The Rise and Fall of Scratchpad Memories

Aviral Shrivastava
Compiler Microarchitecture Lab
Arizona State University
Remember - It is all about Memory!

- First Generation
  - ENIAC, UNIVAC – No memory
- Second Generation
  - IBM 7000 series - Magnetic core memory
- Third Generation
  - IBM 360 - Semiconductor memory
- Fourth Generation
  - PC and onwards - VLSI Memory

- First documented use of Cache
  - IBM 360*
  - “to bridge the speed gap between processor and memory”

- Since then: Caches maybe the most important feature in a processor
- Itanium 2: cache and cache-like structures
  - More than 90% of transistors by count, 70% of chip by area, 50% power, 80% of leakage

*IBM (June, 1968), IBM System/360 Model 85 Functional Characteristics, SECOND EDITION, A22-6916-1.
SPMs for Power, Performance, and Area

CachSPM

- 40% less energy as compared to cache [Banakar02]
  - Absence of tag arrays, comparators and muxes
- 34% less area as compared to cache of same size [Banakar02]
  - Simple hardware design (only a memory array & address decoding circuitry)
- Simpler and cheaper to build and verify

Web page: aviral.lab.asu.edu
SPMs became popular in ES

- DSPs have used SPMs for a long time
  - TI-99/4A, released in 1981 had 256 bytes of SPM

- Gaming Consoles regularly use SPMs
  - SuperH in Sega
  - PS1: could use SPM for stack data
  - PS2: 16KB SPM
  - PS3: Each SPU has 256KB SPM

- Network and Graphics Processors
  - Intel Network processors, and Nvidia Tesla

- Many embedded processors used line locking
  - Coldfire MCF5249, PowerPC440, MPC5554, ARM940, and ARM946E-S

- Several versions of ARM and Renesas have SPMs
  - ARM supports upto 4M of SPM

Web page: aviral.lab.asu.edu
Using SPMs in Embedded Systems

- Programs work without using SPM
  - SPM for optimization
  - Improve power, performance

- Placing frequently used data in SPM
  - Typically arrays
  - Using linker script

All of this was done manually!
Compilers for using SPMs

- As applications become more complex, it was not easy to identify what should be mapped to SPM

- Compiler techniques to use SPM in embedded systems
  - **Global:** Panda97, Brockmeyer03, Avissar02, Gao05, Kandemir02, Steinke02, Grosslinger09
  - **Code:** Janapsatya06, Egger06, Angiolini04
  - **Stack:** Udayakumaran06, Dominguez05
  - **Heap:** Dominguez05, McIlroy08
Compilers to use SPM

- In general-purpose systems
  - Kennedy – proposed to use SPM for register spills

Not much work
- Because caches keep programming and debugging simple

- SPMs have largely remained in embedded systems
- Not popular in general-purpose computing

Times are a changing...
Inevitable March to Multi-Cores

- Marketing Needs
  - Moore’s Law

- Real Needs
  - Temperature and Power Problems
    - Microarchitecture level: Hotspots
    - Chip level: Cooling Efficiency
    - System Level: Total power consumption
  - Only way to improve performance without much increase in power

- Multi-cores
  - Reduce design complexity
  - Spread heat and alleviate hotspots
  - Improve reliability through redundancy
But... how do you scale the memory?

- Coherent-Cache Architectures (Current path)
  - Can still write programs like in the uni-core era, but

- Coherency overheads do not scale
  - Tilera64 has a whole separate mesh network for coherence traffic

- Non-Coherent Cache Architectures
  - 48-core Single-chip Cloud Computer (SCC)

- Partly Coherent
  - TI-6678 – vertically coherent, but horizontally not coherent

- Hybrid
  - Locally coherent, but globally non-coherent

- Caches still consume a very significant amount of power
Software Managed Memory (SMM) Architecture

- Cores have small local memories (scratch pad)
  - Core can only access local memory
  - Accesses to global memory through explicit DMAs in the program
- e.g. IBM Cell architecture, which is in Sony PS3.

![Diagram showing PPE, SPEs, SPU, and LS connected through the Element Interconnect Bus (EIB).]

PPE: Power Processor Element
SPE: Synergistic Processor Element
LS: Local Store

**Diagram Details**
- Off-chip Global Memory
- Element Interconnect Bus (EIB)
- SPE 0, SPE 1, SPE 2, SPE 3, SPE 4, SPE 5, SPE 6, SPU, LS

Web page: aviral.lab.asu.edu
Task based programming, MPI like communication

```
#include<libspe2.h>
extern spe_program_handle_t hello_spu;
int main(void)
{
    int speid, status;
    speid = spe_create_thread(&hello_spu);
}
```

Main Core

Extremely power-efficient computation
- If all code and data fit into the local memory of the cores

<table>
<thead>
<tr>
<th>Processor</th>
<th>Fab</th>
<th>Frequency</th>
<th>GFlops</th>
<th>Power</th>
<th>Power Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell/B.E.</td>
<td>45nm</td>
<td>3.2 GHz</td>
<td>230</td>
<td>50 W</td>
<td>4.6</td>
</tr>
<tr>
<td>Intel i7 4-core Bloomfield 965 XE</td>
<td>45nm</td>
<td>3.2 GHz</td>
<td>70</td>
<td>130 W</td>
<td>0.5</td>
</tr>
</tbody>
</table>
Dynamic code/data management is needed

All code/data must be managed

SPM is for Optimization

SPM is Essential

Previous works are not directly applicable
How to manage data within a core?

Original Code

```c
int global;

f1()
{
    int a,b;
    global = a + b;

    f2();
}
```

Local Memory Aware Code

```c
int global;

f1()
{
    int a,b;
    DMA.fetch(global)
    global = a + b;
    DMA.writeback(global)

    DMA.fetch(f2)
    f2();
}
```
Data Management in LLM multicores

- Manage any amount of **heap, stack** and **code**, in the core of an LLM multi-core

  - **Global data**
    - If small, can be permanently located in the local memory
  
  - **Stack data**
    - ‘liveness’ depends on call path
    - Function stack size know at compiler time, but not stack depth
  
  - **Heap data**
    - dynamic and size can be unbounded
  
  - **Code**
    - Statically linked

- Our strategy
  - Partition local memory into regions for each kind of data
  - Manage each kind of data in a constant amount of space

Web page: aviral.lab.asu.edu
Stack Management: Problem

<table>
<thead>
<tr>
<th>Function</th>
<th>Frame Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>28</td>
</tr>
<tr>
<td>F2</td>
<td>40</td>
</tr>
<tr>
<td>F3</td>
<td>60</td>
</tr>
<tr>
<td>F4</td>
<td>54</td>
</tr>
</tbody>
</table>

Local Memory Size = 128 bytes

Local Memory

Global Memory

Function F1

Function F2

Function F3

Function F4

Local Memory Size = 128 bytes
Stack Management: Solution

- Keep the active portion of the stack on the local memory
- Granularity of stack frames is chosen to minimize management overhead
- It is a dynamic software technique
  - `fci(func_stack_size)`
    - Check for available space in local memory
    - Move old frame(s) to global memory if needed
  - `fco()`
    - Check if the caller frame exists in local memory!
    - Fetch from global memory, if it is absent
Code Management: Problem

- Static Compilation
  - Functions need to be linked before execution
- Divide code part of SPM in regions
- Map functions to these SPM regions
- Functions in the same region replace each other
Code Management: Solution

(a) Application call graph

SECTIONS {
  OVERLAY {
    F1.o
    F3.o
  }
  OVERLAY {
    F2.o
  }
}

(b) Linker script

- # of Regions and Function-To-Region Mapping
  - Two extreme cases
- Need careful code placement – Problem is NP-Complete
  - Minimum data transfer with given space
**Heap Data Management**

```c
typedef struct{
    int id;
    float score;
} Student;

main() {
    for (i=0; i<N; i++) {
        student[i] = malloc( sizeof(Student) );
    }

    for (i=0; i<N; i++) {
        student[i].id = i;
    }
}
```

- **malloc()**
  - Allocates space in local memory

- **New malloc()**
  - May need to evict older heap objects to global memory
  - It may need to allocate more global memory

Heap Size = 32bytes
`sizeof(student)=16bytes`

- New malloc()
Pointer Threat: Problem

F1() {
    int a=5, b;
    fc(F2);
    F2(&a);
    fco(F1);
}

F2(int *a) {
    fc(F3);
    F3(a);
    fco(F2);
}

F3(int *a) {
    int j=30;
    *a = 100;
}

Stack Size= 100 bytes
Stack Size= 70 bytes

Aha! FOUND “a”
Wrong value of “a”
**Pointer Threat: Resolution**

F1() {
    int a=5, b;
    fci(F2);
    F2(&a);
    fco(F1);
}

F2(int *a) {
    fci(F3);
    F3(a);
    fco(F2);
}

F3(int *a) {
    int j=30;
    *a = 100;
    t = g2l(a)
    *t = 100;
    l2p(a, t);
}

F1() {
    int a=5, b;
    fci(F2);
    fco(F1);
}

F2(int *a) {
    fci(F3);
    F3(a);
    fco(F2);
}

F3(int *a) {
    int j=30;
    t = g2l(a)
    *t = 100;
    l2p(a, t);
}

*ptr = val;  \iff  tptr = _g2l(ptr);
*tptr = val;

l2p(ptr, tptr);

val = *ptr;  \iff  tptr = _g2l(ptr);
val = *tptr;
How to evict data to global memory?

- Can use DMA to transfer heap object to global memory
  - DMA is very fast – no core-to-core communication
- But eventually, you can overwrite some other data
  - Need mediation
Our infrastructure includes:

- code overlay script generating tool,
- runtime library implementing the API,
- compiler that inserts API functions in the application.

### Runtime Library API

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void * _malloc(int size, int chunkSize);</code></td>
<td>Memory allocation</td>
</tr>
<tr>
<td><code>void _free (void *ppeAddr);</code></td>
<td>Memory deallocation</td>
</tr>
<tr>
<td><code>void _fci(int func_stack_size);</code></td>
<td>Stack function call initialization</td>
</tr>
<tr>
<td><code>void _fco();</code></td>
<td>Stack function call optimization</td>
</tr>
<tr>
<td><code>void * _g2l(void *ppeAddr, int size, int wrFlag);</code></td>
<td>Memory to SPE address conversion</td>
</tr>
<tr>
<td><code>void * _l2g(void *ppeAddr, void* speAddr, int size);</code></td>
<td>SPE address to memory conversion</td>
</tr>
</tbody>
</table>
Experimental Setup

- Sony PlayStation 3 running a Fedora Core 9 Linux
  - Only 6 SPEs available

- MiBench Benchmark Suite and some other applications

- Runtimes are measured with `spu_decremener()` for SPE and `_mftb()` for the PPE provided with IBM Cell SDK 3.1

- Download GCC compiler patch
  - http://aviral.lab.asu.edu/?p=95
Without management the program crashes! There is no space left in local memory for the stack.

Enable execution for arbitrary stack sizes
But quite high overheads!

```c
int rcount(int n)
{
    if (n==0) return 0;
    return rcount(n-1) + 1;
}
```
How does it work?

- Pretty bad!!
- Several programs run, but with high overhead
- Several program still do not run

- Pointer problem
- How to evict to global memory
- Reduce overheads
  - # of times API functions are called
  - # of times DMA is performed

- Good news: It only gets better from here!
Reduce Data Transfer Overhead

```c
malloc() {
    if (enough space in global memory) then
        write heap data using DMA
    else
        request more space in global memory
}
```

Execution Thread on execution core

Global Memory

DMA write from local memory to global memory
Improving Stack Management

- Opportunities to reduce repeated API calls by consolidation

Sequential Calls

```c
fci(F1);
F1();
F2();
fco(F0);
fco(F0);
fci(max(F1,F2));
F1();
F2();
fco(F0);
```

Nested Call

```c
fci(F1);
F1(
  F2();
}fco(F0);
```

Call in loop

```c
while(<condition>){
  fci(F1);
  F1();
  fco(F0);
}
```

```c
while(<condition>){
  fci(F1);
  F1();
  fco(F0);
}fco(F1);
```

```c
fci(F1);
while(<condition>){
  fci(F1);
  F1();
  fco(F0);
}
```

```c
fci(F1);
while(<condition>){
  fci(F1);
  F1();
  fco(F0);
}fco(F1);
```

```c
fci(F1+{F2);
F1(}F2();
}fco(F0);
```
Find optimal stack management points

- Can consolidate function frame movement
  - Do not need to move functions at every function call

- Formulate the problem as that of inserting cuts in the GCCFG
  - At the cut, dump the SPM contents into global memory
More Stack Management Optimizations

- Movement of functions
  - Biggest contributor
  - Consolidate management for multiple functions

- Pointer management
  - Reduce the number of times p2s is called
    - If stack variable is used continuously – perform p2s only once
    - If the stack variable belongs to the function that is in the SPM, do not need p2s

- Reduce the instructions in management functions
  - SPM-level management is simpler
  - Less fragmentation – so the management code is less
Efficient Execution

- Very few fci and fco calls inserted
- Less number of g2l calls
- Less number of instructions executed at every management point
**Overheads**

### Table 3: Number of \_sstore/\_fci and \_sload/\_fco Calls

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>_sstore/_fci</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CSM</td>
</tr>
<tr>
<td>BasicMath</td>
<td>40012</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>60365</td>
</tr>
<tr>
<td>FFT</td>
<td>7190</td>
</tr>
<tr>
<td>FFT_inverse</td>
<td>7190</td>
</tr>
<tr>
<td>SHA</td>
<td>57</td>
</tr>
<tr>
<td>String_Search</td>
<td>503</td>
</tr>
<tr>
<td>Susan_Edges</td>
<td>776</td>
</tr>
<tr>
<td>Susan_Smoothing</td>
<td>112</td>
</tr>
</tbody>
</table>

### Table 4: Code size of stack manager (in bytes)

<table>
<thead>
<tr>
<th></th>
<th>_sstore/_fci</th>
<th>_sload/_fco</th>
<th>_l2g</th>
<th>_g2l</th>
<th>_wb</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>NF</td>
<td>F</td>
<td>NF</td>
<td>F</td>
</tr>
<tr>
<td>CSM</td>
<td>180</td>
<td>100</td>
<td>148</td>
<td>95</td>
<td>24</td>
</tr>
<tr>
<td>SSDM</td>
<td>46</td>
<td>0</td>
<td>44</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

### Table 5: Dynamic instructions per function

<table>
<thead>
<tr>
<th></th>
<th>_sstore/_fci</th>
<th>_sload/_fco</th>
<th>_l2g</th>
<th>_g2l</th>
<th>_wb</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>NF</td>
<td>F</td>
<td>NF</td>
<td>F</td>
</tr>
<tr>
<td>CSM</td>
<td>180</td>
<td>100</td>
<td>148</td>
<td>95</td>
<td>24</td>
</tr>
<tr>
<td>SSDM</td>
<td>46</td>
<td>0</td>
<td>44</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

### Table 6: Number of pointer mgmt. function calls

<table>
<thead>
<tr>
<th></th>
<th>_l2g</th>
<th>_g2l</th>
<th>_wb</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CSM</td>
<td>SSDM</td>
<td>CSM</td>
</tr>
<tr>
<td>BasicMath</td>
<td>37010</td>
<td>0</td>
<td>123046</td>
</tr>
<tr>
<td>SHA</td>
<td>2</td>
<td>2</td>
<td>163</td>
</tr>
<tr>
<td>Edges</td>
<td>1</td>
<td>0</td>
<td>515</td>
</tr>
<tr>
<td>Smoothing</td>
<td>1</td>
<td>0</td>
<td>515</td>
</tr>
</tbody>
</table>

### Table 2: Comparison of number of DMAs

<table>
<thead>
<tr>
<th></th>
<th>CSM</th>
<th>SSDM</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicMath</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>108</td>
<td>364</td>
</tr>
<tr>
<td>FFT</td>
<td>26</td>
<td>14</td>
</tr>
<tr>
<td>FFT_inverse</td>
<td>26</td>
<td>14</td>
</tr>
<tr>
<td>SHA</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>String_Search</td>
<td>380</td>
<td>342</td>
</tr>
<tr>
<td>Susan_Edges</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Susan_Smoothing</td>
<td>12</td>
<td>4</td>
</tr>
</tbody>
</table>

Web page: aviral.lab.asu.edu
Minimal Overhead

- 4% of execution time spent on management
Comparison with Caches

- Cache Miss penalty = \# misses * miss latency
- SPM miss overhead = \# API function calls * no. of instructions in API function + \# times DMA is called * delay of the DMA (dep. on DMA size)

Cache is better when miss latency < 260 ps

260 ps = 0.86 * cycle time
The main core does not choke on the memory requests from several cores
Summary

- SPMs are an embedded system technology
- SPMs will be needed in general-purpose computing

- Will need to manage stack, heap, and code
  - Do not work without management

- Need different strategies for different data
  - Code (statically linked)
  - Stack (Circular)
  - Heap (High associativity)

- Overheads of Software Data Management
  - DMA overhead can be comparable or better than cache

- We have just begun – lots of room for improvement
Communication Management

- No problem in MPI-style
  - Communication is explicit

- For multi-threaded programs
  - Replace load => coh_load(), and store => coh_store()
  - Too much overhead for sequential consistency

- Weak Consistency models allow for efficient software implementations of coherency protocols
  - Lazy vs. Eager
  - Invalidate vs. Update

- Page based granularity in multi-processor systems
  - Need finer granularity in multi-cores

Web page: aviral.lab.asu.edu
Real-time Multicores

- Data and communication management in software
  - Better timing guarantees
    - Managing data at its natural granularity simplifies WCET calculation
    - e.g., find out how many instruction cache misses vs. find out how many function swaps
  - Not only lower WCET, but tighter WCET estimate

- Excellent platform for Real-time Systems
- Can tune the management policy to improve WCET

- Software Branch Hinting
  - Close to 1-bit HBP performance
  - Can place hints to achieve tighter WCET

Web page: aviral.lab.asu.edu