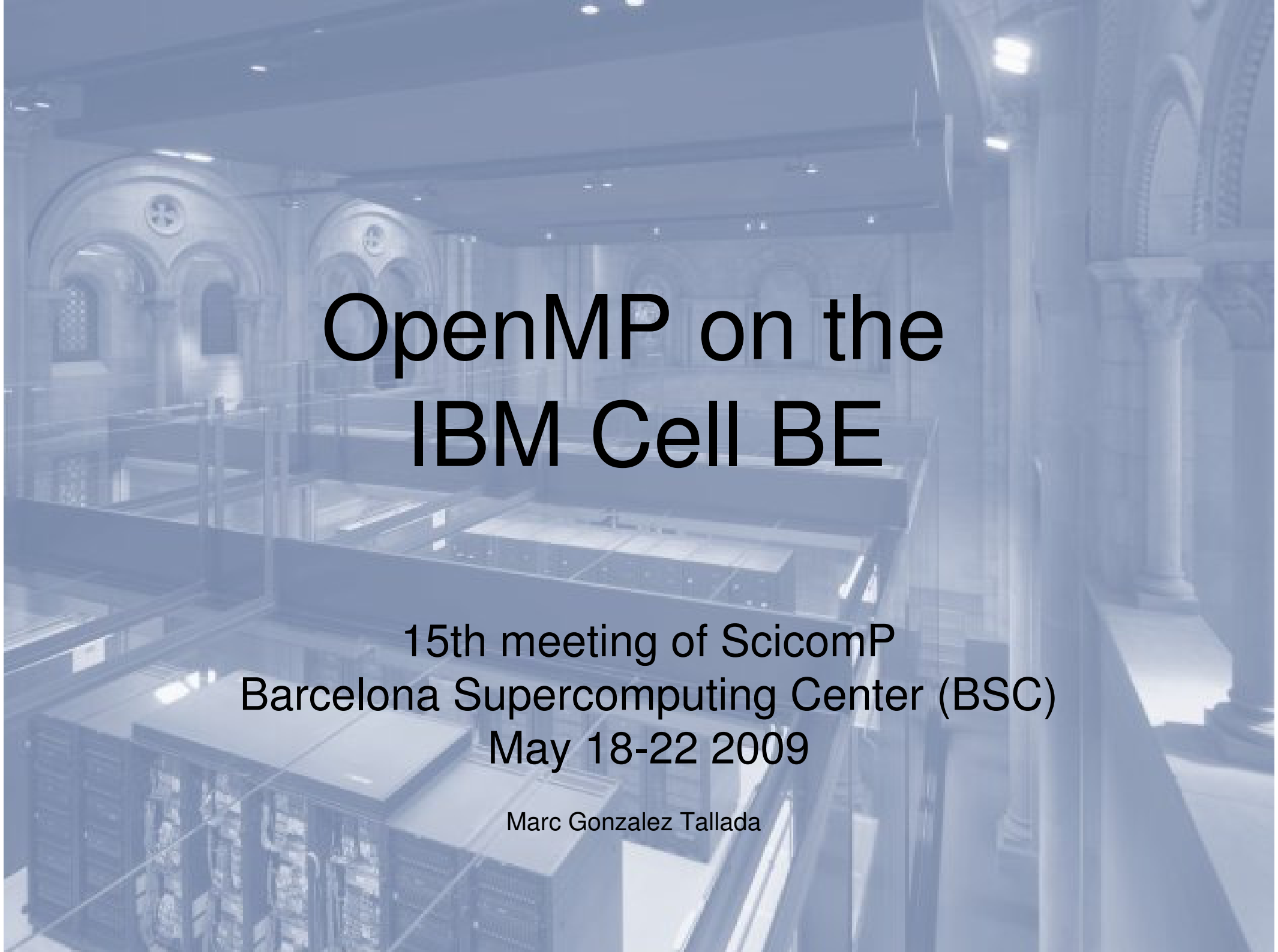


OpenMP on the IBM Cell BE

15th meeting of ScicomP
Barcelona Supercomputing Center (BSC)
May 18-22 2009

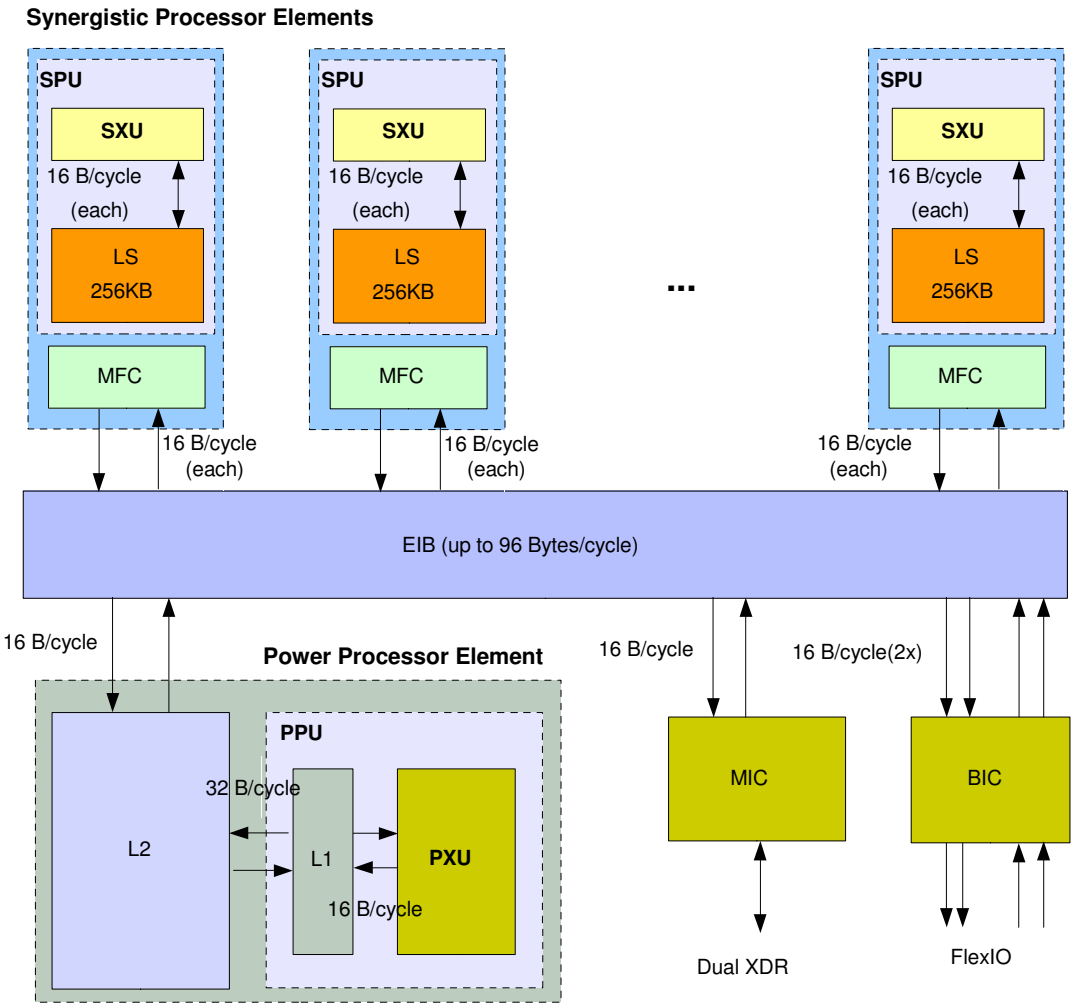
Marc Gonzalez Tallada

Index

- OpenMP programming and code transformations
  - Tiling and Software cache transformations
  - Sources of overheads
- Performance
  - Loop level parallelism
  - Double buffer
  - Combine OpenMP and SIMD parallelism

Introduction

- The Cell BE Architecture – multi core design that mixes two architectures
  - One core based on Power PC architecture (PPE)
  - Eight cores based on the Synergistic Processor Element (SPE).
- SPEs are provided with local stores
  - Load and Store instruction in SPE can address only Local Store
  - Data transfer to/from main memory is explicitly performed under software control.



Cell programmability

- Transform original code
  - Allocate buffers in the local store
  - Introduce DMA operations within the code
  - Synchronizatio ts
  - Translate ress space space
- T
  - V ptimized codes but at cost of **programmability**
    - Manual SIMD coding
    - Overlap of communication with computation

PERFORMANCE but not PROGRAMMABILITY

- **Automatic** solution
  - Tiling, Double Buffer
    - Good sol egular applic
    - A erable compile time
      ache
    - sually **performance** is limited to the available information at compile time
    - Very difficult to generate code that overlaps computation with communication

PROGRAMMABILITY but not PERFORMANCE

Can the Cell BE be programmed as a cache-based multi-core ?

- OpenMP programming model
  - Parallel region
  - Variable scoping
    - PRIVATE, SHARED, THREADPRIVATE ...
  - Worksharing constructs
    - DO, SECTIONS, SINGLE
  - Synchronization constructs
    - CRITICAL, BARRIER, ATOMIC
  - Memory consistency
    - FLUSH

```
#pragma omp parallel private(c,i) shared(a, b, d)
{
for (i=0; i<N; i++) c[i]= ...;
...
#pragma omp for scheduling(static) reduction(s)
for (i=0; i<N; i++) {
  a[i] = c[b[i]] + d[i];
  s = s + a[i];
}
...
#pragma omp barrier

...

#pragma omp critical
{
  s = s + c[0];
}
}
```

Hardware does not impose any restriction to the model!

IBM Cell BE can be programmed as a cache based multi-core

Main problem to solve ...

- Transform original code
  - Allocate buffers in the local store
  - Introduce DMA operations within the code
  - Synchronization statements
  - Translate from original address space to local address space

- Compile-time predictable access
  - a[i], d[i], b[i], s
- Unpredictable access
  - c[b[i]]

```
#pragma omp parallel private(c,i) shared(a, b, d)
{
for (i=0; i<N; i++) c[i]= ...;
...
#pragma omp for scheduling(static) reduction(+:s)
for (i=0; i<N; i++) {
  a[i] = c[b[i]] + d[i];
  s = s + a[i];
}
...
#pragma omp barrier

...

#pragma omp critical
{
  s = s + c[0];
}

}
```

Software Cache + Tiling techniques

Introduction

- Code transformation: poor information at compile-time
  - Only software cache

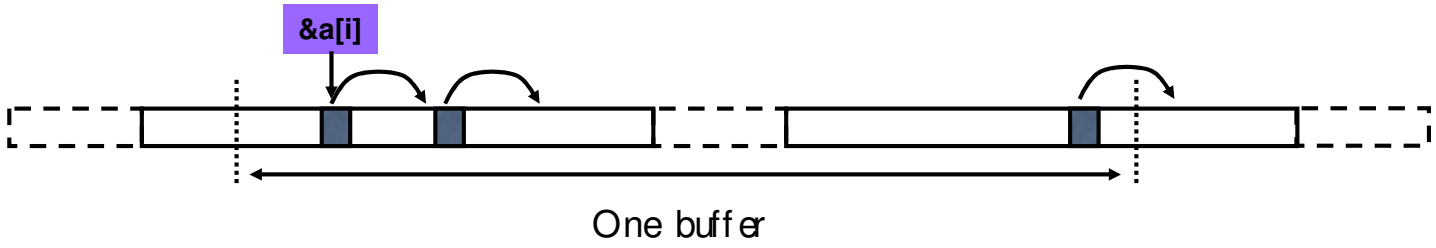
```
#pragma omp for scheduling(static) reduction(+:s)
for (i=0; i<N; i++) {
  a[i] = c[b[i]] + d[i];
  s = s + a[i];
}
```

```
...
tmp_s = 0.0;
for (i=start;i<end;i++){
 if (!HIT(h1, &d[i]))
   MAP(h4, &d[i]);
 if (!HIT(h2, &b[i]))
   MAP(h2, &b[i]);
 tmp01 = REF(h1, &d[i]);
 tmp02 = REF(h2, &b[i]);
 if (!HIT(h4, &c[tmp02]))
   MAP(h4, &c[tmp02]);
 tmp03 = REF(h4, &c[tmp02]);
 if (!HIT(h3, &a[i]))
   MAP(h3, &a[i]);
 REF(h3, &a[i])=tmp03 + tmp01;
 tmp_s = tmp_s + REF(h3, &a[i]);
}
atomic_add(s, tmp_s, ...);
omp_barrier();
```

Memory handler (h?): contains pointer to buffer in local store
HIT: executes cache lookup, updates memory handler
REF: performs address translation and actual memory access

For strided memory references

- Enable compiler optimizations for memory references that expose a strided access pattern
  - Execute control code at buffer level, not at every memory instance
  - Maximize the overlap between computation and communication
  - Try to compute the number of iterations that can be executed before needing to change buffer



Hybrid code transformation

- Organize the LS in two storages:
  - Predictale access
  - Software cache for unpredictable access

```
#pragma omp for scheduling(static) reduction(+:s)
for (i=0; i<N; i++) {
  a[i] = c[b[i]] + d[i];
  s = s + a[i];
}
```

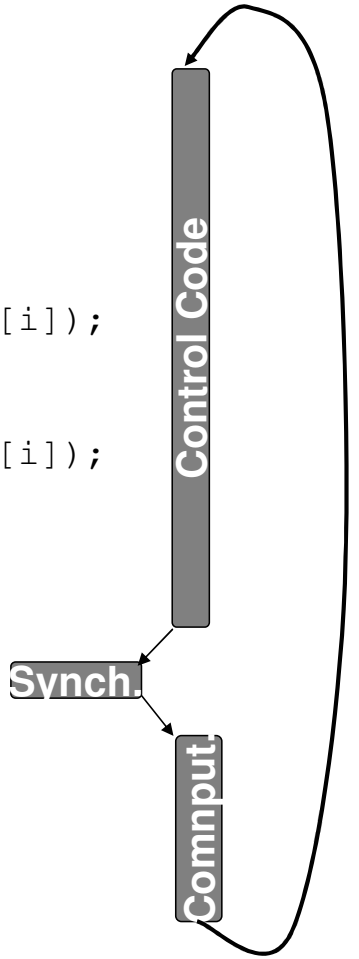
```
...
for (i=start;i<n;i++){
    tmp01 = REF(h1, &d[i]);
    tmp02 = REF(h2, &b[i]);
    if (!HIT(h4, &c[tmp02]))
      MAP(h4, &c[tmp02]);
    tmp03 = REF(h4, &c[tmp02]);
    REF(h3, &a[i])=tmp03 + tmp01;
    tmp_s = tmp_s + REF(h3, &a[i]);
}
```

```
...
tmp_s = 0.0;
i=start;
while (i< end){
  n = end;

  if (!AVAIL(h1, &d[i]))
    MMAP(h1, &d[i]);
  n = min(n, i+AVAIL(h1, &d[i]);
  if (!AVAIL(h2, &b[i]))
    MMAP(h2, &b[i]);
  n = min(n, i+AVAIL(h2, &b[i]);
  if (!AVAIL(h3, &a[i]))
    MMAP(h3, &a[i]);
  n = min(n, i+AVAIL(h3, &a[i]);

  HCONSISTENCY(n, h3);
  HSYNC(h1, h2, h3);
  start = i;
  for (i=start;i<n;i++){
    ...
  }
}
atomic_add(s, tmp_s, ...);
omp_barrier();
```

Execution model

- Loops execute in three different phases
  - Control code
    - Allocate buffers
    - Program DMA transfers
    - Consistency
  - Synchronize with DMA
  - Execute a burst of computation
    - Might include some control code, DMA programming and synchronization

```
...
tmp_s = 0.0;
i=0;
while (i< upper_bound){
  n = N;

  if (!AVAIL(h1, &d[i]))
    MMAP(h1, &d[i]);
  n = min(n, i+AVAIL(h1, &d[i]);
  if (!AVAIL(h2, &b[i]))
    MMAP(h2, &b[i]);
  n = min(n, i+AVAIL(h2, &b[i]);
  if (!AVAIL(h3, &a[i]))
    MMAP(h3, &a[i]);

  HCONSISTENCY(n, h3);
  HSYNC(h1, h2, h3);
  start = i;
  for (i=start;i<n;i++){
     ...

  }
}
atomic_add(s, tmp_s, ...);
omp_barrier();
```



Compiler limitations: Memory alias

- Compiler limitations
  - What if *a,b,c* or *d* are memory alias ?
    - How to allocate buffers consistently ?
  - What if some element in a buffer is also referenced through the software cache ?
  - Memory aliasing
    - Avoid pointer usage
    - Avoid function calls: use inline annotations

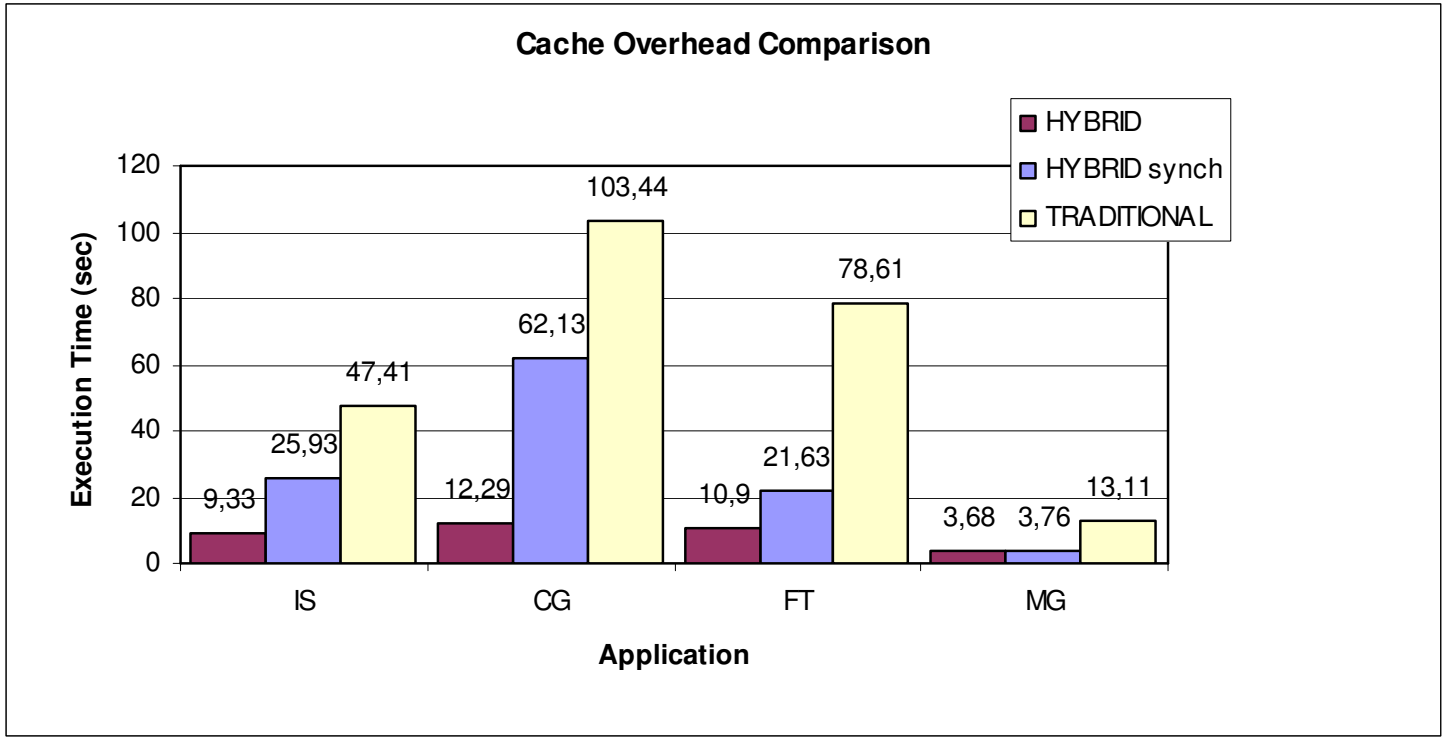
```
#pragma omp parallel private(c,i) shared(a, b, d)
{
for (i=0; i<N; i++) c[i]= ...;
...
#pragma omp for scheduling(static) reduction(+:s)
for (i=0; i<N; i++) {
  a[i] = c[b[i]] + d[i] + ...;
  s = s + a[i];
}
...
#pragma omp barrier

...

#pragma omp critical section
{
  s = s + c[0];
}

}
```

Memory Consistency

- Maintain a relaxed consistency model according to the OpenMP memory model
- Based on Atomicity and Dirty bits
- When data in a buffer has to be evicted, the write-back process is composed by three steps:
  1. Atomic Read
  2. Merge
  3. Atomic Write

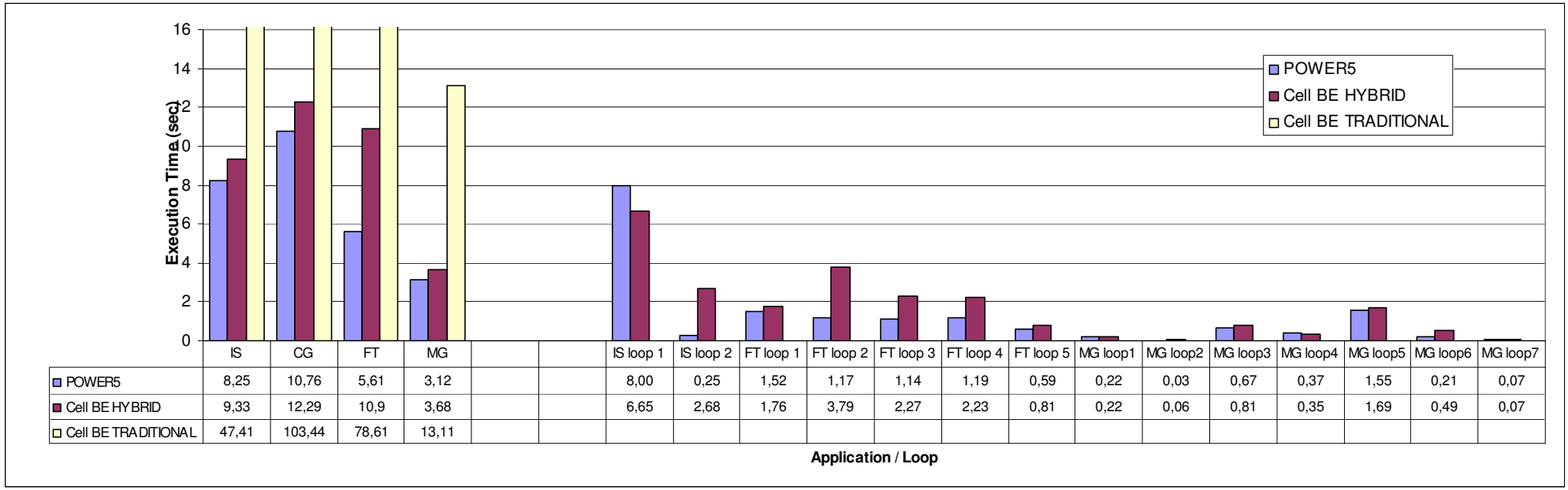
Evaluation

- Comparison to a traditional software cache
  - 4-way, 128-byte cache line, 64KB of capacity
  - Write-back implemented through Dirty-Bits and atomic (synchronous) data transfers



Evaluation: Comparing performance with Power 5

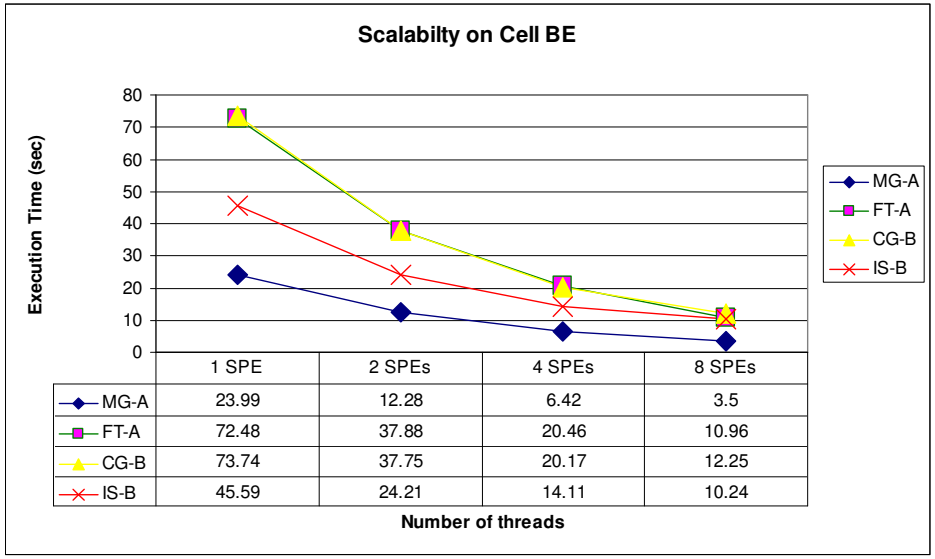
- POWER5-based blade with two processors running at 1.5 GHz
  - 16 GB of memory (8 GB each processor)
  - Each processor
    - 2 core with 2 SMT threads each
    - Shared 1,8 MB L2



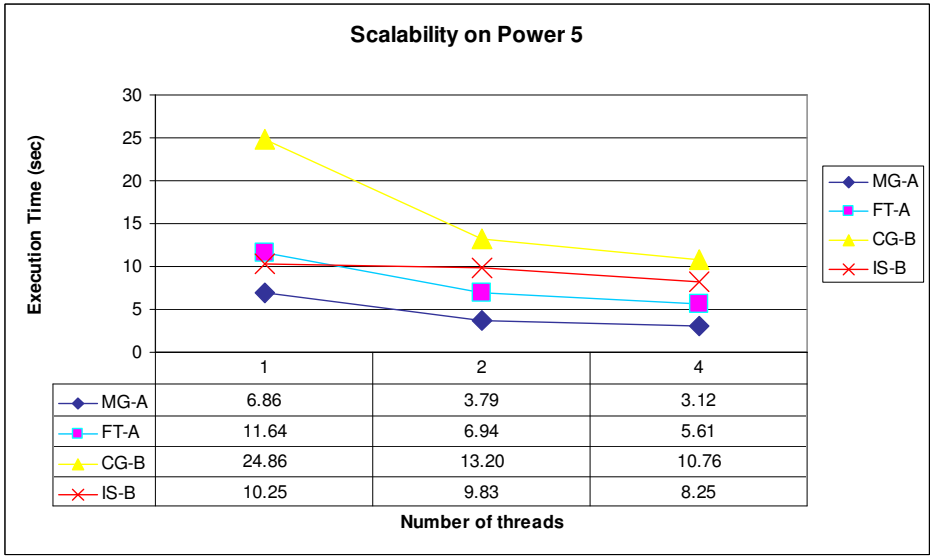
	IS	CG	FT	MG			IS loop 1	IS loop 2	FT loop 1	FT loop 2	FT loop 3	FT loop 4	FT loop 5	MG loop1	MG loop2	MG loop3	MG loop4	MG loop5	MG loop6	MG loop7
POWER5	8,25	10,76	5,61	3,12			8,00	0,25	1,52	1,17	1,14	1,19	0,59	0,22	0,03	0,67	0,37	1,55	0,21	0,07
Cell BE HYBRID	9,33	12,29	10,9	3,68			6,65	2,68	1,76	3,79	2,27	2,23	0,81	0,22	0,06	0,81	0,35	1,69	0,49	0,07
Cell BE TRADITIONAL	47,41	103,44	78,61	13,11																

Evaluation: Scalability

- Cell BE versus Power5



	1 SPE	2 SPEs	4 SPEs	8 SPEs
MG-A	23.99	12.28	6.42	3.5
FT-A	72.48	37.88	20.46	10.96
CG-B	73.74	37.75	20.17	12.25
IS-B	45.59	24.21	14.11	10.24



	1	2	4
MG-A	6.86	3.79	3.12
FT-A	11.64	6.94	5.61
CG-B	24.86	13.20	10.76
IS-B	10.25	9.83	8.25

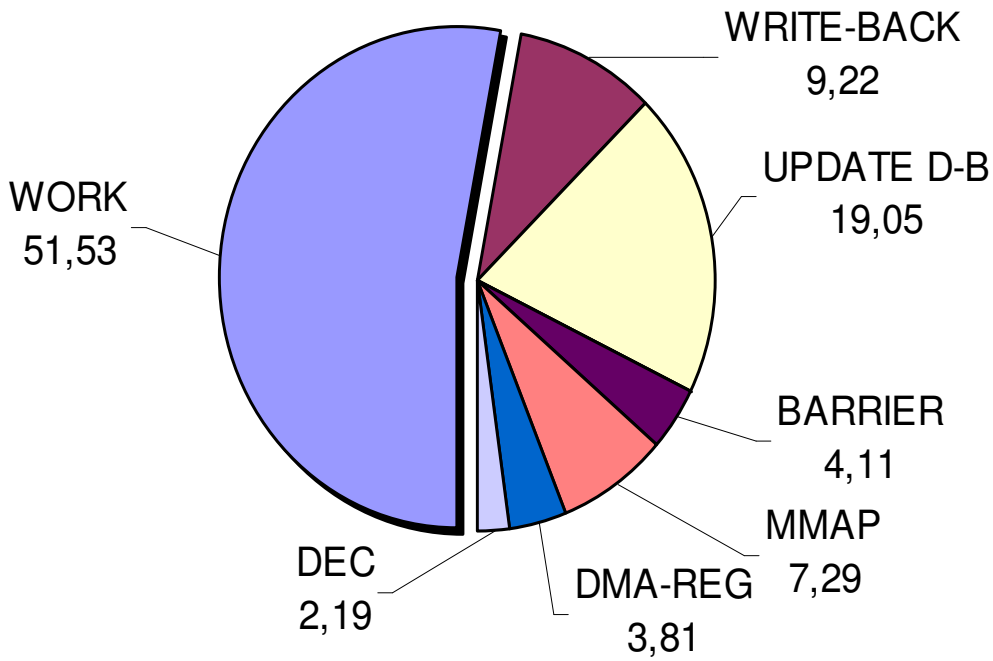
Runtime activity

- Number of iterations per runtime intervention
  - Buffer size: 4KB

MG	2 SPE					4 SPEs					8 SPEs				
	kernel	iters cnt	4KB buffer	TRANFER	ITERATIONS	kernel	iters cnt	4KB buffer	TRANFER	ITERATIONS	kernel	iters cnt	4KB buffer	TRANFER	ITERATIONS
1	213310272	2788302	17025381	6,11	76,50	106655136	1393961	8511458	6,11	76,51	53327648	696943	4255404	6,11	76,52
2	95494660	1401200	8842580	6,31	68,15	47747270	700650	4421740	6,31	68,15	23873710	350385	2211260	6,31	68,14
3	33554432	196096	196096	1,00	171,11	16777216	98048	98048	1,00	171,11	8388608	49024	49024	1,00	171,11
4	786412	8098	32392	4,00	97,11	393216	4032	16128	4,00	97,52	196648	2026	8104	4,00	97,06
5	795076	7741	30964	4,00	102,71	401860	3886	15544	4,00	103,41	205232	2005	8020	4,00	102,36
CG	**2 SPE**					**4 SPEs**					**8 SPEs**				
	kernel	iters cnt	4KB buffer	TRANFER	ITERATIONS	kernel	iters cnt	4KB buffer	TRANFER	ITERATIONS	kernel	iters cnt	4KB buffer	TRANFER	ITERATIONS
1	225000	444	2664	6,00	506,76	112500	222	1332	6,00	506,76	56244	114	684	6,00	493,37
2	5624700	11100	11100	1,00	506,73	2812200	5550	5550	1,00	506,70	1406100	2850	2850	1,00	493,37
3	5624700	11100	22200	2,00	506,73	2812200	5550	11100	2,00	506,70	1406100	2850	5700	2,00	493,37
4	224988	444	888	2,00	506,73	112488	222	444	2,00	506,70	56244	114	228	2,00	493,37
5	224988	444	444	1,00	506,73	112488	222	222	1,00	506,70	56244	114	114	1,00	493,37
6	5624700	11100	44400	4,00	506,73	2812200	5550	22200	4,00	506,70	1406100	2850	11400	4,00	493,37
7	187490	370	740	2,00	506,73	93740	185	370	2,00	506,70	46870	95	190	2,00	493,37
8	187490	370	740	2,00	506,73	93740	185	370	2,00	506,70	46870	95	190	2,00	493,37
FT	**2 SPE**					**4 SPEs**					**8 SPEs**				
	kernel	iters cnt	4KB buffer	TRANFER	ITERATIONS	kernel	iters cnt	4KB buffer	TRANFER	ITERATIONS	kernel	iters cnt	4KB buffer	TRANFER	ITERATIONS
1	134217728	524288	4194304	8,00	256,00	67108864	262144	2097152	8,00	256,00	33554432	131072	1048576	8,00	256,00
2	134217728	524288	4194304	8,00	256,00	67108864	262144	2097152	8,00	256,00	33554432	131072	1048576	8,00	256,00
3	117440512	458752	3670016	8,00	256,00	58720256	229376	1835008	8,00	256,00	29360128	114688	917504	8,00	256,00
IS	**2 SPE**					**4 SPEs**					**8 SPEs**				
	kernel	iters cnt	4KB buffer	TRANFER	ITERATIONS	kernel	iters cnt	4KB buffer	TRANFER	ITERATIONS	kernel	iters cnt	4KB buffer	TRANFER	ITERATIONS
1	11534336	11264	11264	1,00	1024,00	5767168	5632	5632	1,00	1024,00	2883584	2816	2816	1,00	1024,00
2	23068672	22528	22528	1,00	1024,00	23068672	22528	22528	1,00	1024,00	23068672	22528	22528	1,00	1024,00

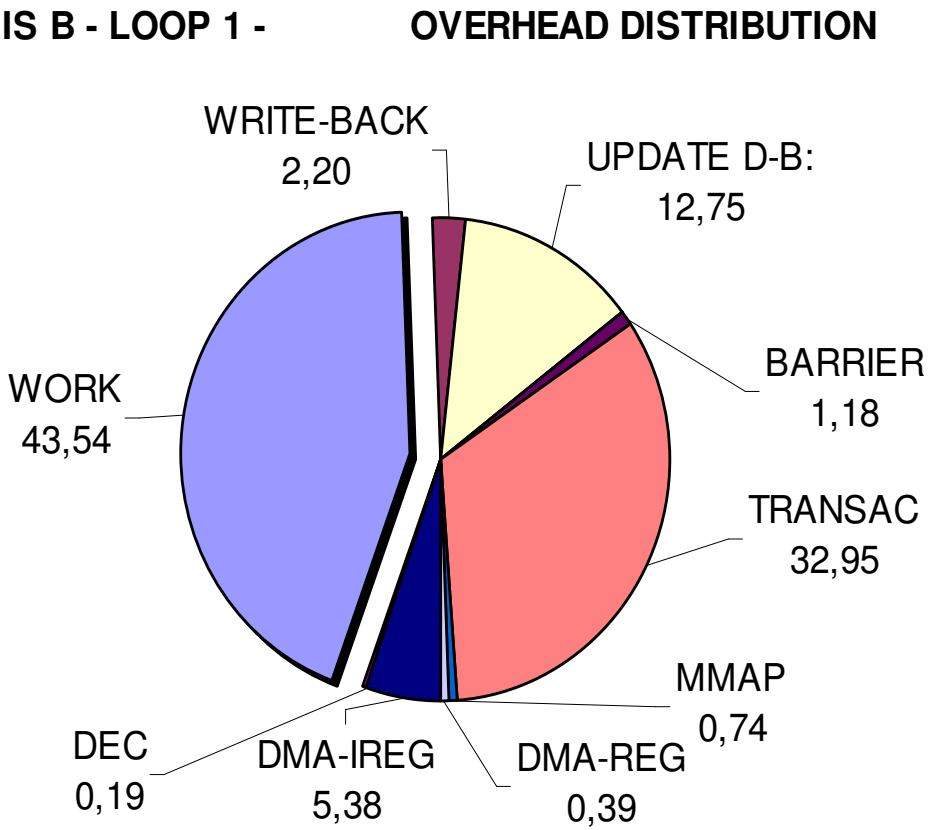
Evaluation: Overhead Distribution

MG A - LOOP 5 - OVERHEAD DISTRIBUTION



WORK: time spent in actual computation.
WRITE-BACK: time spent in the write-back process.
UPDATE D-B: time spent updating the dirty-bits information.
DMA-IREG: time spent synchronizing with the DMA data transfers in the TC.
DMA-REG: time spent synchronizing with the DMA data transfers in the HLC.
DEC: time spent in the pinning mechanism for cache lines.
TRANSAC: time spent executing control code of the TC.
BARRIER: time spent in the barrier synchronization at end of parallel computation.
MMAP: time spent in executing look-up, placement/replacement actions and DMA programming.

Evaluation: Overhead Distribution

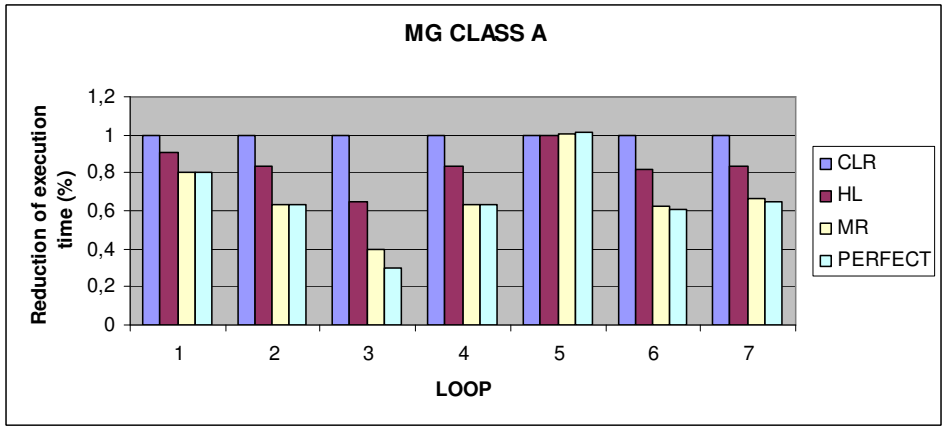


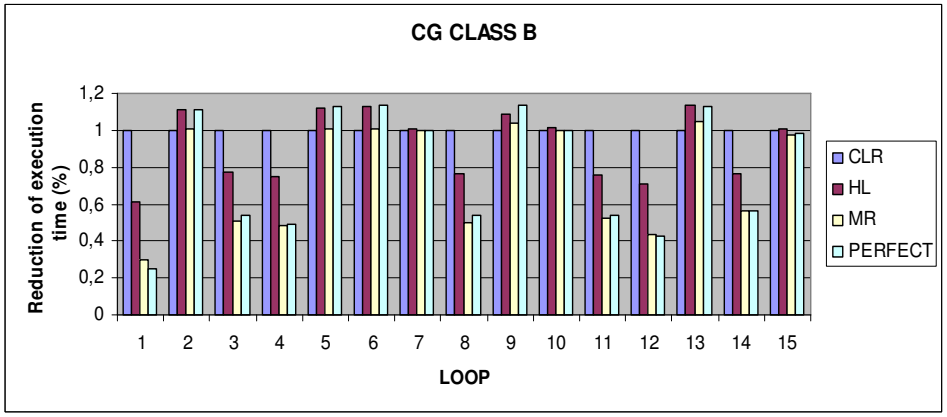
WORK: time spent in actual computation.
WRITE-BACK: time spent in the write-back process.
UPDATE D-B: time spent updating the dirty-bits information.
DMA-IREG: time spent synchronizing with the DMA data transfers in the TC.
DMA-REG: time spent synchronizing with the DMA data transfers in the HLC.
DEC: time spent in the pinning mechanism for cache lines.
TRANSAC: time spent executing control code of the TC.
BARRIER: time spent in the barrier synchronization at end of parallel computation.
MMAP: time spent in executing look-up, placement/replacement actions and DMA programming.

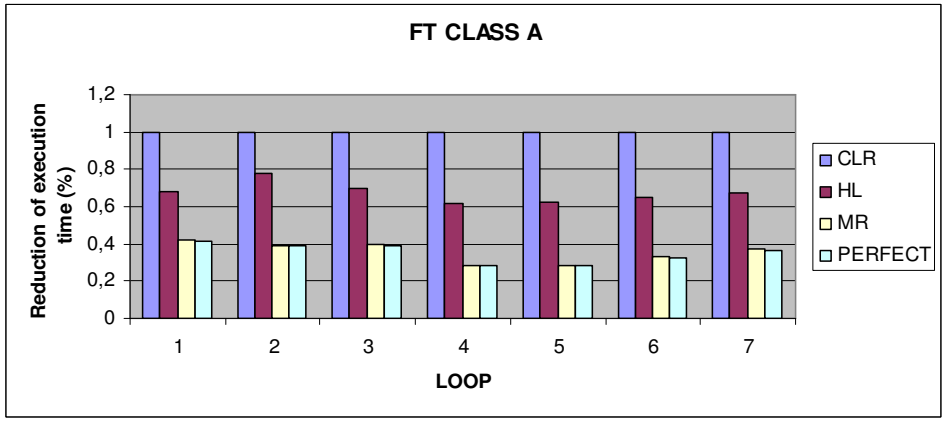
Memory Consistency

- Maintain a relaxed consistency model, following the OpenMP memory model
- Important sources of overhead
  - Dirty Bits: every store operation is monitored
  - Atomicity at write-back process
- Optimizations to smooth the impact of this overhead
  - Several observations for scientific parallel codes:
    - Most of cache lines are modified by one execution flow
    - Buffers usually are totally modified, not requiring atomicity at the moment of write-back
    - Aliasing between data in a buffer and data in the software cache, rarely occur

Evaluation: Memory Consistency







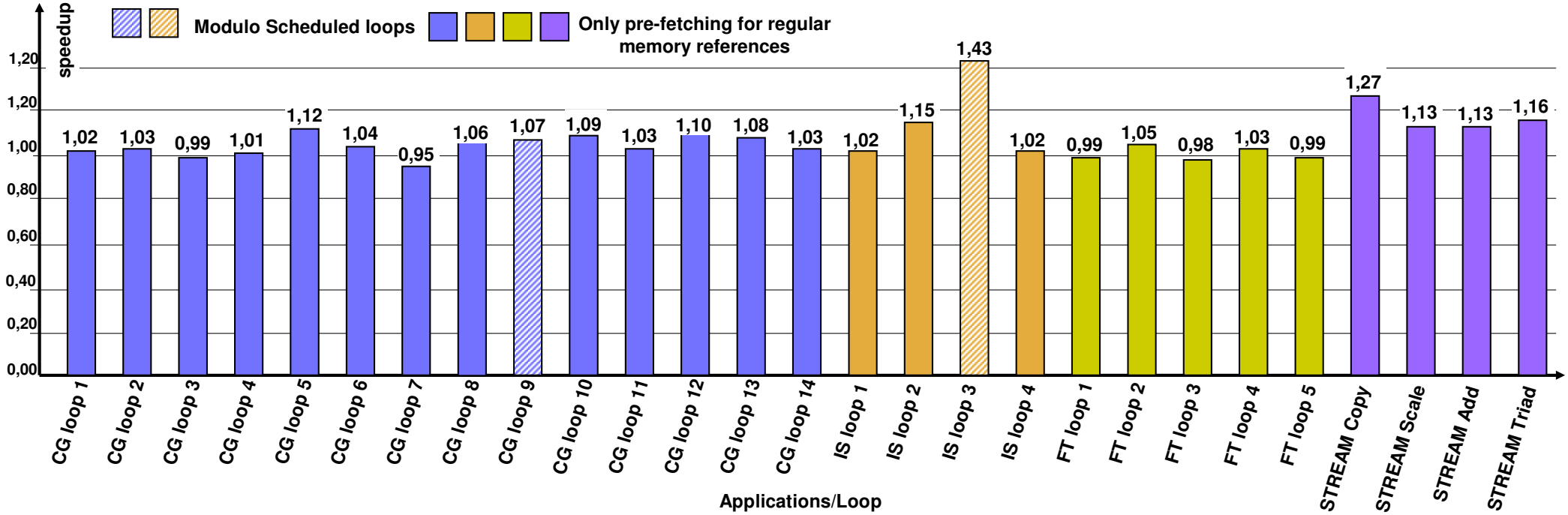
CLR: data eviction based on 128-byte hardware cache line reservation
HL: data eviction is done at buffer level. No alias between data in buffer and data in the software cache.
MR: data eviction is done at buffer level. No alias between data in buffer and data in the software cache, and single writer.
PERFECT: data eviction is freely executed, without atomicity nor dirty-bits

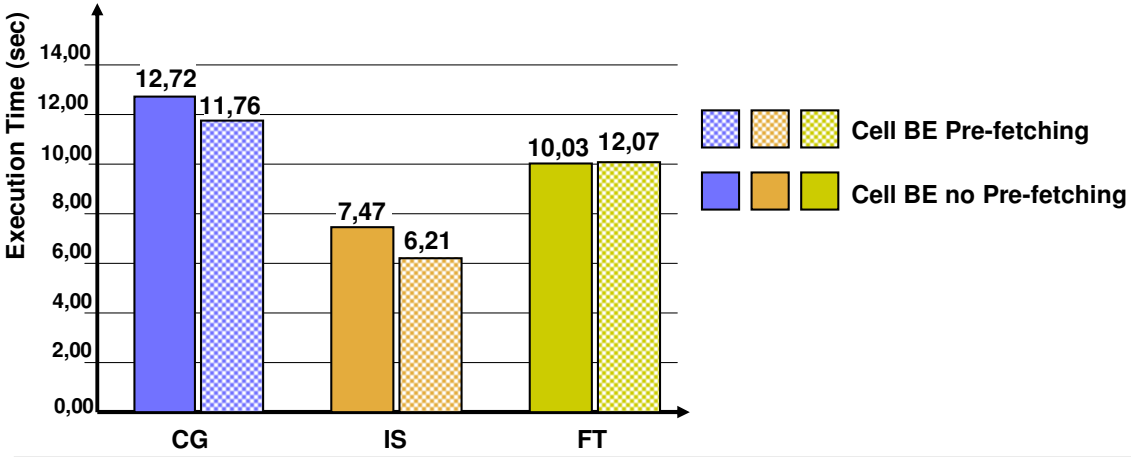
Double buffer techniques

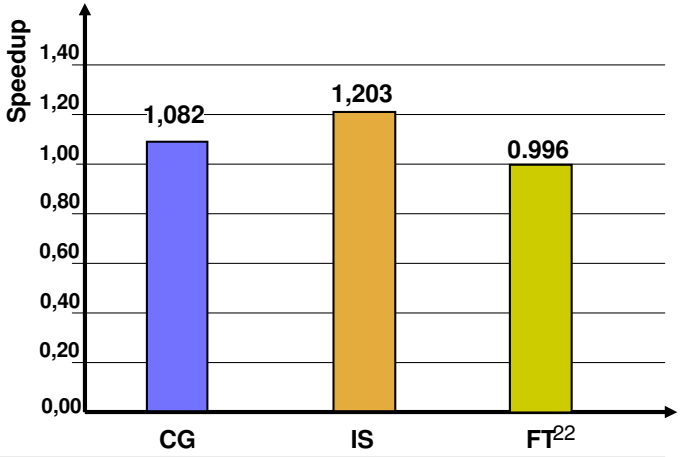
- Double buffer does not come for free
  - Implies executing more control code
  - Requires to adapt the computational bursts to data transfer times
  - Depends on the available bandwidth, which depends it self on the number of executing threads

Evaluation: pre-fecth of data

- Speedups and Execution Times



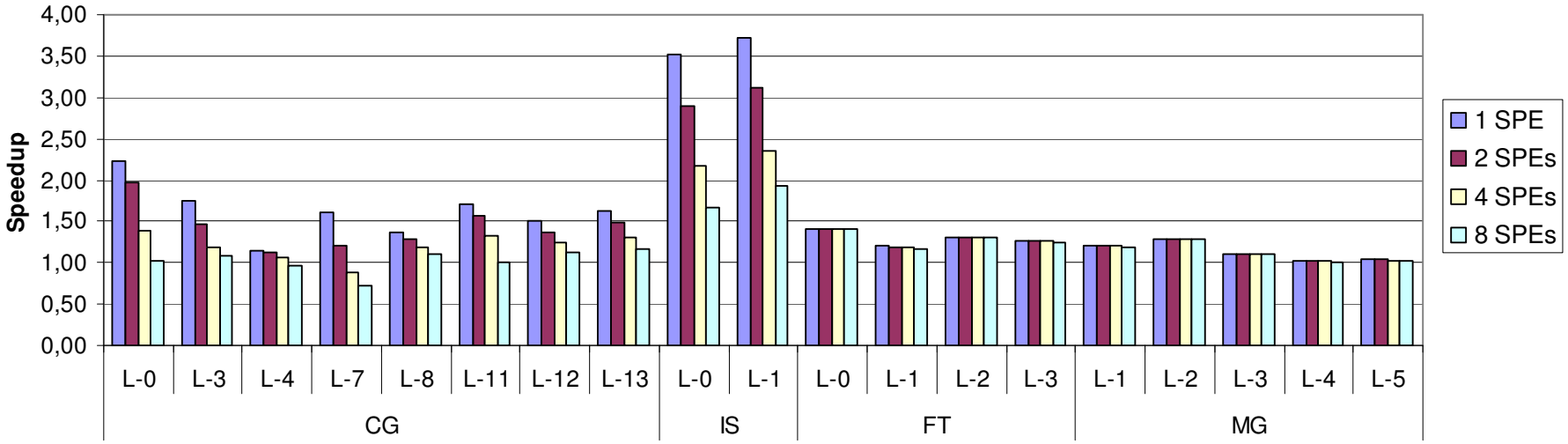




Combining OpenMP with SIMD execution

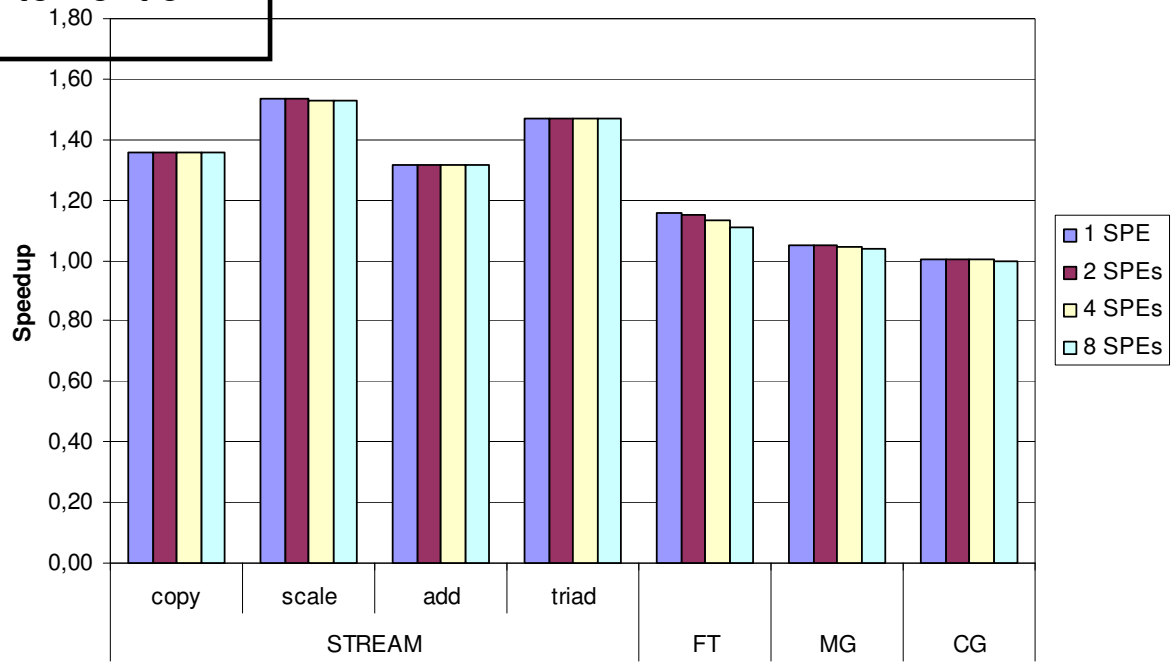
- Actual effect
  - Limited by the execution model
    - Only affects the computational bursts
  - Very dependant on runtime parameters
    - Number of threads
    - Number of iterations per runtime intervention

MG	8 SPEs				
	kernel	iters cnt	4KB buffer	TRANFER	ITERATIONS
1	53327648	696943	4255404	6,11	76,52
2	23873710	350385	2211260	6,31	68,14
3	8388608	49024	49024	1,00	171,11
4	196648	2026	8104	4,00	97,06
5	205232	2005	8020	4,00	102,36
CG	8 SPEs				
	kernel	iters cnt	4KB buffer	TRANFER	ITERATIONS
1	56244	114	684	6,00	493,37
2	1406100	2850	2850	1,00	493,37
3	1406100	2850	5700	2,00	493,37
4	56244	114	228	2,00	493,37
5	56244	114	114	1,00	493,37
6	1406100	2850	11400	4,00	493,37
7	46870	95	190	2,00	493,37
8	46870	95	190	2,00	493,37
FT	8 SPEs				
	kernel	iters cnt	4KB buffer	TRANFER	ITERATIONS
1	33554432	131072	1048576	8,00	256,00
2	33554432	131072	1048576	8,00	256,00
3	29360128	114688	917504	8,00	256,00
IS	8 SPEs				
	kernel	iters cnt	4KB buffer	TRANFER	ITERATIONS
1	2883584	2816	2816	1,00	1024,00
2	23068672	22528	22528	1,00	1024,00



Combining OpenMP with SIMD execution

- Actual effect
  - Limited by the execution model
    - Only affects the computational bursts
  - Very dependant on runtime parameters
    - Number of threads
    - Number of iterations per runtime intervention



Conclusions

- OpenMP transformations
  - Remember, three phases
  - Very conditioned to memory aliasing
    - Try to avoid pointers, introduce inline annotations ...
  - We can reach similar performance as what we would obtain from a cache based multi-core
- Double-buffer effectiveness
  - Depending on the number of threads, access patterns, bandwidth
  - Ranging between 10%-20% of speedup
- SIMD effectiveness
  - Only affects the computational phase
  - Limited by alignment constraints

Questions