Coconut Multicore Framework: Static Analysis Assisted Scheduling

Kevin Browne
brownek@mcmaster.ca

McMaster University
IBM Center for Advanced Studies Fellow

May 20, 2009

Cell BE Multicore Issues and Solutions

Coconut Multicore Framework

Coconut Advantages

Verification Tool

Performance Simulator

Current Status and Future Work

Multicore Scheduling Problems

Program Correctness

- Difficult to fix race condition bugs produce unpredictable program state due to un-synchronized concurrent events
- Increasing number of cores only increases difficulty
- Development costs quickly become prohibitive

Optimal Scheduling

- Main memory bandwidth key bottleneck
- Communication patterns may also cause bottlenecks
- Number of cores needing data likely to increase faster than bandwidth

New Tools, Frameworks and Solutions Emerge

Cell BE Solutions

- Accelerated Library Framework (ALF) [CHKW08]
- Cellgen [SYR+08]
- Cell Superscalar (CellSs) [BPBL06]
- CorePy [MML07]
- Mercury Multi-core Framework [BCG+06]
- MPI Microtask [OIS+06]
- MSL [ZLRA08]

New Tools, Frameworks and Solutions Emerge

More Cell BE Solutions

- Octopiler [EOO+06]
- OpenCL [Mun08]
- Peakstream [Pap07]
- Rapidmind [MWHL06]
- Sequoia [FHK+06]
- SysCellC [KHH08]

Our Solution: Coconut Multicore Framework

- Runtime system asynchronously executes streams of atomic virtual opcodes (AVOps) on each processor element to co-ordinate parallelism

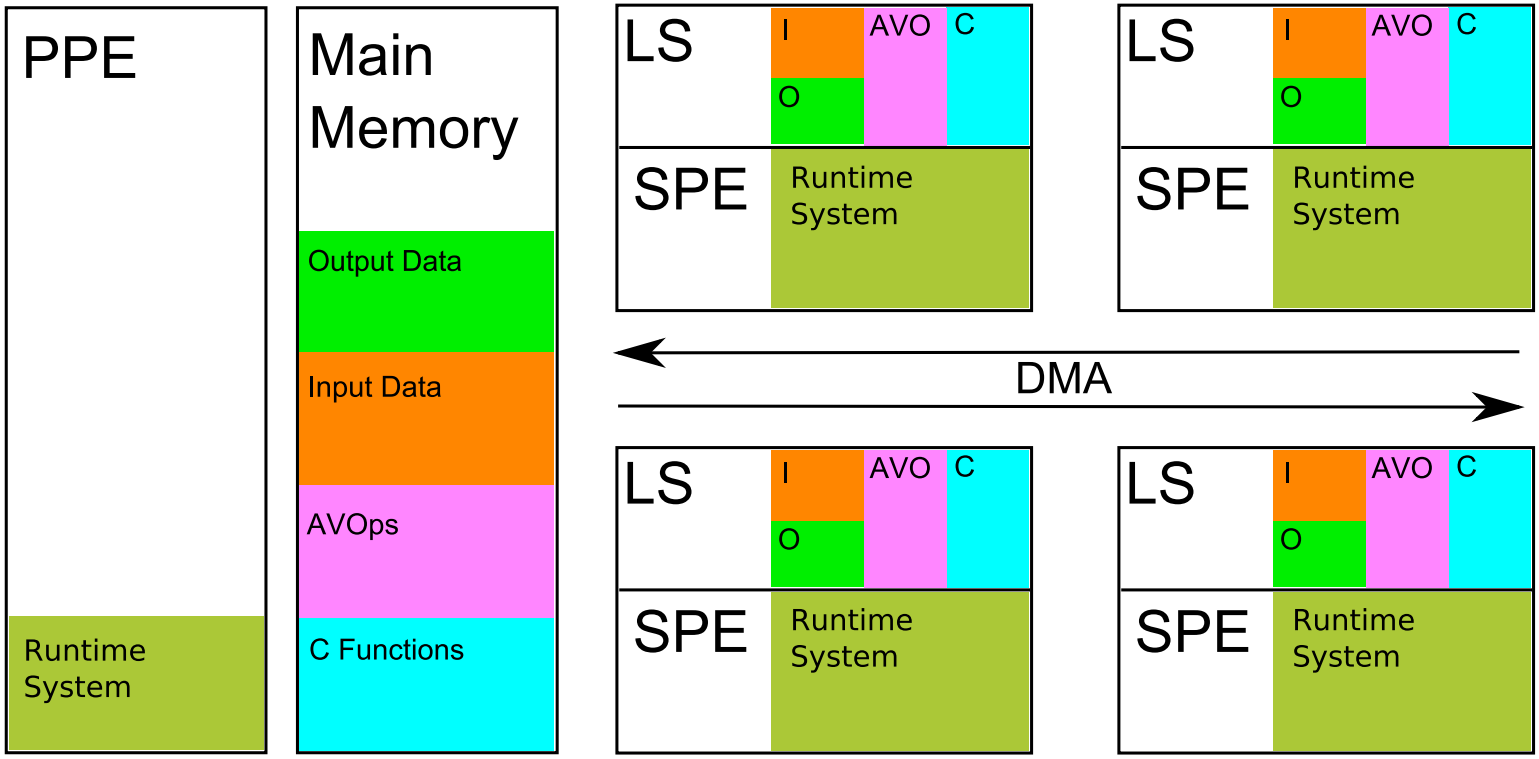


Figure: Coconut Multicore Framework view at execution time

Coconut Multicore Framework: Runtime System

- Runtime system thread on each SPE, executes AVOps
- Runtime system loads computational kernels at start-up
- AVOps are sent from the MIC to each SPE, buffered locally
- AVOps execute computational kernels exploiting SIMD ILP

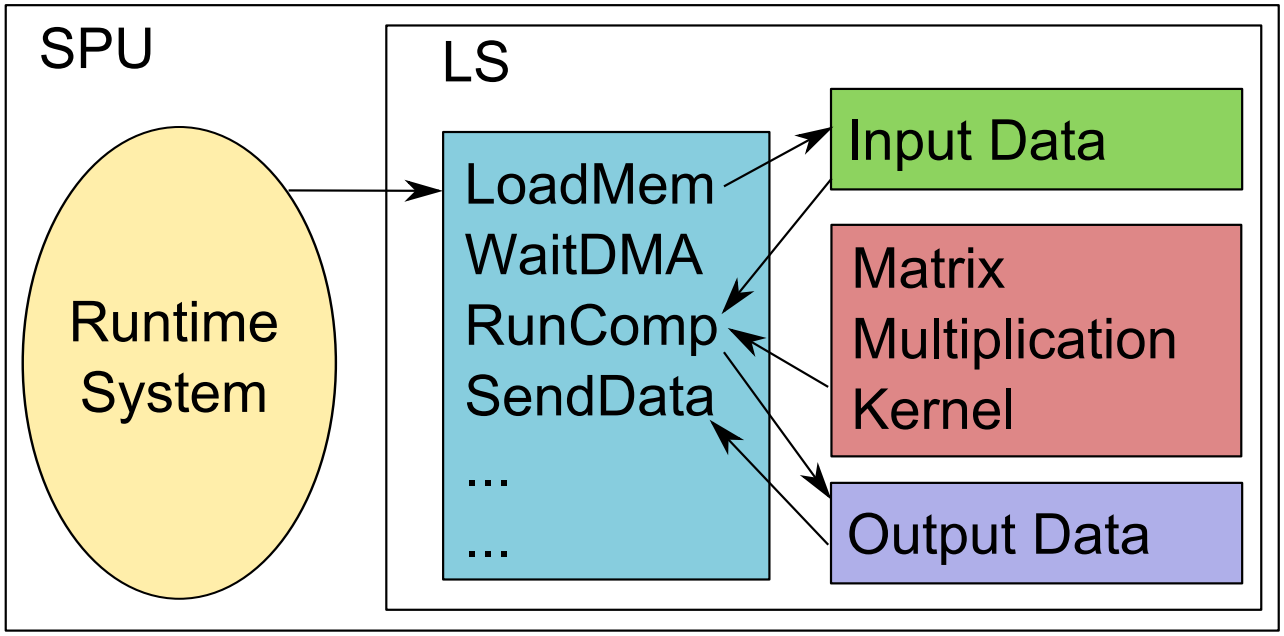


Figure: SPE Runtime thread internal view

Coconut Multicore Framework: AVOps

Main memory data transfers:

- **LoadMemory** - loads data from main memory
- **StoreMemory** - stores data to main memory
- **WaitDMA** - waits for main memory transfer completion

Inter-SPE data transfers:

- **SendData** - sends data to an SPE
- **WaitData** - waits for data from an SPE

Inter-SPE signalling:

- **SendSignal** - sends a signal to an SPE
- **WaitSignal** - waits for a signal from an SPE

Executing kernels:

- **RunComputation** - executes kernel (e.g. C function)

Why Runtime + AVOps Abstractions?

- Runtime system can be implemented on different architectures
- Separation of levels of parallelism (SIMD ILP in computational kernels vs. Process-level parallelism in AVOps code) allows developer to focus on problems independently
- AVOps *expose* important aspects of parallelism by abstracting away all but the necessary information
- Abstraction allows for linear-time static analysis of AVOps programs
- Higher-level abstractions of problems have served us well - Coconut SIMD ILP optimization techniques are included in Cell BE SDK 3.0 SPU-MASS library (4x faster than alternative SimdMath library created in C)

AVOps Expose Parallel Synchronization

AVOps Program Example

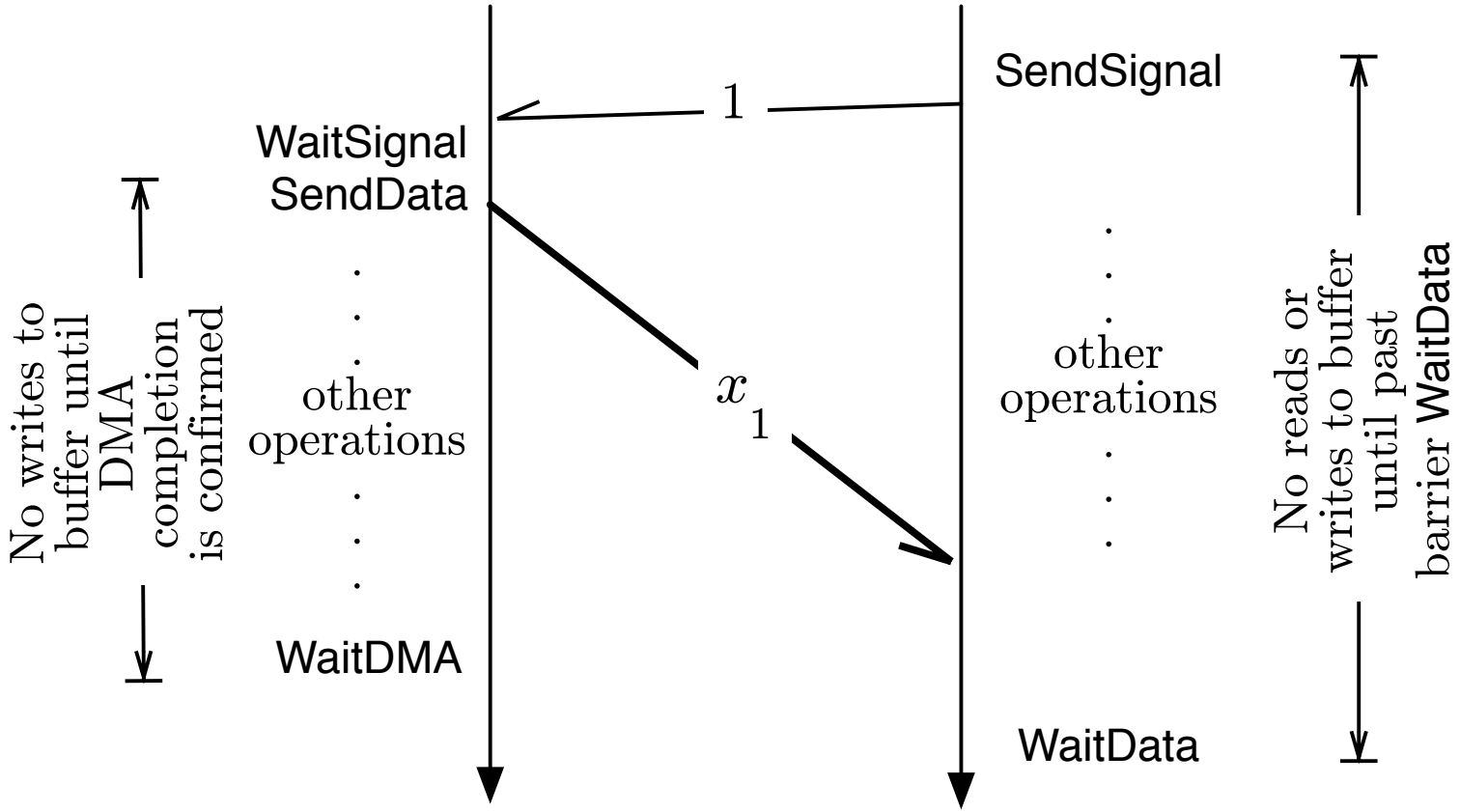
```
SPE 1, SendSignal (Signal 2) (SPE 0)
SPE 0, WaitSignal (Signal 2)
SPE 0, SendData (SPE 1) (Data Loc. Info.) (DataTag 2)
SPE 1, WaitData (DataTag 2)
```

Synchronization Bugs Exposed!

- Waits without associative Sends will deadlock system
- If multiple Sends target the same unit (either signalling or communication), without some distinction made between the associative Waits as to which is waiting for what, deadlock may occur
- Kernel execution could be attempted before Wait operation for input data block, completion unpredictable results

AVOps Expose Communication Latency

- Wait operation could occur before transfer initiated by Send is complete leading to stalls and latency
- Optimize performance by minimizing occurrences of waiting



Verification Tool Leverages Exposed Synchronization

- Exposed parallelism synchronization allows us to automatically check for parallel bugs at compile-time
- Linear-time verification tool created to verify no deadlocks exist and that results are independent of execution order
- Keeps track of state with constant space requirements
- Result: no parallel debugging required!
- Saves development costs through increased productivity
- Allows developer to focus on optimization, instead of having to make algorithm straight forward for correctness purposes (at cost of lost optimizations)
- Ensures correctness for critical tasks (e.g. MRI imaging)

Book chapter in "Process Algebra for Parallel and Distributed Processing" [AG08] defines the technique thoroughly

Key Verification Ideas Explained

- Assume program of AVOps has certain properties when presented to the algorithm as a stream of AVOps: (SPE x, OP)
- Program is *order independent* if given the same input (in main memory) all possible execution orders produce the same output (in main memory)
- A program is *locally sequential* if every (SPE_A, SendSignal SPE_B Signal_ID) is followed by a corresponding (SPE_B,WaitSignal Signal_ID), and every (SPE_A, SendData SPE_B Location Size Data_ID) is followed by corresponding (SPE_B, WaitData Data_ID) and (SPE_A,WaitDMA Data_ID) instructions.

Verification Example

AVOps with a race condition!

SP0: SendSignal (Sig 1) (SP2)
SP1: SendSignal (Sig 1) (SP2)
SP2: WaitSignal (Sig 1)
SP2: SendData (Data A) (SP0)
SP2: WaitSignal (Sig 1)
SP2: SendData (Data B) (SP0)

State

Two SPEs are signalling SPE 2 to request data, but with the same signal.

Verification Example

Let's look at how the Verification tool analyzes things.

SP0: SendSignal (Sig 1) (SP2)
SP1: SendSignal (Sig 1) (SP2)
SP2: WaitSignal (Sig 1)
SP2: SendData (Data A) (SP0)
SP2: WaitSignal (Sig 1)
SP2: SendData (Data B) (SP0)

We add the SendSignal to the State.

State

(SP2, Sig 1) $\rightarrow$ (SP0)

Verification Example

Race condition detected by checking the state.

SP0: SendSignal (Sig 1) (SP2)
SP1: SendSignal (Sig 1) (SP2)
SP2: WaitSignal (Sig 1)
SP2: SendData (Data A) (SP0)
SP2: WaitSignal (Sig 1)
SP2: SendData (Data B) (SP0)

Problem, we already have an incoming Signal 1 on SPE 2!

State

(SP2, Sig 1) $\rightarrow$ (SP0)

Verification Explained

The real trick: iteratively building a partial-order of AVOp execution from a presentation order.

SP0: SendSignal (Sig 1) (SP2)
SP2: WaitSignal (Sig 1)
SP2: SendData (Data A) (SP0)

A signal is sent from SP0 to SP2.

State

(SP2, Sig 1) $\rightarrow$ (SP0)

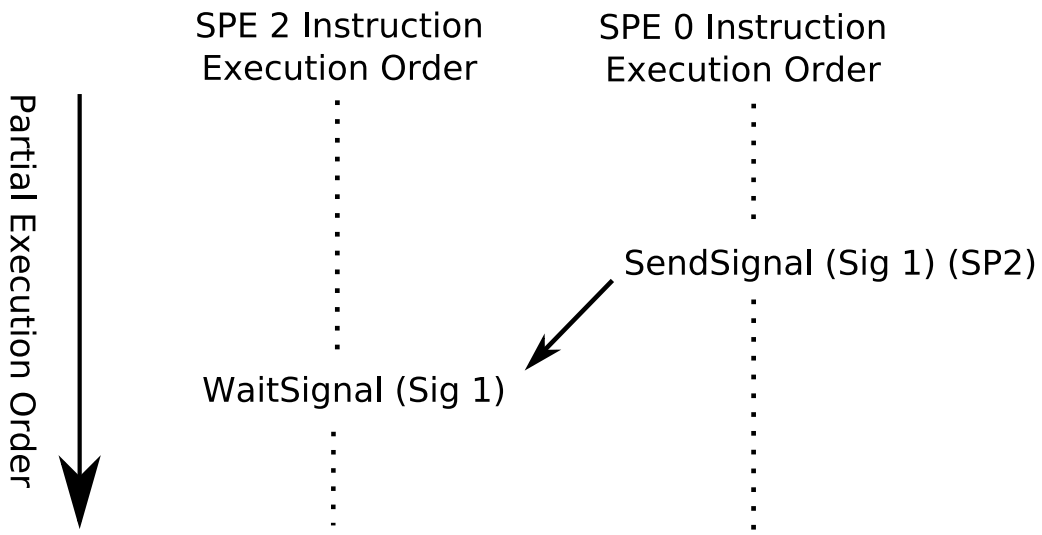
Verification Explained

We **know** that the SPE 2 WaitSignal spins until SendSignal on SPU 0 executes. Therefore all instructions on SPE 2 after the WaitSignal execute after the SPU 0 SendSignal.

SP0: SendSignal (Sig 1) (SP2)
SP2: WaitSignal (Sig 1)
SP2: SendData (Data A) (SP0)

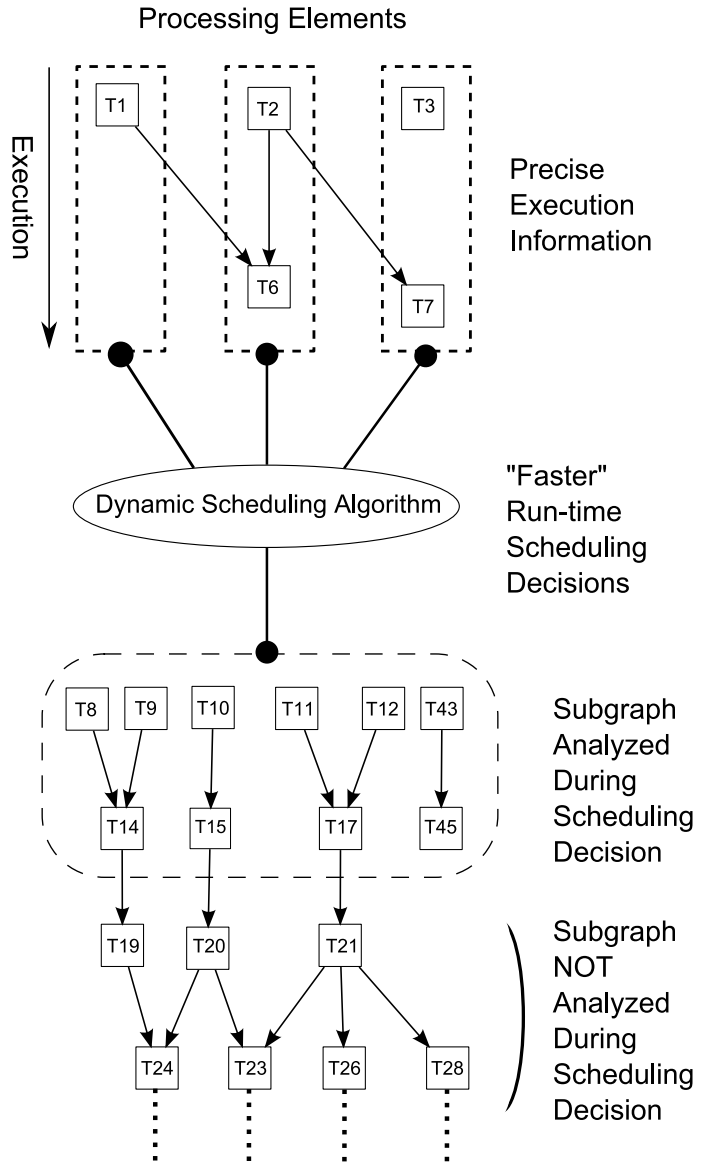
A signal from SP0 is caught on SP2.

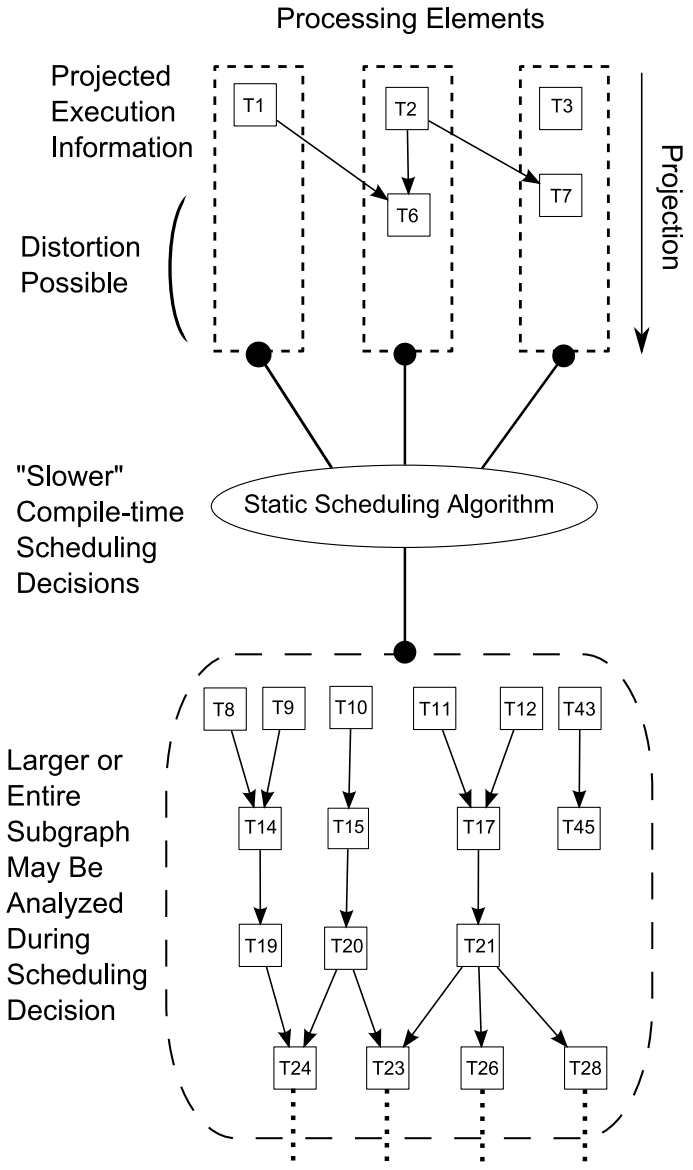
State



Why Leverage Exposed Communication Latency?

- Communication cost info critical for static scheduling [GRV04]





Communication Cost Important to Cell BE Performance

- Memory access is **"the"** performance bottleneck for most algorithms on Cell BE [VKJ+07]

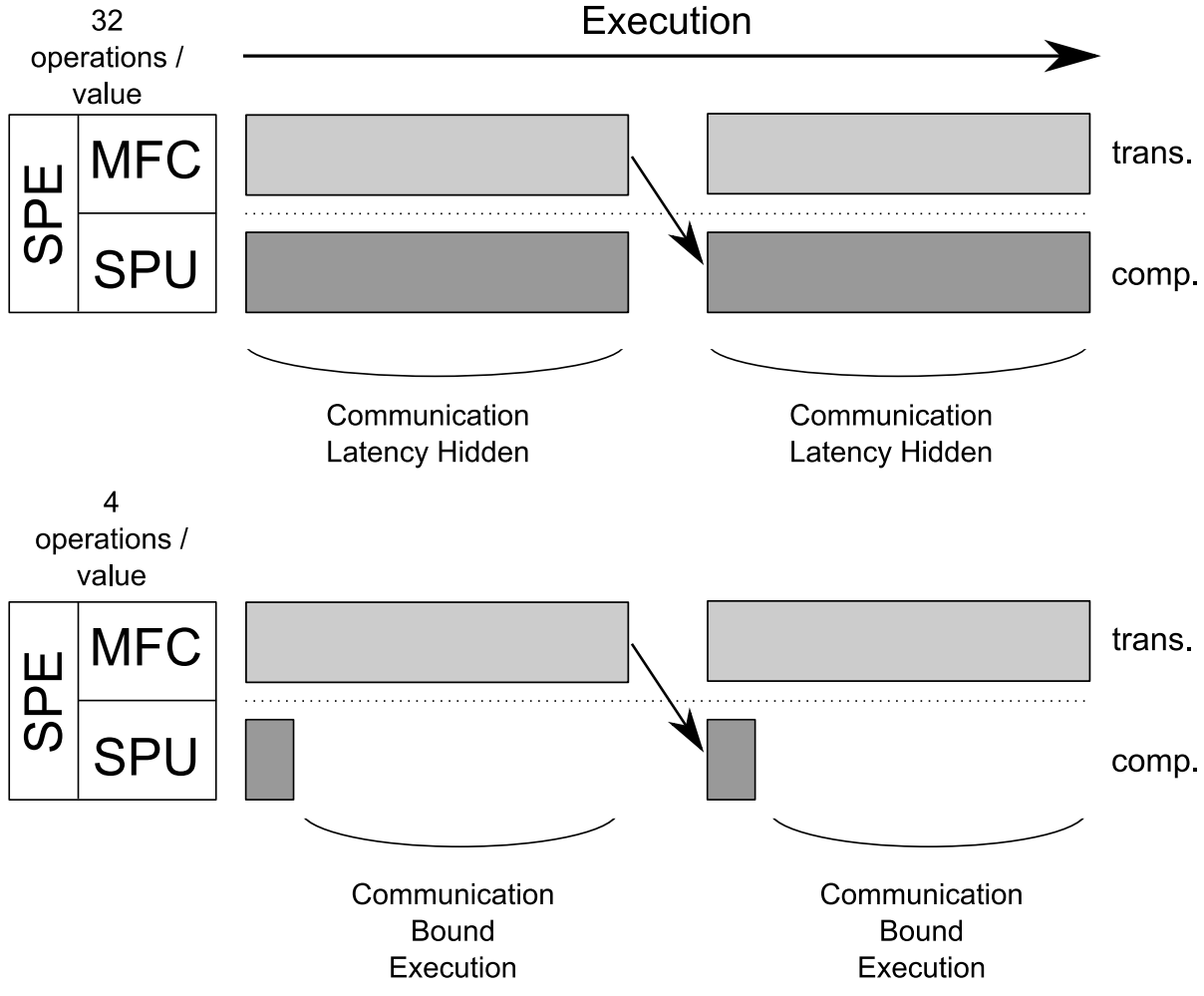


Figure: Attempt latency hiding through overlapping computation

Main Memory Bottleneck Analysis

- 25.6 GB/s combined Inbound/Outbound RAM bandwidth
- 32 operations per value required to hide latency of main memory access with computation
- Intra-SPE or inter-SPE re-use of data may alleviate bottleneck

Table: Assuming 8 SPEs request 2 64x64 input blocks each

Operations per Value	**Computation Time (ns)**	**(Transfer / Computation) Time**	**Potential Latency Hiding**
32	10,240	1	100%
16	5,120	2	50%
4	1,280	8	12.5%
1	320	32	3.125%

Processor Data Sharing as a Solution?

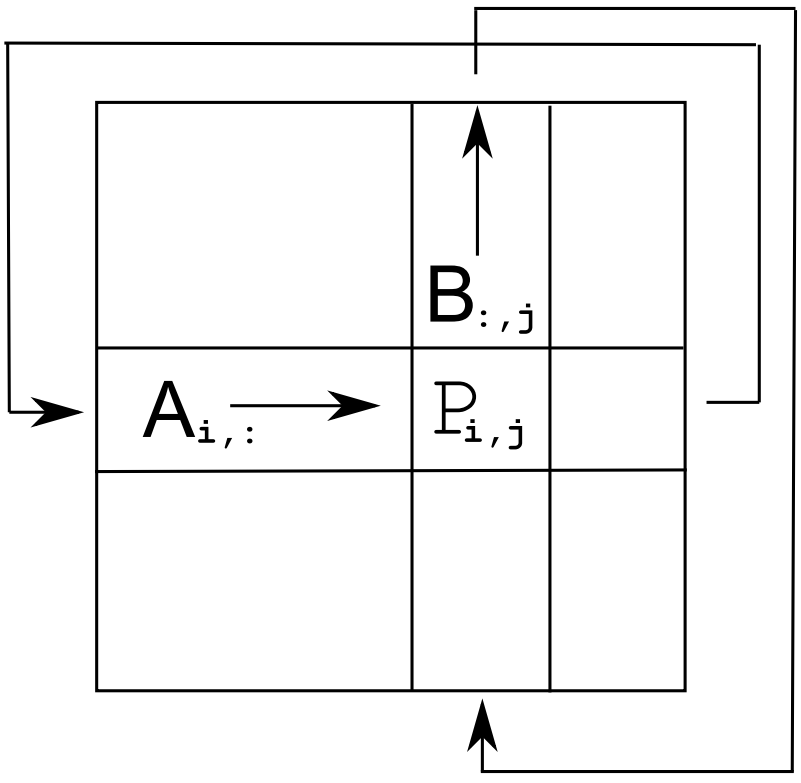


Figure: Cannon's algorithm computes block $C(i,j)$ at each processor $P(i,j)$.

- Data re-use within an SPE LS reminiscent of cache optimization
- Inter-SPE data re-use not unprecedented conceptually, used in multiprocessor algorithms (e.g. Cannon's algorithm)
- Difference with network-on-a-chip Cell BE is smaller granularity, tighter processor coupling, network design

Communication Cost Modelling on Cell BE

Single SPE Model:

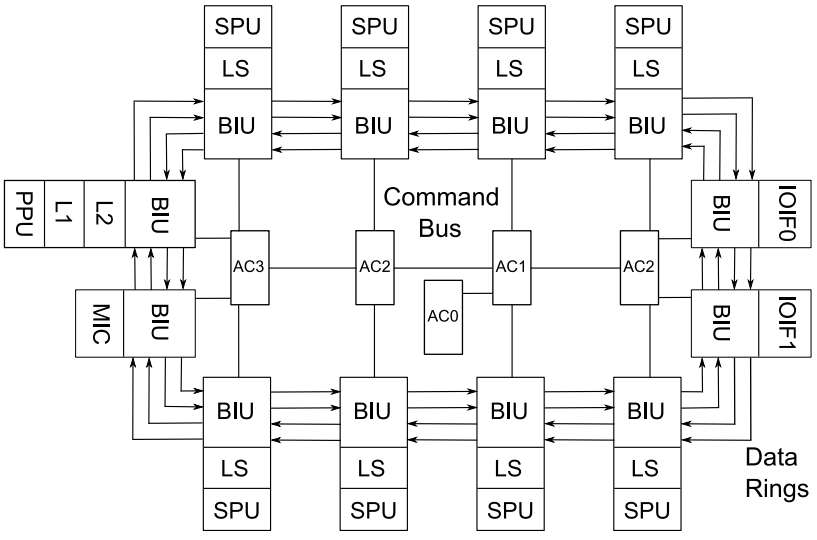
$$S + (D * b)$$

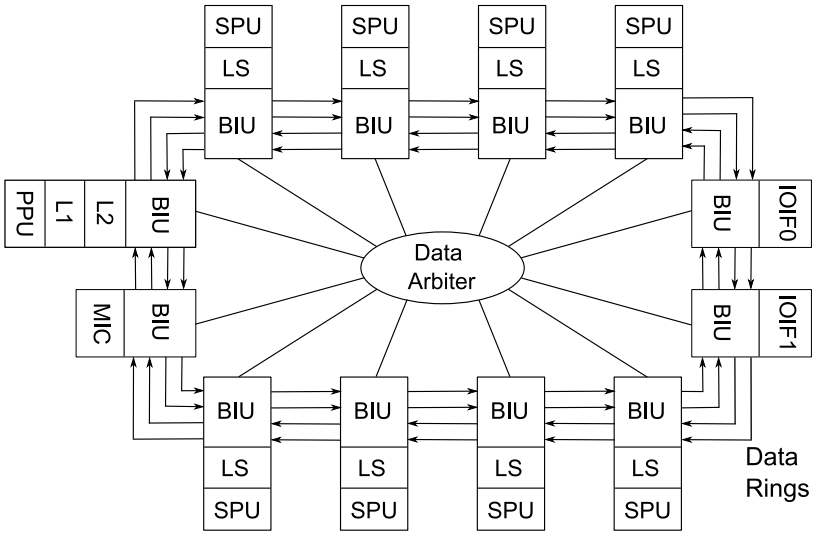
S - start-up cost
D - transfer time per byte
b - number of bytes

- Model was effectively used to optimize use of static buffers for DMA on a single SPE [CS06]
- Dividing transfer rate by number of concurrent main memory read transfers would likely be effective at extending the model to multiple SPEs
- Problem: extending model to inter-SPE communication

A Look At EIB Internals

- 4 rings for data transfer (2 clockwise, 2 counterclockwise)
- Command Bus sets up commands, handles coherency, tree-structured network
- Data Arbiter controls access to EIB rings on a per packet basis, star-structured network
- Elements connected via a Bus Interface Unit (BIU)





EIB Performance Characteristics

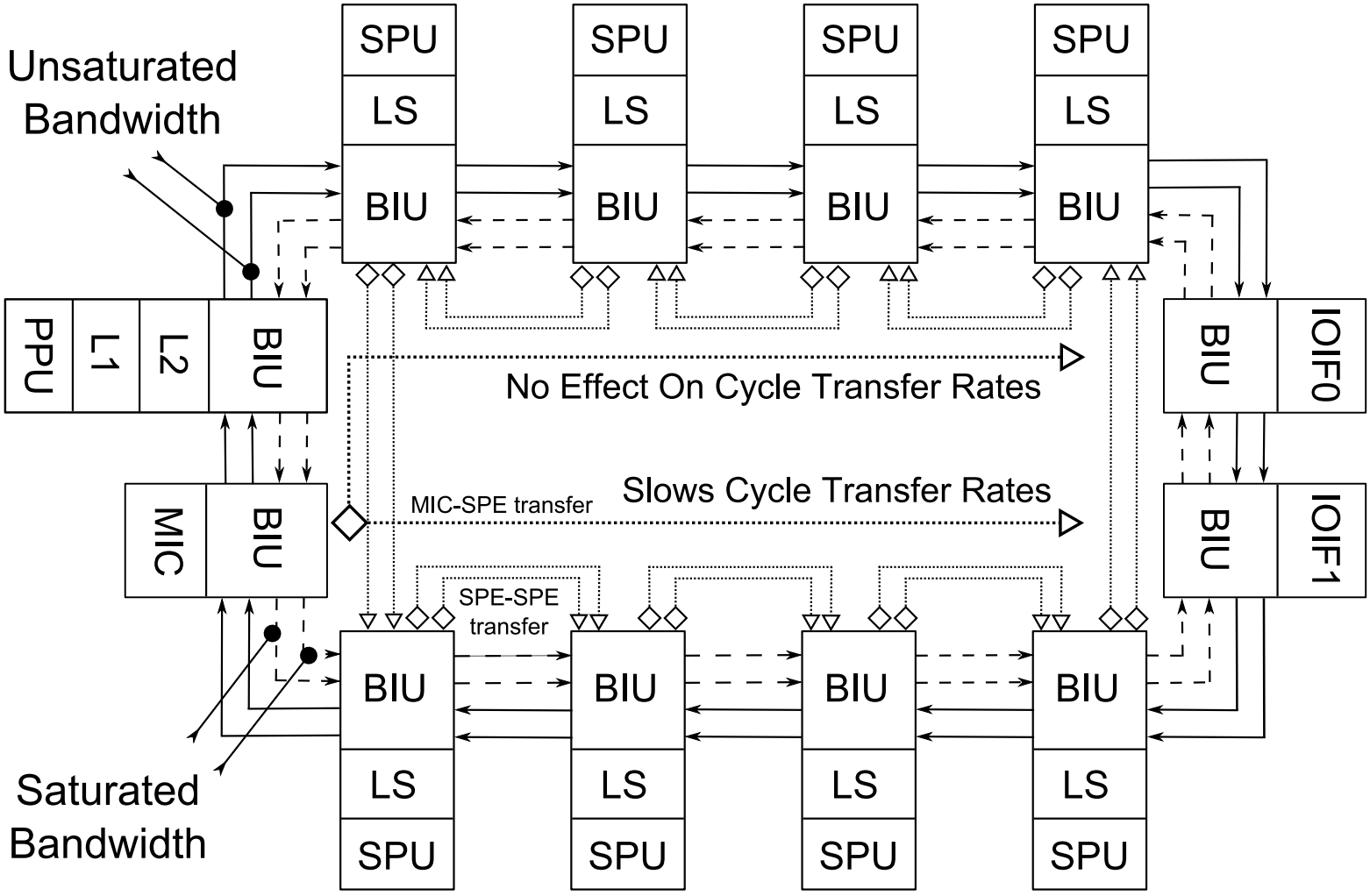
- MIC limited by 25.6 GB/s combined inbound/outbound RAM access
- MFC of SPEs support 25.6 GB/s inbound/outbound
- Each ring supports 3 concurrent non-overlapping transfers
- Eight concurrent transfers supported due to Command Bus snooping limitations, as long as paths do not overlap, limiting total bandwidth to 204.8 GB/s
- Transfers broken up into 128-byte packet size
- Packets will always take shortest distance path across data rings
- Round robin scheduling of packets
- MIC transfers have highest priority level, all others equal

Communication Pattern Bottleneck

- Inefficient communication patterns can reduce EIB bandwidth utilization to 38% of the 204.8 GB/s theoretical peak [CRDI07]
- Overlapping and lengthy (six hop) transfer paths can severely hamper EIB performance by allowing only one transfer per ring
- Individual DMA units only capable of processing so many DMAs concurrently, can be overloaded if the target or source of too many transfers
- Large DMA message sizes more efficient than many smaller message sizes
- Under heavy load, gets have better contention resolution than puts [KPP06]

Communication Pattern Bottleneck

- Circular passing of data only utilizes two rings - only 6 simultaneous transfers possible = 75% of bandwidth utilized



Inter-SPE Transfer Performance Optimization

- Communication patterns matter to algorithm performance during periods of high SPE-SPE communication and low dependence on main memory data transfers (otherwise main memory becomes bottleneck)
- Physical locations of SPEs matter to detection of inefficient communication patterns
- We don't actually know physical SPE location for SPEs assigned to us by the OS, impedes optimization of communication pattern
- However Coconut Runtime system can 'ping' SPEs at start-up to determine location

Performance Simulator

- Created to evaluate the effectiveness of different communication patterns
- Efficient linearly bounded complexity due to high-level exposure of Multicore-level parallelism and the limited possible state of the EIB
- Quick compile-time report of algorithm performance, specific occurrences and severance of latency for each Wait operation
- More efficient for development cycle time than performance profiling or simulation of binary executables

Performance Simulator Report Example

```
SPE 0 Statistics Breakdown
--------------------------

Total Instructions: 5184

Total Waits: 2080
Total Latency Waits: 1667
Total Leadency Waits: 413
Total Runtime Cost of Latency: 616.942

All Latency Wait Instructions:

Linenumber  AVOp                          Latency
----------  ----------------------------  -------
13          WaitDMA 0                     0.5144
14          WaitSignal 4                  0.2624
```

Performance Simulator Design - Different Approaches

Assume Best or Worst Case

- Cheap computation
- Information not very useful

Discrete Event Simulator

- Very expensive to compute
- Information very useful
- We don't know enough about EIB design and behaviour to model at this level, information is not in public domain

Continuous Mathematical Model

- Reasonable to compute
- Useful information
- Doesn't require information about non-public EIB design features

Performance Simulator Design - Key Idea

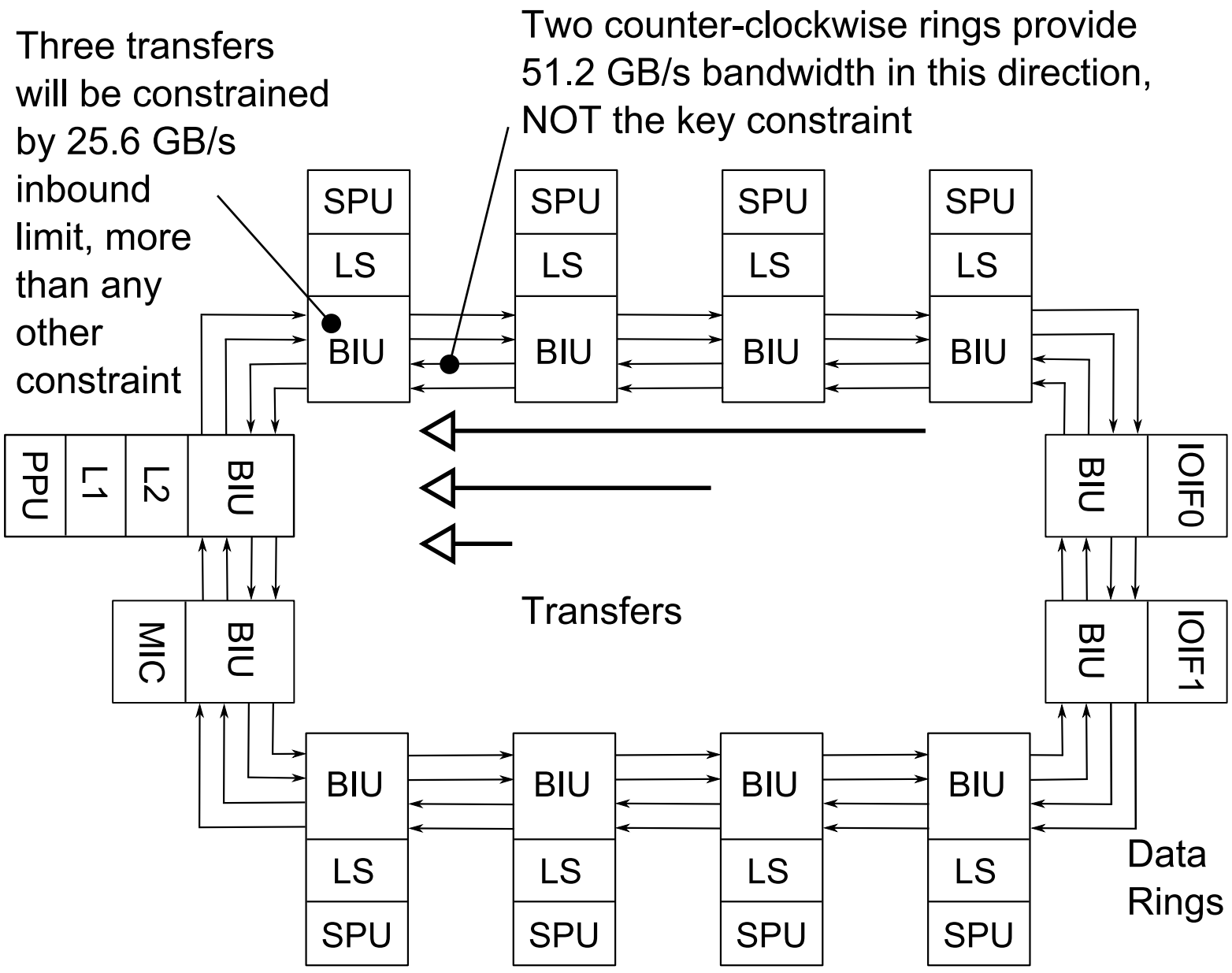
Different Bottlenecks:

- Main memory (MIC)
- Each MFC's inbound and outbound limitation
- Each direction over the ring has a 6 concurrent transfer limitation
- Transfers with overlapping paths will have to take turns using ring bandwidth

Basic Algorithm:

1. Distribute MIC bandwidth evenly to MIC transfers
2. Identify the bottleneck with the most contention
3. Distribute available bandwidth evenly to transfers associated with the bottleneck
4. Repeat 2-3 until average of remaining total EIB bandwidth is less than the worst resource contention restriction on transfers
5. Distribute remaining bandwidth evenly to remaining transfers

Worst Bottleneck Example



Performance Simulator: Passive Usage

- Performance simulation for theoretical architectures is possible
- Comparison of algorithms to contrast effectiveness of main memory bottleneck alleviation

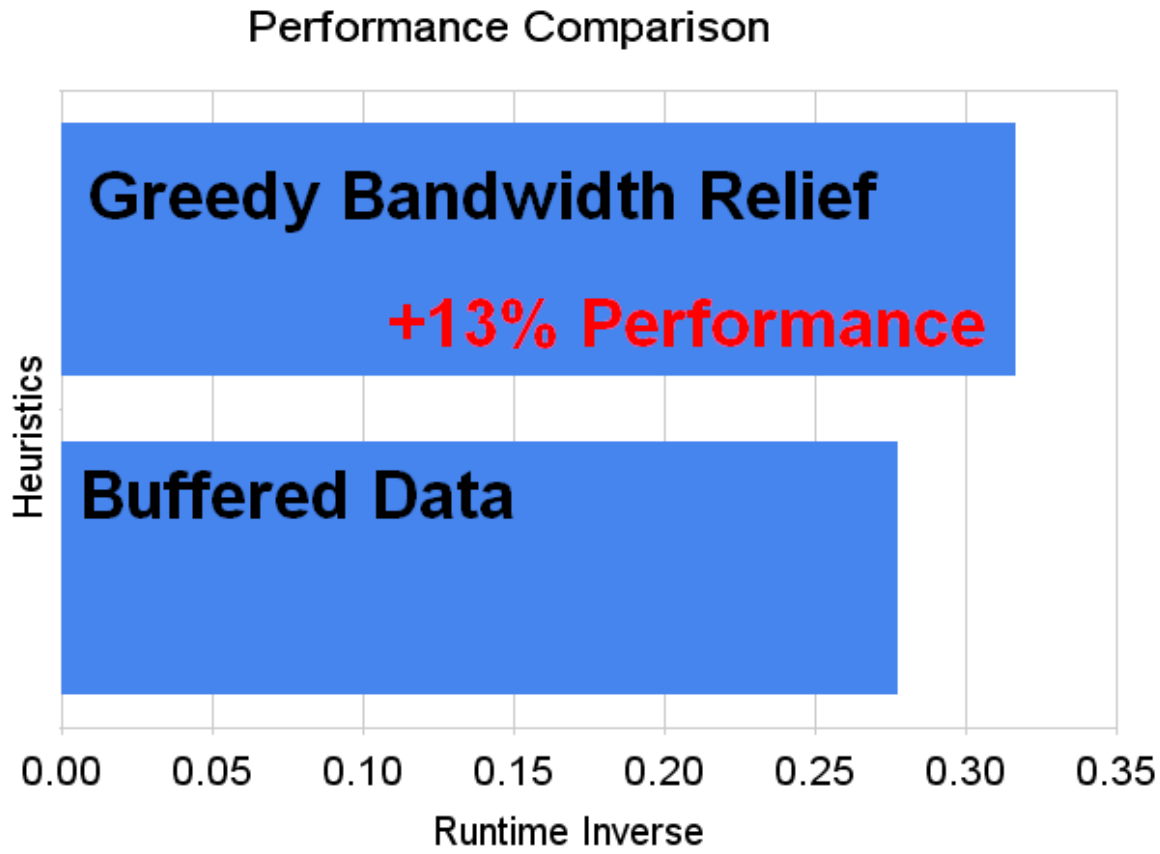
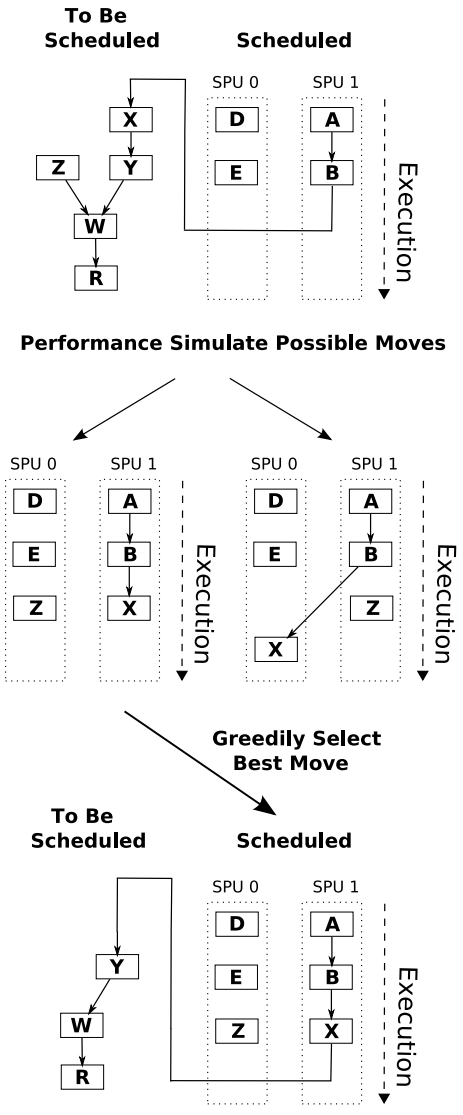


Figure: Greedy Inter-SPE data sharing for Matrix Multiplication improves performance on experimental architecture

Performance Simulator: Active Usage



- Static scheduling algorithms may be informed by detailed performance simulation when making scheduling decisions
- Performance Simulator is computationally efficient enough to make this practical at compile-time
- Simulate result of alternative scheduling decisions to determine best move

Project Status

Current Work

- Integrating performance simulation into static scheduling of tasks
- Enhancing performance simulator reporting techniques (transfer specific latency, leadency analysis)
- Tuning performance simulator against results on actual hardware
- Fence and barrier data transfers to increase EIB determinism

Future Work

- Performance simulation informed scheduling of DAGs, loops
- Applying ILP scheduling techniques to Multicore level

Thanks

- Christopher Anand (supervisor)
- Wolfram Kahl (co-supervisor)
- Gabriel Grant (implemented Runtime system)
- Shiqi Cao (helped implement Coconut components)

Legal Notices

- Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc., in the United States, other countries, or both.
- IBM is a registered trademark of International Business Machines Corporation in the United States, other countries, or both.
- Other company, product, and service names may be trademarks or service marks of others.


Michael Alexander and William Gardner, editors.
Process Algebra for Parallel and Distributed Processing: Algebraic Languages in Specification-Based Software Development.
Chapman and Hall, 2008.
(to appear).


Brian Bouzas, Robert Cooper, Jon Greene, Michael Pepe, and Myra Jean Prelle.
Multicore framework: An api for programming heterogeneous multicore processors.
Technical report, Mercury Computer Systems, Inc. Technical Report, 2006.


Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta.
CellSs: A programming model for the Cell BE architecture.
IEEE, November 2006.


Catherine H. Crawford, Paul Henning, Michael Kistler, and Cornell Wright.

Accelerating computing with the cell broadband engine processor.
In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 3–12, New York, NY, USA, 2008. ACM.


T. Chen, R. Raghavan, J. N. Dale, and E. Iwata.
Cell broadband engine architecture and its first implementation: a performance view.
IBM J. Res. Dev., 51(5):559–572, 2007.


Tong Chen and Zehra Sura.
Optimizing the use of static buffers for dma on a cell chip.
In *In The 19th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2006*, 2006.


A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo.
Using advanced compiler technology to exploit the performance of the cell broadband enginetm architecture.

IBM Syst. J., 45(1):59–84, 2006.


Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan.
Sequoia: programming the memory hierarchy.
In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83, New York, NY, USA, 2006. ACM.


Arnaud Giersch, Yves Robert, and Frederic Vivien.
Scheduling tasks sharing files from distributed repositories.
In *In Euro-Par 2004: Parallel Processing: 10th International Euro-Par Conference, volume 3149 of Lecture Notes in Computer Science*, pages 246–253. Springer-Verlag, 2004.


Linda Kaouane, Dominique Houzet, and Sylvain Huet.
Syscellc: Systemc on cell.
In *ICCSA '08: Proceedings of the 2008 International Conference on Computational Sciences and Its Applications*,


Michael Kistler, Michael Perrone, and Fabrizio Petrini.

IEEE Micro, 26(3):10–23, 2006.


C. Mueller, B. Martin, and A. Lumsdaine.

Technical report, In Proc. 1st STI/Georgia Tech Workshop on Software and Applications for the Cell/B.E. Processor, 2007.


A. Munshi.


Michael D. McCool, Kevin Wadleigh, Brent Henderson, and Hsin-Ying Lin.


M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani.

IBM Syst. J., 45(1):85–102, 2006.


Matthew Papakipos.


Scott Schneider, Jae-Seung Yeom, Benjamin Rose, John C. Linford, Adrian Sandu, and Dimitrios S. Nikolopoulos.


M. K. Velamati1, A. Kumar, N. Jayam, N. G. Senthilkumar, P. K. Baruah, R. Sharma, S. Kapoor, and A. Srinivasan.

David Zhang, Qiuyuan J. Li, Rodric Rabbah, and Saman Amarasinghe.

SIGARCH Comput. Archit. News, 36(2):18–27, 2008.