

# Optimizing Compiler for a Cell Processor

Eichenberger et al.

Presented by Chris Bunch

CS 290N

# Needs for a new architecture

- Numerically intensive workloads:
  - Multimedia
  - Game applications
    - Parallel: Image processing / game physics
    - Serial: Networking / game AI
- Need a fast response time and first-class programming environment

# Introducing Cell

- Novel heterogeneous architecture
  - 1 Power Processing Element (PPE)
    - 64-bit multi-threaded processor
    - 2 levels of globally coherent cache
  - 8 Synergistic Processing Elements (SPEs)
    - Optimized for streaming workloads
    - Each contain local memory and a globally coherent DMA engine
  - 1 Element Interconnect Bus (EIB)

# The Cell Architecture

- PPE accelerates multimedia with IBM's Vector Multimedia eXtensions (VMX)
- SPE's local memory is non-coherent, but its DMA engine is globally coherent
- SPEs are SIMD, 128 bits of data at a time
- Instructions can take up to 3 128 bit operands and output 1 128 bit result.

# SPE's Execution Units

- Each SPE has 7 execution units that are organized into “even and odd pipes”
- Parallelism across pipes
- Heterogeneous pipe processors:

Instructions	Pipe	Lat.
arithmetic, logical, compare, select	even	2
shift, rotate, byte sum/diff/avg	even	4
float	even	6
16-bit integer multiply-accumulate	even	7
128-bit shift/rotate, shuffle, estimate	odd	4
load, store, channel	odd	6
branch	odd	1-18

# Memory

- Memory instructions: Fully pipelined 16-byte accesses
- Instruction fetching / DMA transfers: 128-byte accesses
- Memory has one port, so all 3 of these compete for it

# Alleviating this Problem

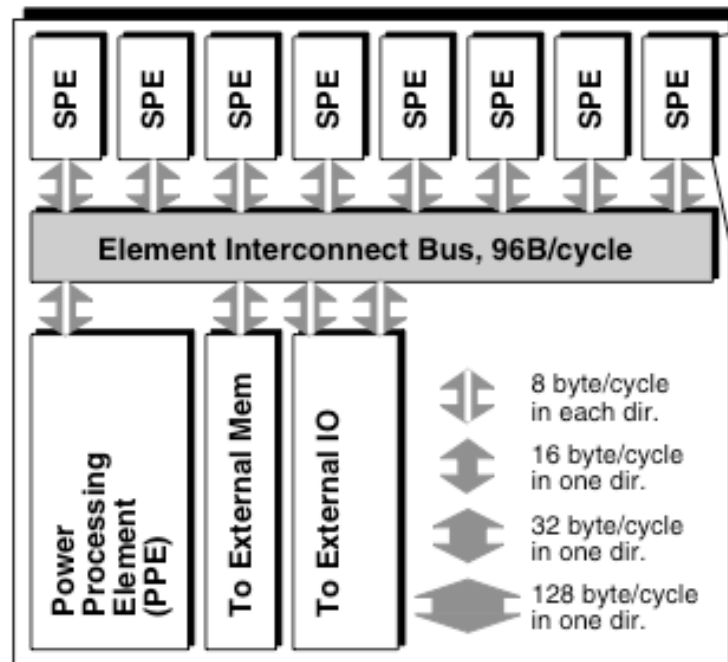
- Instruction fetches occur during idle memory cycles
- 32-bit instruction buffer can store up to 3.5 fetches (really 2.5)
- Explicit instructions can be used to initiate an inline instruction fetch

# Data Transfer

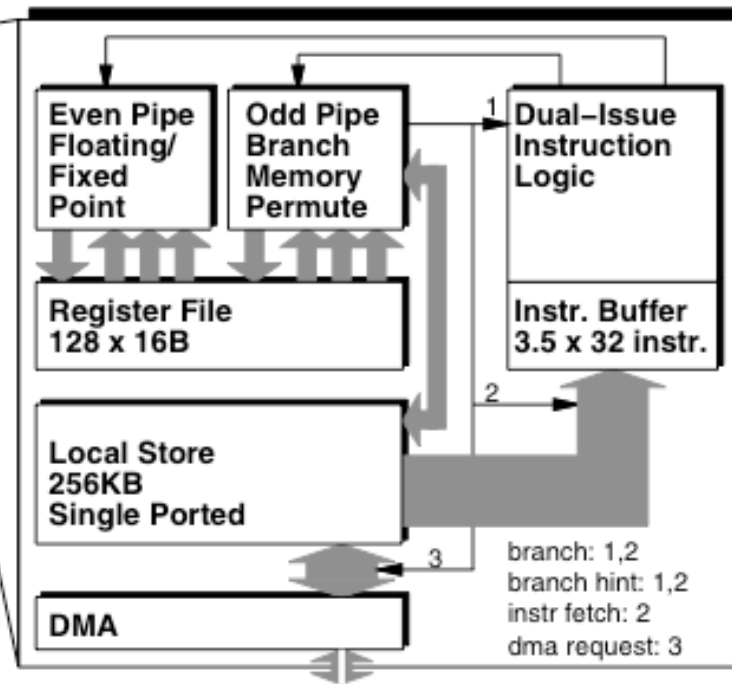
- DMA and local memory = 128-byte chunks
- DMA and EIB = 8 bytes/cycle
- PPE and SPEs can initiate DMA requests from/to each other's local memory as well as from/to global memory

# Putting It All Together

a) CELL Processor



b) Synergistic Processing Element (SPE)



# Contributions to Cell

- Offloading traditional hardware features to the compiler. Avg Speedup = 1.3
- Automatically generate SIMD code that fully utilizes the SPEs and the PPE. Speedup = 9.9
- Gather the memory across all the processors to present a shared memory model to the user. Speedup = 7.1

# Optimization 1: Let the Compiler do it!

- In order to perform operations across multiple registers, the data must have the same offset.
- If not aligned, the hardware permutes the data so that it is (expensive!)
- The compiler allocates 128-bit chunks of memory for scalars.
- (Downside) Registers are 128-bits wide.

# Branch Prediction

- By default, always predicts not taken.
- Branch hint instruction can override this behavior.
- This instruction also causes the branch's next 32 instructions to be pre-fetched.

# Branch Optimizations

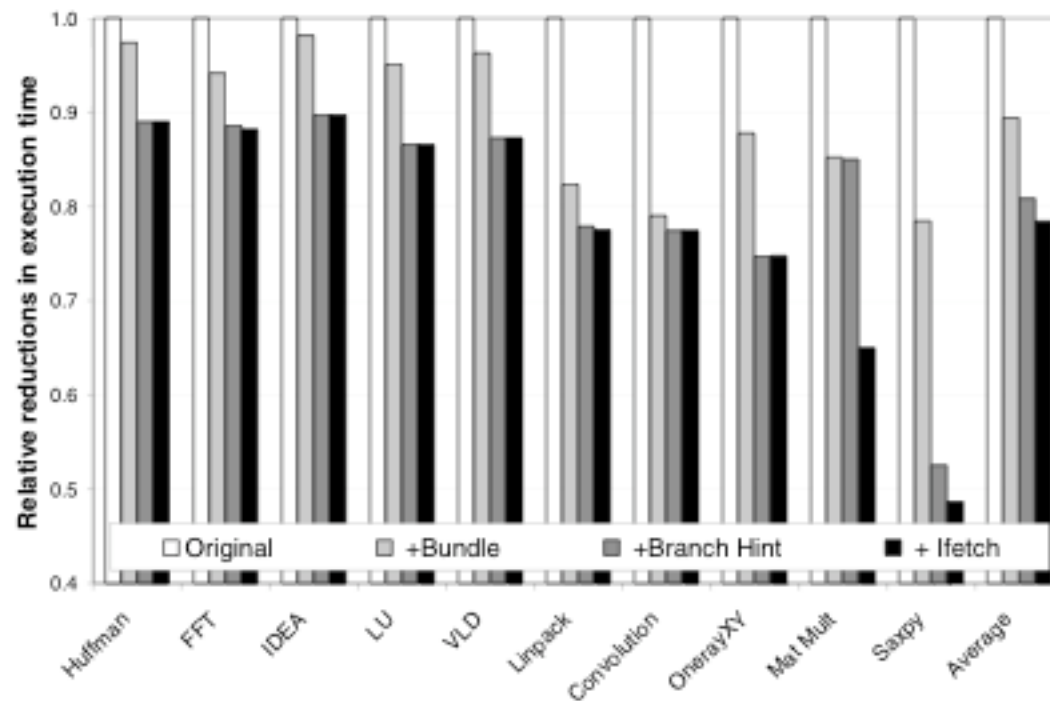
- Branching penalty is up to 18 cycles
- Hint for branch instruction ( $8 < \text{HfB} < 256$ )
- Two args: Branch location, likely target
- Can only do one at a time, but are valid until another is inserted (good for loops)

# Starvation

- Instruction buffer can hold 3.5 instructions, but 1 is reserved for HfB
- Since local memory has only one port, we could fill the buffer and starve the system in 40 dual-issued cycles.
- Do useful work and add nops as needed.

# Reduction in Execution Time

- Ranges from 11% - 51% (Avg 22%)



# Optimization 2: SIMD Code

- SIMD parallelism is great
- SPEs and the PPE have different instruction sets, but the same general SIMD framework.
- Find the parallelism, extract it, and convert it to the right ISA based on whether it's a SPE or the PPE

# Basic Block Level Simdization

- When is it useful to do this?
- Checking loop bounds, convert scalar data to a vector

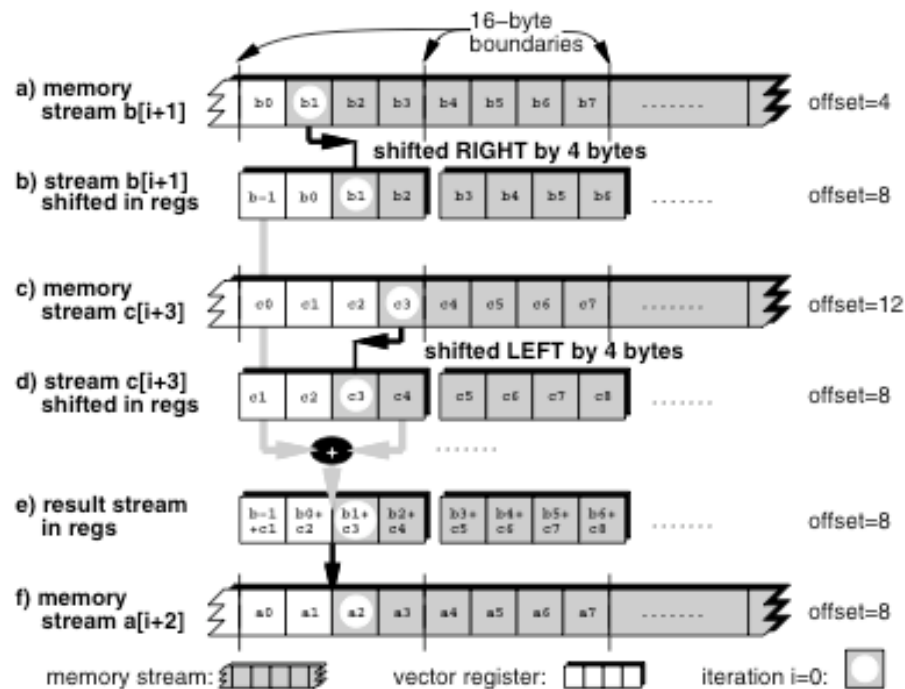
```
    for (i=0; i<n; i++) {  
1:   q = quads[i];  
2:   v[i].x=w0*q.p[0].x+w1*q.p[1].x+w2*q.p[2].x;  
3:   v[i].y=w0*q.p[0].y+w1*q.p[1].y+w2*q.p[2].y;  
4:   v[i].z=w0*q.p[0].z+w1*q.p[1].z+w2*q.p[2].z;  
5:   v[i].w=w0*q.p[0].w+w1*q.p[1].w+w2*q.p[2].w;  
    }
```

# A Similar Problem Reappears

- (Problem) Memory subsystems must be 16-byte aligned, otherwise we have to re-align the data. Need the same offset again.
- Can't use the same solution as before.
- (Solution) Align the data ourselves in a SIMD fashion.

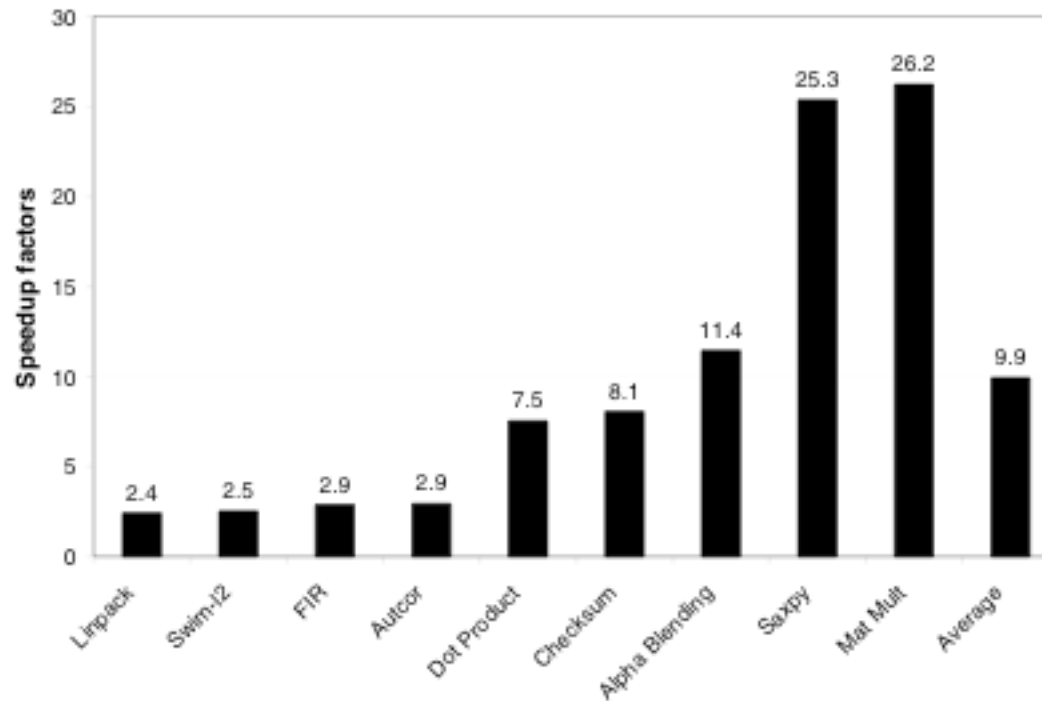
# Example

```
for (i=0; i<100; i++) {
    a[i+2] = b[i+1] + c[i+3];
}
```



# Speedup Incurred

- Speedup Factors: 2.4 - 26.2 (Avg 9.9)



# Optimization 3: MIMD Code

- Distribute loop slices to SPEs
- Needs a user directive to do so, and requires abstracting all the SPE's memory into a shared memory model
- Each SPE manages a 4-way set associate Software Cache

# Software Cache

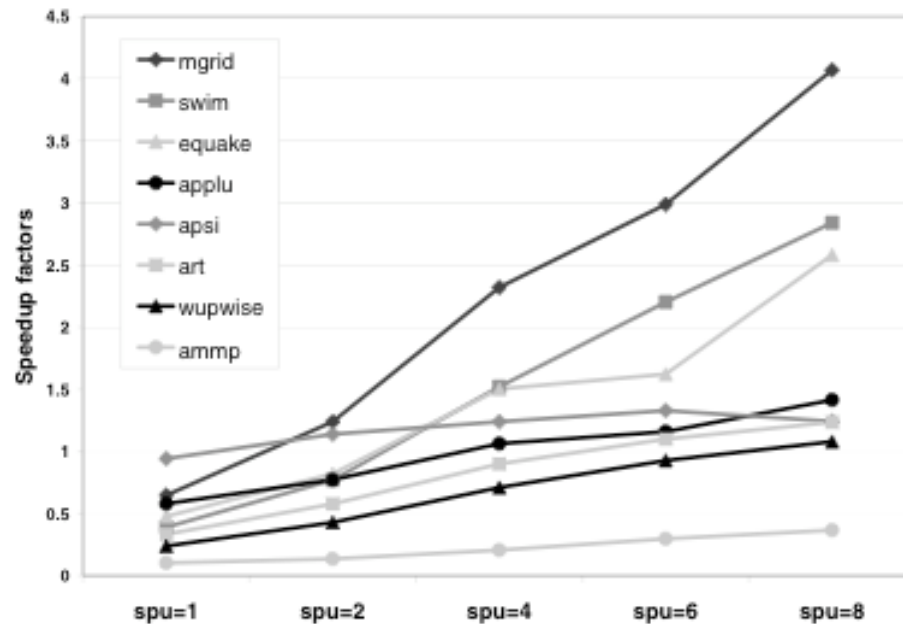
- Similar operation as a hardware cache
- Cache probes cost most (not misses)
- Uses invalidation for cache coherence
- Static data buffering: data not shared is treated differently, difficulties with this...
- Extension - Optimize DMA calls

# Code Partitioning

- Work needs to be efficiently sent to each SPE (low overhead)
- Use prefetching to hide this latency
- Slowdown is not too bad working on one SPE v. the PPE (debatable...)
- EEMBC Benchmark: 2% - 10% Worse
- CJPEG - Slowdown of 2.7 on SPE

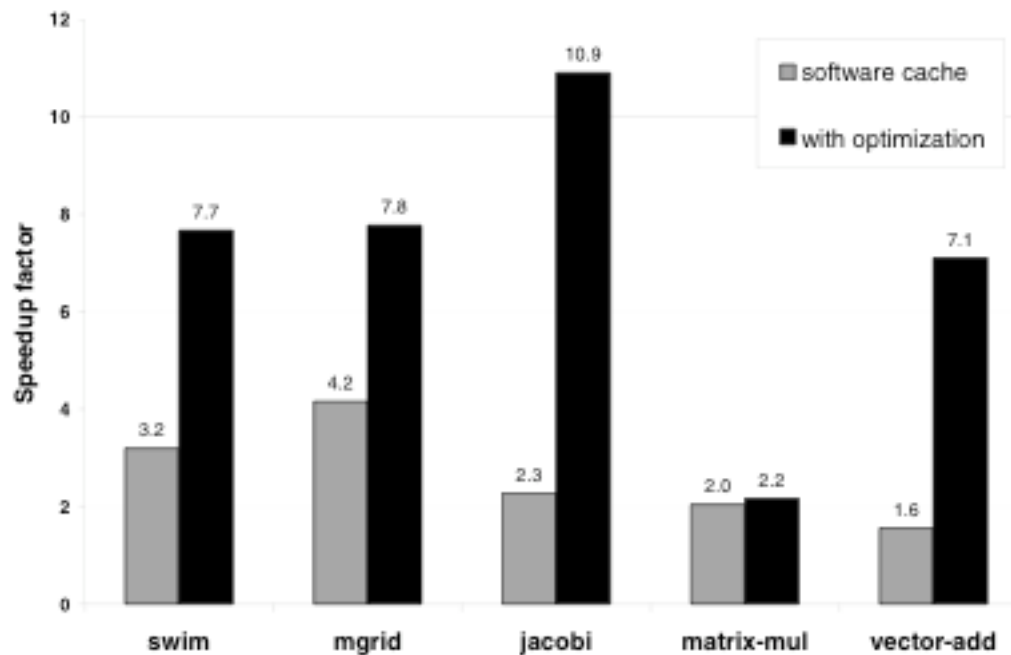
# Speedup Incurred

- Had to reduce data set sizes
- Does not include SIMD optimization



# With Code Partitioning

- Using all 8 SPEs:



# Drawbacks

- These benchmarks suffer when there is:
  - Poor data locality: Requires DMA
    - DMA optimization needs work
  - Frequent cache flushing (expensive!)
    - Needs work optimizing this
  - Optimizations turned off
    - Needed smaller code for the SPEs

# Review

- Cell Architecture
- The compiler's new responsibilities:
  - Data Management
  - Branch Prediction
  - Code Generation
  - Shared Memory Model
- Extensions

# Criticisms

- Speed of SPE v. PPE is unclear
  - In EEMBC, a “good” result was a SPE ~2% slower than a PPE
  - Without simdization, a SPE is 2x faster than a PPE
  - Could be because the PPE is better at vector arithmetic and this work may have been mostly vector-based (not obvious).

# Praise

- Lots of interesting opportunities for Cell
  - Heterogeneous system
  - Great speedups achieved so far
    - But how bad was the old system?
- Lots of areas to branch off onto from this paper
  - Merging SIMD code generation and MIMD code generation