len = read_regs.result_count;

    // (2) read results structs
    for (i=0; i < results_buf->len; i++) {
        uint32_t readdata = ioread32(((uint32_t *)dev.virtbase) + ARB_RESULT_READ_ADDR(i));
        results_buf->events_vec[i] = *((arb_result_t *) &readdata);
    }

    return results_buf;
}
```
Hardware: Grouping

Grouping Module

```
assign calc_write_selector = {{MAX_GAME_ID{1'b0}},{write} << wriedata[23:20];

// instance array of calc managers
calc_manager calc_man[MAX_GAME_ID: 0] (;
  .clk(clk), // duplicated
  .reset(reset), // duplicated
  .start(start), // duplicated
  .flushed(flushed), // split across instances
  .write(calc_write_selector), // split across instances
  .wriedata(wriedata), // duplicated
  .found_result(found_result), // split across instances
  .done(calc_done), // split across instances
  .result(result) // split across instances
);
```
Hardware Calculation: Event Writing Phase

Event Writing Phase

32-bit event

32 bit event writedata from avalon bus

event is "home"

8 bit address

next_even_addr

next_odd_addr

Input Event BRAM
(up to 256 32-bit events)

Port 1

home event

away event

Port 2

home event

away event
Hardware Calculation: Comparison Phase

Comparison Phase (After start signal received)

- **Iterator**
  - start
  - flushed
  - even_addr
  - odd_addr

- **Input Event BRAM** (up to 256 32-bit events)
  - Port 1
    - home event
    - away event
  - Port 2
    - home event
    - away event

- **Arbitrage Calculator**
  - home event
  - away event
  - found_result

- 32-bit event
- 1-bit start
- 8 bit addr
- 32-bit result
- 1-bit found_result
- 1-bit flushed
Hardware Calculation: calc_odds

assign a20 = a[19:0];
assign b20 = b[19:0];

assign ab = a20 * b20;
assign aplusb = {10'b0, a20 + b20, 10'b0};

always_comb begin
  if (aplusb < ab) begin
    found_result = 1;
    arb_prob = a20 + b20;
  end else begin
    found_result = 0;
    arb_prob = 0;
  end
end
Synchronization: A more detailed view

- Instance arrays help make things simpler!
- Narrow vectors (clk, writedata) duplicated: each instance gets a copy
- Wide vectors (calc_write_selector) distributed: each instance gets a slice
Synchronization: write_manager

Comparison Manager

32-bit result
found_result = 1
flushed = 1

Comparison Manager

32-bit result
found_result = 0
flushed = 0

Comparison Manager

32-bit result
found_result = 1
flushed = 0

Comparison Manager

32-bit result
found_result = 1
flushed = 0

4-bit source_index = 0

result found!
flushed raised

8-bit result_count

result found: result_count incremented & used as write address

Result B/R/M
Synchronization: write_manager

Comparison Manager

32-bit result
found_result = 0
flushed = 0

Comparison Manager

32-bit result
found_result = 0
flushed = 0

Comparison Manager

32-bit result
found_result = 1
flushed = 0

Comparison Manager

32-bit result
found_result = 1
flushed = 0

previous
"flushed"
lowered

no
result
found

4 bit source_index = 1

8 bit result_count

Result BRAM
Synchronization: write_manager

Comparison Manager
- 32-bit result
  - found_result = 0
  - flushed = 0

Comparison Manager
- 32-bit result
  - found_result = 0
  - flushed = 0

Comparison Manager
- 32-bit result
  - found_result = 1
  - flushed = 1

Comparison Manager
- 32-bit result
  - found_result = 1
  - flushed = 0

4 bit source_index = 2
8 bit result_count

result found: result_count incremented & used as write address

Result WRAM

"flushed" raised
Synchronization: write_manager

Comparison Manager

32-bit result
found_result = 0
flushed = 0

Comparison Manager

32-bit result
found_result = 0
flushed = 0

Comparison Manager

32-bit result
found_result = 0
flushed = 0

Comparison Manager

32-bit result
found_result = 0
flushed = 0

4 bit source_index = 3

8 bit result_count

result found: result_count incremented & used as write address

Result BRAM

previous "flushed" lowered

result found!

"flushed" raised
Efficiency

Pure Python Implementation
Arbitrage Detection: 0.220 seconds

FPGA Implementation
Arbitrage Detection: 0.006 seconds