Execution Model For Data Communication in the CELL/BE

Jairo Balart
Marc Gonzalez
Xavier Martorell
Eduard Ayguade

Barcelona Supercomputing Center

Kathryn O’brien
Kevin O’brien

IBM T.J. Watson Research Center
Index

- Motivation
- Asynchronous Communication
- Compiler Code Generation
- Evaluation
- Conclusions & Future Work
Cell Broadband Engine

- **Heterogeneous Multiprocessor**
  - Power processor
  - Synergistic processing elements

- **Power Proc. Element (PPE)**
  - general purpose
  - running full-fledged OSs
  - 2 levels of globally coherent cache

- **Synergistic Proc. Element (SPE)**
  - SPU optimized for computation density
  - 128 bit wide SIMD
  - Fast local memory (256 Kb)
  - Globally coherent DMA

- **High bandwidth**
  - Element interconnect bus
  - Integrated MIC and BIC – access dual XDR and I/O devices e.g., GPU
Execute computation in SPU

- **Transform original code**
  - Introduce DMA operations within the code
  - Synchronization statements
  - Translate from original address space to local address space

- **Manual solution**
  - Very optimized codes but at cost of programmability
    - Overlap of communication with computation

- **Automatic solution**
  - Compiler and runtime system support
  - Tiling, Double Buffer
    - Good solution for regular applications
  - Software Cache
    - Usually performance is limited to the available information at compile time
    - Very difficult to generate code that overlaps computation with communication
Move data with a Software Cache

- **Protocol to treat a memory reference**
  - Check for data already present in local storage
  - In case not present, decide where to place it
  - If necessary, decide what to send out from local storage
  - Perform DMA operations
  - Synchronize with DMA
  - Translate from virtual address space to local storage address space
Overlap communication with code execution

- **What we would like**
  - Decouple code related to lookup, placement and replacement from DMA program

```c
for (i=0; i<NUM_ITERS; i++) {
    v1[i] = v2[i];
    v3[v1[i]]++;
}
```

```
1  ld @v2[i], r1
2  st r1, @v1[i]
3  ld @v3[r1], r2
4  inc r2
5  st r2, @v3[r1]
```

- Ensure there is no conflict between B1 and B2
- Memory barrier time could be filled with A, B, C from iteration i+1
Avoid stalls caused by flush operations

- We would like also
  - Avoid an immediate reuse of a portion of Local Storage that has been flushed

```c
for (i=0;i<NUM_ITERS;i++) {
    v1[i] = v2[i];
    v3[v1[i]]++;
}
```

### Diagram:

- A: Check presence
- B: Placement/Replacement
- C: Translate
- D: Access

- Memory Barrier

- DMA OUT 1

- DMA IN 1

- Worst case, an immediate reuse
Avoid stalls caused by flush operations

- After a flush operation
  - Delay as much as possible next use of space in Local Storage

```c
for (i=0;i<NUM_ITERS;i++) {
    v1[i] = v2[i];
    v3[v1[i]]++;
}
```

```
1 ld @v2[i], r1
2 st r1, @v1[i]
3 ld @v3[r1], r2
4 inc r2
5 st r2, @v3[r1]
```
Index

- Motivation
- Software Cache with Asynchronous Communication
- Compiler Code Generation
- Evaluation
- Conclusions & Future Work
Software Cache Implementation (1)

- **Cache Parameters**
  - $C = \textit{capacity}$
  - $N = \textit{number of cache lines} \ (C / L)$
  - $L = \textit{cache line size}$
  - $S = \textit{number of sets}$

- **Cache Structures**
  - **Directory**
    - Directory is composed by $S$ lists
      - **FULL ASSOCIATIVE**
  - **FREE list**
    - Initially, cache lines are stored in this list
  - **CLEAN list**
    - List of cache lines that have been used for READ operations, but are no longer in use
  - **DIRTY list**
    - List of cache lines that have been used for READ/WRITE operations, but are no longer in use
Software Cache Implementation (2)

**Cache Line Information**

- **Reference counter**
  - Counts how many memory references point to the cache line

- **Different state values**
  - **USED** \( (\text{ref. counter} \neq 0) \)
  - **UNUSED** \( (\text{ref. counter} = 0) \)

- **Different attributes**
  - **CLEAN** (not modified)
  - **DIRTY** (modified)
  - **FLUSHED** (cache line has been flushed to main memory)
  - **LOCKED** (cache line is excluded from replacement mechanisms)

- **State and attributes determine the list where to place the cache line**
  - Directory lists
  - CLEAN/DIRTY lists
Software Cache Implementation (3)

- **Cache Look Up**
  - Implemented in two phases
    - First phase
      - For each memory reference the implementation *keeps track* of the base address of the cache line in virtual memory
        > @vm_cacheline
      - For each memory reference the implementation checks for a change on the cache line
        > In case a change occurred, go through **second phase**
          
          \[
          \text{MASK} = \sim(L-1) \quad \text{CHANGE} = ( (@addr \text{ AND } \text{MASK}) \neq \text{@vm_cacheline})
          \]

    - Second phase:
      - Access directory
        > Hash function: most significant bits of @vm_cacheline modulo S
        > Look for the cache line in the synonymous list
Software Cache Implementation (4)

**Reference Counter**

- Used to control *Write Back* and *Replacement policy*
- Modified when Look Up mechanism detects a change in a memory reference
  
  • Decrement for previous referenced cache line
    
    – If ref. counter = 0
    
    > **Not extracted** from directory
    
    > For **MODIFIED** lines, *immediate write back* and place the cache line in the **DIRTY list** as **LAST**
    
    > For **UNMODIFIED** lines, place the cache line in the **CLEAN list**
  
  • Increment for new referenced cache line
Software Cache Implementation (5)

- **Write Back**
  - Only affects modified cache lines
  - Performed when the *reference counter equals 0*
  - Implemented through a *asynchronous* DMA operation
    - *Record* the *TAG* used in the DMA operation

- **Cache Line Placement**
  - Apply hash function, place line in a set
  - Initially, all cache lines are in the *FREE* list
  - When a miss occurs
    - If *FREE list* is not empty, take the *FIRST* of the list
    - If *FREE list* is empty, invoke replacement action
Software Cache Implementation (6)

- **Cache Line Replacement**
  - If **CLEAN list is not empty**,
    - Take one cache line in the list
    - **Extract** from directory and CLEAN list, then **insert** in directory
  - If **DIRTY list** not empty
    - Select **FIRST** from DIRTY list
    - **Extract** from directory and DIRTY list, then **insert** in directory
    - Perform **DMA synchronization** using the TAG used at Write Back execution
  - If both **CLEAN/DIRTY lists** are **empty**, replace with the first line entered in the cache
Software Cache Implementation (7)

- **Address Translation**
  - Computes offset within a cache line
    - OFFSET = @addr AND (L-1)
  - Add OFFSET to @cacheline
“Double Buffer” behavior can be achieved

- Replacement assigns FIRST available cache line in DIRTY list
- Replacement takes back a modified cache line as LAST in DIRTY list
- A cache line is never assigned again before all other available cache lines have been
- $N$ greater than the number of memory “objects” in the code
  - **Best case, $N = 2 \times \text{no. memory operations}**

```c
for (i=0;i<NUM_ITERS;i++) {
    v1[i] = v2[i];
    v3[v1[i]]++;
}
```
Index

- Motivation
- Software Cache with Asynchronous Communication
- Compiler Code Generation
- Evaluation
- Conclusions & Future Work
Code Generation (1): Basic algorithm

- **Given a code region**
  - Compute must-alias sets and for each set, assign a **identifier**
    - Identifier that links uses of data to runtime structures to perform **look up** (phase 1) and **translation**

- **For each Basic Block**
  - For each referenced set
    - **Lookup** (1st phase)
      - Check if cache line has changed
    - **Memory map operation**
      - Look up (2nd phase), Placement / Replacement, Communication (DMA)
      - Tagged with an identifier
    - **Memory barrier**
      - Blocks execution until DMA pending operations are completed
      - Tagged with an identifier
    - **Address translator**
      - Responsible for address translation
      - Tagged with an identifier
Example

```c
for (i=0;i<NUM_ITERS;i++) {
    v1[i] = v2[i];
    v3[v1[i]]++;
}
```

_LOOKUP:
- Phase 1: checks for cache line change

_MMAP:
- Phase 2: access directory
- Placement/Replacement
- Increments/decrements number of references
- Update cache line flags
- Programs non-blocking DMA

_MEM_BARRIER:
- Blocks until pending DMA complete

_LD,_ST:
- Translates and executes memory access

```c
for (i=0;i<NUM_ITERS;i++) {
    if (_LOOKUP(0, ,&v2[i],...)) {
        _MMAP(0,&v2[i],...);
        _MEM_BARRIER(0);
    }
    if (_LOOKUP(1, ,&v1[i],...)) {
        _MMAP(1,&v1[i],...);
        _MEM_BARRIER(1);
    }
    _LD(0,&v2[i],_int_tmp00);
    _ST(1,&v1[i],_int_tmp00);
    if (_LOOKUP(2, ,&v3[_int_tmp00],...)) {
        _MMAP(2,&v3[_int_tmp00],...);
        _MEM_BARRIER(2);
    }
    _LD(2,&v3[_int_tmp00],_int_tmp01);
    _int_tmp01++;
    _ST(2,&v3[_int_tmp00],_int_tmp01);
}
```
Code Generation (3): Optimization for MMAP

- **Within a BB**
  - Define the *distance* for a MMAP operation
    - *Maximum number* of MMAP operations between a LD/ST with the same identifier as the MMAP
      - Example
        
        ```c
        if (_lookup_01) {
            _MMAP(0,...);
            _MMAP(1,...);
            _MMAP(2,...);
            _MMAP(3,...);
            _MMAP(4,...);
            _MEM_BARRIER(0);
        }
        _LD(0,...);
        ...
        ```

- **Within a BB**
  - Try to move upward MMAP operations
    - As much as possible but
      - Never cross a dependence edge
      - Never make any *distance greater or equal* than the *number of cache lines*
Code Generation (4): Example

Example

```c
for (i=0; i<NUM_ITERS; i++) {
  if (_LOOKUP(0, , &v2[i], ...)) {
    _MMAP(0, &v2[i], ...);
    _MEM_BARRIER(0);
  }
  if (_LOOKUP(1, , &v1[i], ...)) {
    _MMAP(1, &v1[i], ...);
    _MEM_BARRIER(1);
  }
  _LD(0, &v2[i], _int_tmp00);
  _ST(1, &v1[i], _int_tmp00);
  if (_LOOKUP(2, , &v3[_int_tmp00], ...)) {
    _MMAP(2, &v3[_int_tmp00], ...);
    _MEM_BARRIER(2);
  }
  _LD(2, &v3[_int_tmp00], _int_tmp01);
  _int_tmp01++;
  _ST(2, &v3[_int_tmp00], _int_tmp01);
}
```
Code generation (5): Avoiding conflicts

- **Under particular configurations**
  - C = 64 Kb  
    - L = 128 bytes  
    - N = 512
  - C = 64 Kb  
    - L = 512 bytes  
    - N = 128
  - C = 64 Kb  
    - L = 1024 bytes  
    - N = 64

- Take care of distance between memory map and use of data
  - _MMAP, _LD, _ST
- Ensure that no distance becomes greater or equal than N memory map operations are performed
Code Generation (6): Strided accesses

- Example

Lock cache lines for stride accesses

```c
for (i=0;i<NUM_ITERS;i++) {
  _LOOKUP(1, ,&v1[i],...,_lookup_01));
  _LOOKUP(0, ,&v2[i],...,_lookpu_01));
  if (_lookup_01) {
    _MMAP(0,&v2[i],...);
    _MMAP(1,&v1[i],...);
    _MEM_BARRIER(0,1);
  }
  _LD(0,&v2[i],_int_tmp00);
  _ST(1,&v1[i],_int_tmp00);
  if (_LOOKUP(2, ,&v3[_int_tmp00],...)) {
    _MMAP(2,&v3[_int_tmp00],...);
    _MEM_BARRIER(2);
  }
  _LD(2,&v3[_int_tmp00],_int_tmp01);
  _int_tmp01++;
  _ST(2,&v3[_int_tmp00],_int_tmp01);
}

_lb_01 = 0; _ub_01 = NELEM;
_work_01 = (_lb_01 < _ub_01);
while (_work_01) {
  _start_01 = _lb_01;
  _LOOKUP(0, ,&v2[i],...,_lookpu_01);
  if (_lookup_01) _MMAP(0, &v2[_start_01], ..., LOCK);
  _LOOKUP(1, ,&v1[i],...,_lookup_01);
  if (_lookup_01) _MMAP(1, &v1[_start_01], ..., LOCK);
  _next_iters_01 = LS_PAGE_SIZE;
  _NEXT_MISS(0, &v2[_start_01], float, sizeof(float), _next_iters_01);
  _NEXT_MISS(1, &v1[_start_01], float, sizeof(float), _next_iters_01);
  _end_01 = _start_01 + _next_iters_01;
  if (_end_01 > _ub_01) _end_01 = _ub_01;
  _lb_01 = _end_01;
  _work_01 = (_lb_01 < _ub_01);
  _MEM_BARRIER();
  for (int i = _start_01; i < _end_01; i=i+1) {
    _LD(0,&v2[i],_int_tmp00);
    _ST(1,&v1[i],_int_tmp00);
    if (_LOOKUP(2, &v3[_int_tmp00],...)) {
      _MMAP(2,&v3[_int_tmp00],...);
      _MEM_BARRIER(2);
    }
    _LD(2,&v3[_int_tmp00],_int_tmp01);
    _int_tmp01++;
    _ST(2,&v3[_int_tmp00],_int_tmp01);
  }
}
```

Affects the distance bound

© 2007 IBM Corporation
Unrolling inner loop by 2

```c
_lb_01 = 0; _ub_01 = NELEM;
_work_01 = (_lb_01 < _ub_01);
while (_work_01) {
    _start_01 = _lb_01;
    _LOOKUP(1, &v1[i], ... , _lookup_01);
    _LOOKUP(0, &v2[i], ... , _lookup_01);
    if (_lookup_01) {
        _MMAP(0, &v2[ _start_01], ... , LOCK);
        _MMAP(1, &v1[ _start_01], ... , LOCK);
    }
    next_iters_01 = LS_PAGE_SIZE;
    _NEXT_MISS(0, &v2[ _start_01], float, sizeof(float)...
    _NEXT_MISS(1, &v1[_start_01], float, sizeof(float)...
    _end_01 = _start_01 + _next_iters_01;
    if (_end_01 > _ub_01) _end_01 = _ub_01;
    _lb_01 = _end_01;
    _work_01 = (_lb_01 < _ub_01);
    _MEM_BARRIER();
} for (int i = _start_01; i < _end_01; i=i+2) {
    /* Iteration i */
    _LD(0, &v2[i], _int_tmp00);
    _ST(1, &v1[i], _int_tmp00);
    if (_LOOKUP(2, &v3[i], ... , _lookup_01)) {
        _MMAP(2, &v3[i], ... , LOCK);
        _MEM_BARRIER(2);
    }
    _LD(2, &v3[_int_tmp00], _int_tmp01++);
    _ST(2, &v1[_start_01], ... , LOCK);
    _LOOKUP(2, &v3[_int_tmp00], ... , _lookup_01);
    /* Iteration i+1 */
    _LD(0, &v2[i+1], _int_tmp00);
    _ST(1, &v1[i+1], _int_tmp00);
    if (_LOOKUP(2, &v3[i+1], ... , _lookup_01)) {
        _MMAP(2, &v3[i+1], ... , LOCK);
        _MEM_BARRIER(2);
    }
    _LD(2, &v3[_int_tmp00], _int_tmp01++);
    _ST(2, &v1[_start_01], ... , LOCK);
    _LOOKUP(2, &v3[_int_tmp00], ... , _lookup_01);
} for (int i = _start_01; i < _end_01; i=i+2) {
    /* Iteration i */
    _LD(0, &v2[i], _int_tmp00);
    _ST(1, &v1[i], _int_tmp00);
    if (_LOOKUP(2, &v3[i], ... , _lookup_01)) {
        _MMAP(2, &v3[i], ... , LOCK);
        _MEM_BARRIER(2);
    }
    _LD(2, &v3[_int_tmp00], _int_tmp01);
    _int_tmp01++; _ST(2, &v1[_start_01], ... , LOCK);
    _LOOKUP(2, &v3[_int_tmp00], ... , _lookup_01);
    /* Iteration i+1 */
    _LD(0, &v2[i+1], _int_tmp00);
    _ST(1, &v1[i+1], _int_tmp00);
    if (_LOOKUP(2, &v3[i+1], ... , _lookup_01)) {
        _MMAP(2, &v3[i+1], ... , LOCK);
        _MEM_BARRIER(2);
    }
    _LD(2, &v3[_int_tmp00], _int_tmp01);
    _int_tmp01++; _ST(2, &v1[_start_01], ... , LOCK);
    _LOOKUP(2, &v3[_int_tmp00], ... , _lookup_01);
} 
```
Index

- Motivation
- Software Cache with Asynchronous Communication
- Code Generation
- Evaluation
- Conclusions & Future Work
Evaluation

- **Hardware**
  - Dual Cell BE-based blade, two Cell Broadband Engine processors
    - 3.2 GHz (SMT enabled)
    - 512 MB each processor

- **Software**
  - Linux Fedora Core 5
    - Linux Kernel 2.6.18 (NUMA enabled)
  - SPE Management Library 1.1 (libspe)
  - GNU Toolchain 4.1.0 for the CBE's PPU
Evaluation

- **VECTOR ADD**

  ```c
  for (i=0;i<NUM_ITERS;i++) {
    v1[i] = v[i] + v2[i] + v3[i];
  }
  ```

- 32Mb each vector
- Repeated 40 times
- PPU time: 2.50 secs
- 64Kb of SC
- No SIMD

<table>
<thead>
<tr>
<th>Cache Line Size</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous DMA</td>
<td>12.97</td>
<td>8.09</td>
<td>5.57</td>
<td>4.32</td>
<td>3.92</td>
<td>3.79</td>
</tr>
<tr>
<td>Synchronous DMA</td>
<td>22.48</td>
<td>12.94</td>
<td>8.08</td>
<td>5.65</td>
<td>4.42</td>
<td>4.06</td>
</tr>
<tr>
<td>Overlap %</td>
<td>42.30</td>
<td>37.48</td>
<td>31.06</td>
<td>23.54</td>
<td>11.31</td>
<td>6.65</td>
</tr>
</tbody>
</table>
Evaluation

- **JACOBI**
  - Size of working set 1.5 Mb
  - Repeated 100 times

```c
for (iters=0; iters<NITERS; iters++) {
    for (i=STENCIL_WIDTH; i<N-STENCIL_WIDTH; i++) {
        stencil_sweep (coeff, &Blin[i*CB], &Alin[(i-STENCIL_WIDTH)*CB]);
    }
    tmpptr = Alin; Alin = Blin; Blin = tmpptr;
}

void stencil_sweep(float coeff[STENCIL_WIDTH+1],
                    float computed_row[CB],
                    float WA[2*STENCIL_WIDTH+1][CB])
{
    int i, j;

    for (i=0; i<STENCIL_WIDTH; i++) {
        for (j=STENCIL_WIDTH; j<CB-STENCIL_WIDTH; j++){
            computed_row[j]+= coeff[STENCIL_WIDTH-i] * WA[i][j];
        }
    }

    for (j=STENCIL_WIDTH; j<CB-STENCIL_WIDTH; j++){
        for (i=1; i<=STENCIL_WIDTH; i++) {
            computed_row[j]+= coeff[i] * (WA[STENCIL_WIDTH][j-i] + WA[STENCIL_WIDTH][j+i]);
        }
    }
}
```
**Evaluation**

- **JACOBI**
  - Size of working set 1.5 Mb
  - Repeated 100 times
  - PPU time: 1.96 secs
  - 64 Kb of SC
  - No SIMD

![Graph showing execution time for JACOBI](image)

<table>
<thead>
<tr>
<th>Cache Line Size</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous DMA</td>
<td>10.47</td>
<td>7.14</td>
<td>5.47</td>
<td>4.65</td>
<td>4.19</td>
<td>4.06</td>
</tr>
<tr>
<td>Synchronous DMA</td>
<td>13.35</td>
<td>8.68</td>
<td>6.36</td>
<td>5.19</td>
<td>4.49</td>
<td>4.20</td>
</tr>
<tr>
<td>Overlap %</td>
<td>21.57</td>
<td>17.74</td>
<td>13.86</td>
<td>10.40</td>
<td>6.68</td>
<td>3.33</td>
</tr>
</tbody>
</table>
Index

- Motivation
- Software Cache with Asynchronous Communication
- Code Generation
- Evaluation
- Conclusions & Future Work
Conclusions and Future Work

Conclusions

– Implementation of a Software Cache able of
  • Decouple cache management code with communication
  • Makes possible overlap communication with computation
– Define a compiler algorithm for compiler code generation
– Preliminary evaluation

Future Work

– More evaluation
  • Regular and Irregular codes
– Implement consistency model in the Software Cache
END
Evaluation

- IS