On the analysis of virtual platform generated traces

Frédéric Pétrot and Marcos Cunha

Laboratoire TIMA / System Level Synthesis Group
Université Grenoble Alpes

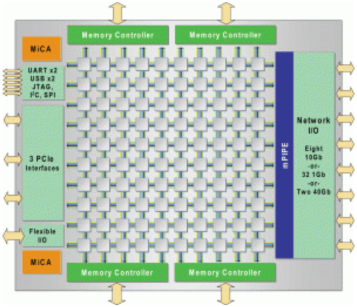
Outline

New architectures, new challenges

- “The processor is the NAND gate of the future”, *dixit Chris Rowen*
- Not quite there yet, but getting close, ...

New architectures, new challenges

- “The processor is the NAND gate of the future”, *dixit Chris Rowen*
- Not quite there yet, but getting close, ...



Tilera (Tile-Mx)

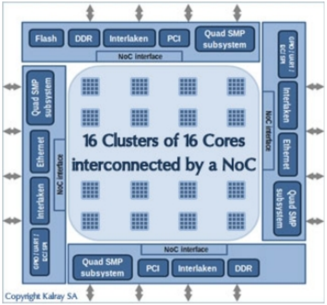
- 100 processors on a chip
- 64-bit ARM
- Chipwide hardware cache coherency
- Power Consumption $\approx$ 100 W

New architectures, new challenges

- "The processor is the NAND gate of the future", *dixit Chris Rowen*
- Not quite there yet, but getting close, ...

Kalray (MPPA - Bostan)

- 256 processors on chip (16 clusters of 16 PE)
- 64-bit 3-issue VLIW
- Caches but no hardware cache coherency at all
- Power Consumption $\approx$ 25 W

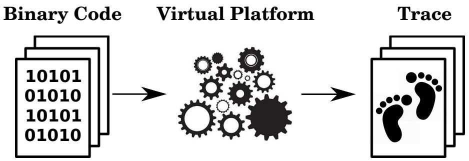


Some SW bugs in *Multi/Many core* architectures

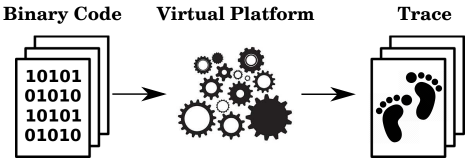
- **Architecture specific bugs**
  - Hardware Software integration mismatches
  - Bad understanding or wrong usage of specific mechanisms by the application/os developper
  - Typical example: access to cached variable that may have been modified

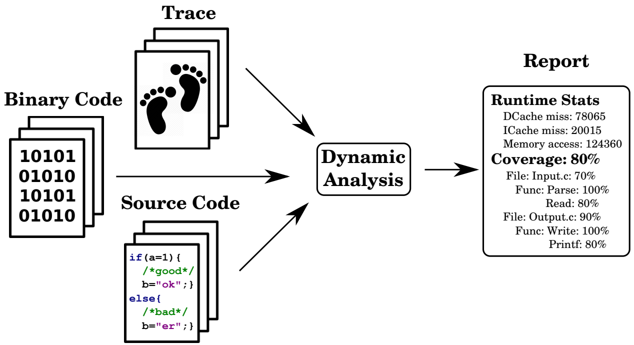
- **Functional bugs**
  - Due (mainly) to parallel execution
  - Potentially sporadic since depending on the execution order (non determinism)

Trace based debug and analysis



Trace based debug and analysis





Traces

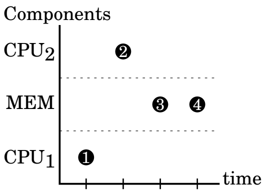
- Set of events giving a view of the system behavior
- Usually generated per component
- Additional relations necessary

Event ID	Component ID	Type of Event	Cycle Number	Data
1	CPU_1	INSTRUCTION	1235678	PC=0x000000A0
2	CPU_2	INSTRUCTION	1235679	PC=0x000000B0
3	MEMORY_1	READ	1235680	ADDR=0xDEADBEEF
4	MEMORY_1	READ	1235781	ADDR=0xDEADBEEF
...	...	...	...	...

Traces

- Set of events giving a view of the system behavior
- Usually generated per component
- Additional relations necessary

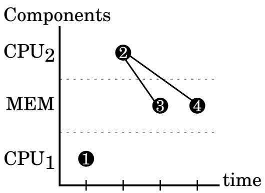
Event ID	Component ID	Type of Event	Cycle Number	Data
1	CPU_1	INSTRUCTION	1235678	PC=0x000000A0
2	CPU_2	INSTRUCTION	1235679	PC=0x000000B0
3	MEMORY_1	READ	1235680	ADDR=0xDEADBEEF
4	MEMORY_1	READ	1235781	ADDR=0xDEADBEEF
...	...	...	...	...



Traces

- Set of events giving a view of the system behavior
- Usually generated per component
- Additional relations necessary

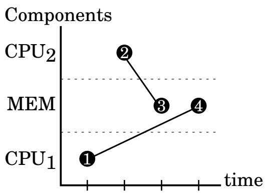
Event ID	Component ID	Type of Event	Cycle Number	Data
1	CPU_1	INSTRUCTION	1235678	PC=0x000000A0
2	CPU_2	INSTRUCTION	1235679	PC=0x000000B0
3	MEMORY_1	READ	1235680	ADDR=0xDEADBEEF
4	MEMORY_1	READ	1235781	ADDR=0xDEADBEEF
...	...	...	...	...



Traces

- Set of events giving a view of the system behavior
- Usually generated per component
- Additional relations necessary

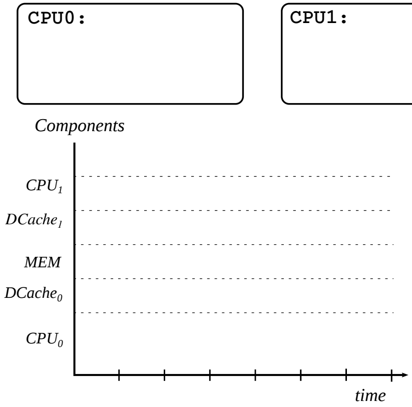
Event ID	Component ID	Type of Event	Cycle Number	Data
1	CPU_1	INSTRUCTION	1235678	PC=0x000000A0
2	CPU_2	INSTRUCTION	1235679	PC=0x000000B0
3	MEMORY_1	READ	1235680	ADDR=0xDEADBEEF
4	MEMORY_1	READ	1235781	ADDR=0xDEADBEEF
...	...	...	...	...



Trace definition

- Goal :
  - Capturing traces representing the parallel system behavior
- Définition : $T = (E, <, \leftarrowtail, \prec)$
  - T : Traces
  - E : Events
  - Relations :
    - $<$: Strict total order of event within *a given* component
    - $\leftarrowtail$: Causality between events belonging to *different* components
    - $\prec$: Sytem total order based on a *shared* component
- Important feature:
  - No timestamping

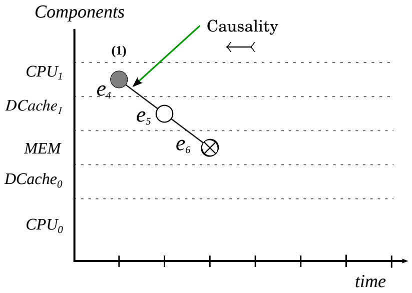
Trace representation



Trace representation

```
CPU0:
```

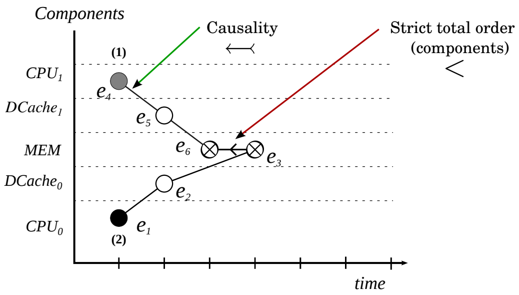
```
CPU1:
    ldr r0,[r1]
```



Trace representation

```
CPU0:
    str r0,[r1]
```

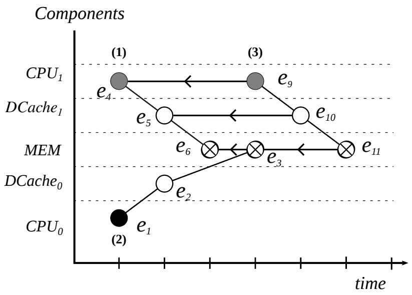
```
CPU1:
    ldr r0,[r1]
```



Trace representation

```
CPU0:
   str r0,[r1]
```

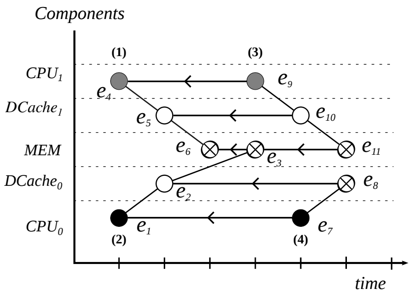
```
CPU1:
   ldr r0,[r1]
   str r0,[r2,#0]
```



Trace representation

```
CPU0:
    str r0,[r1]
    ldr r0,[r2,#0]
```

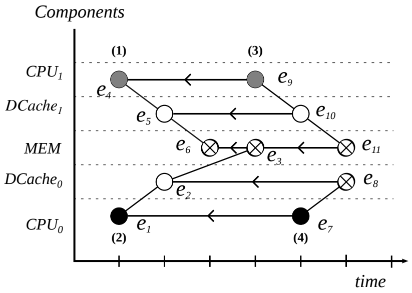
```
CPU1:
    ldr r0,[r1]
    str r0,[r2,#0]
```

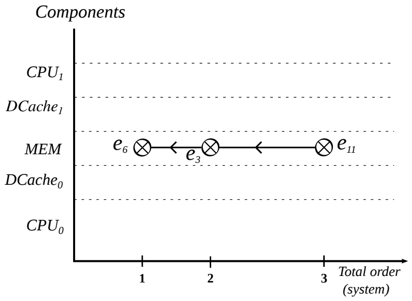


Trace representation

```
CPU0:
    str r0,[r1]
    ldr r0,[r2,#0]
```

```
CPU1:
    ldr r0,[r1]
    str r0,[r2,#0]
```

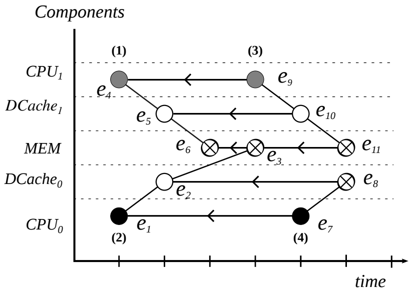


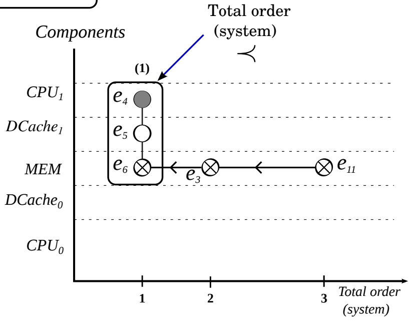


Trace representation

```
CPU0:
    str r0,[r1]
    ldr r0,[r2,#0]
```

```
CPU1:
    ldr r0,[r1]
    str r0,[r2,#0]
```

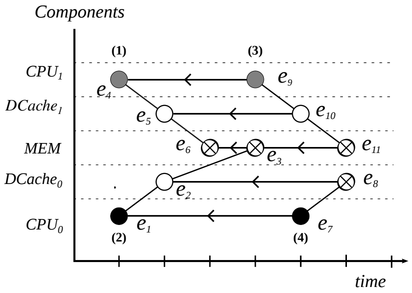


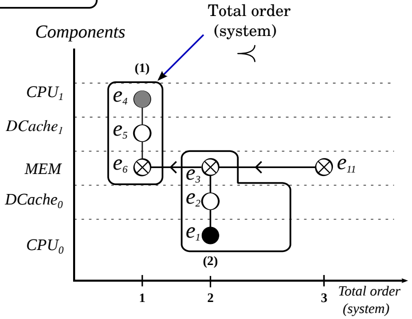


Trace representation

```
CPU0:
    str r0,[r1]
    ldr r0,[r2,#0]
```

```
CPU1:
    ldr r0,[r1]
    str r0,[r2,#0]
```

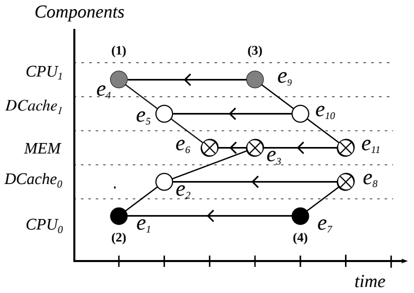


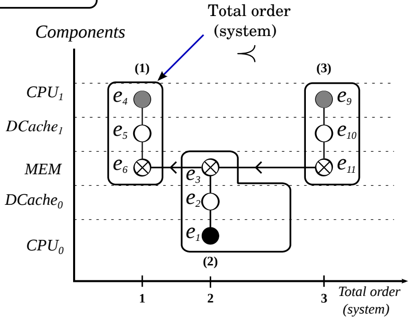


Trace representation

```
CPU0:
    str r0,[r1]
    ldr r0,[r2,#0]
```

```
CPU1:
    ldr r0,[r1]
    str r0,[r2,#0]
```

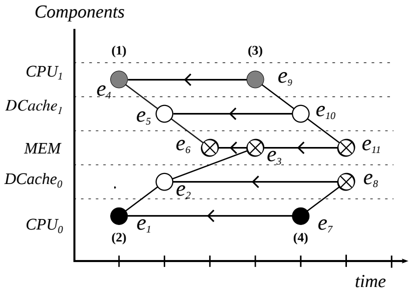

Components
CPU1
DCache1
MEM
DCache0
CPU0
(1)
(3)
(2)
(4)
e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9 e_{10} e_{11}
time

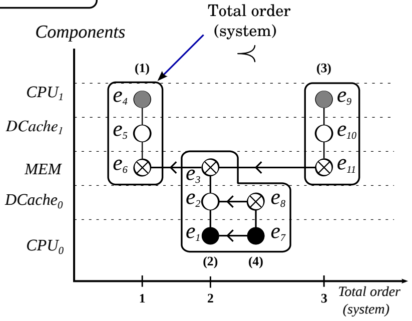


Trace representation

```
CPU0:
    str r0,[r1]
    ldr r0,[r2,#0]
```

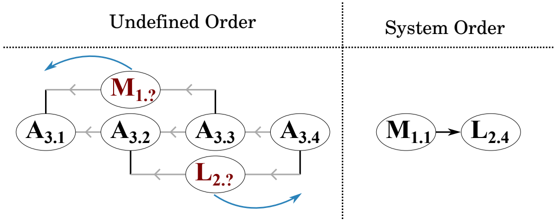
```
CPU1:
    ldr r0,[r1]
    str r0,[r2,#0]
```





Forward and *Rewind* operations

- Goal
  - Assign total order to causality chains without *shared* component
- Operations
  - *Rewind* ($\lll$) : Total order assign to previous shared event
  - *Forward* ($\ggg$) : Total order assign to next shared event



Trace based cache-coherence analysis

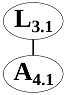
- Goal :
  - Detect cache coherence issues in SW cache coherence protocols
- Example :
  - *Write-through*
    - **Reads :** from cache or memory
    - **Write :** always into memory, also in cache on hit
- Formalize problem as a graph analysis per memory block
- Express set of rules that check for violation

write-through rule

- Checks if a cache accesses “decayed” data
- Exemple :
  - 3 processors (id = 1,2,3)
  - 1 memory (id = 4)
- Verification rule
  1. $S_i \prec L_j$
  2. $\nexists S_k$ such that $S_i \prec S_k \prec L_j$ with $k \in C$
  3. $\nexists L_j^*$ such that $S_i \prec L_j^* \prec L_j$ with $L_j^* \neq L_j$
  4. $\nexists L_j$ such that $L_j \leftarrowtail A_l$

write-through rule

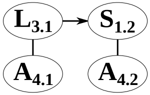
- Checks if a cache accesses “decayed” data
- Exemple :
  - 3 processors (id = 1,2,3)
  - 1 memory (id = 4)



- Verification rule
  1. $S_i \prec L_j$
  2. $\nexists S_k$ such that $S_i \prec S_k \prec L_j$ with $k \in C$
  3. $\nexists L_j^*$ such that $S_i \prec L_j^* \prec L_j$ with $L_j^* \neq L_j$
  4. $\nexists L_j$ such that $L_j \leftarrowtail A_l$

write-through rule

- Checks if a cache accesses “decayed” data
- Exemple :
    - 3 processors (id = 1,2,3)
    - 1 memory (id = 4)



- Verification rule
    1. $S_i \prec L_j$
    2. $\nexists S_k$ such that $S_i \prec S_k \prec L_j$ with $k \in C$
    3. $\nexists L_j^*$ such that $S_i \prec L_j^* \prec L_j$ with $L_j^* \neq L_j$
    4. $\nexists L_j$ such that $L_j \leftarrowtail A_l$

write-through rule

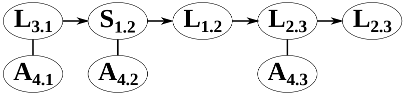
- Checks if a cache accesses “decayed” data
- Exemple :
  - 3 processors (id = 1,2,3)
  - 1 memory (id = 4)

$$\begin{tikzcd}
\mathbf{L_{3.1}} \arrow[r] \arrow[d, dash] & \mathbf{S_{1.2}} \arrow[r] \arrow[d, dash] & \mathbf{L_{1.2}} \\
\mathbf{A_{4.1}} & \mathbf{A_{4.2}} &
\end{tikzcd}$$

- Verification rule
  1. $S_i \prec L_j$
  2. $\nexists S_k$ such that $S_i \prec S_k \prec L_j$ with $k \in C$
  3. $\nexists L_j^*$ such that $S_i \prec L_j^* \prec L_j$ with $L_j^* \neq L_j$
  4. $\nexists L_j$ such that $L_j \leftarrowtail A_l$

write-through rule

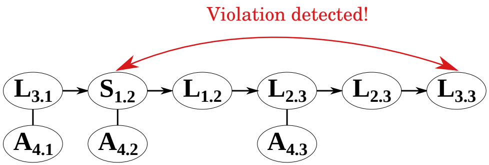
- Checks if a cache accesses “decayed” data
- Exemple :
  - 3 processors (id = 1,2,3)
  - 1 memory (id = 4)



- Verification rule
  1. $S_i \prec L_j$
  2. $\nexists S_k$ such that $S_i \prec S_k \prec L_j$ with $k \in C$
  3. $\nexists L_j^*$ such that $S_i \prec L_j^* \prec L_j$ with $L_j^* \neq L_j$
  4. $\nexists L_j$ such that $L_j \leftarrow\!\prec A_l$

write-through rule

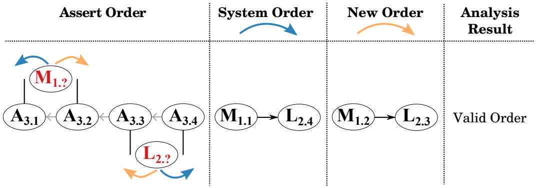
- Checks if a cache accesses "decayed" data
- Exemple :
    - 3 processors (id = 1,2,3)
    - 1 memory (id = 4)



- Verification rule
    1. $S_i \prec L_j$
    2. $\nexists S_k$ such that $S_i \prec S_k \prec L_j$ with $k \in C$
    3. $\nexists L_j^*$ such that $S_i \prec L_j^* \prec L_j$ with $L_j^* \neq L_j$
    4. $\nexists L_j$ such that $L_j \leftarrowtail A_l$

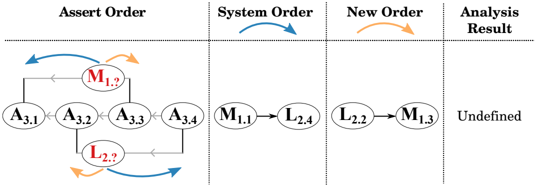
False positives

- Assignments due to *forward* ($\lll$) and *rewind* ($\ggg$)
- Removal
    - If problem, apply opposite operation
    - If order is identical, the problem is confirmed
    - Otherwise, we don't know
- Limitation
    - Possible false positives do exist



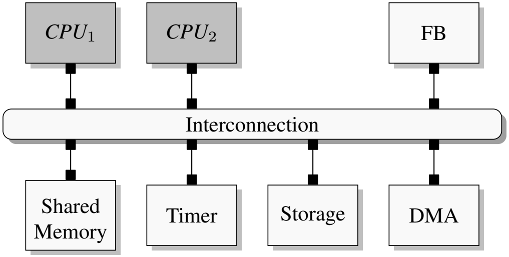
False positives

- Assignments due to *forward* ($\lll$) and *rewind* ($\ggg$)
- Removal
  - If problem, apply opposite operation
  - If order is identical, the problem is confirmed
  - Otherwise, we don't know
- Limitation
  - Possible false positives do exist



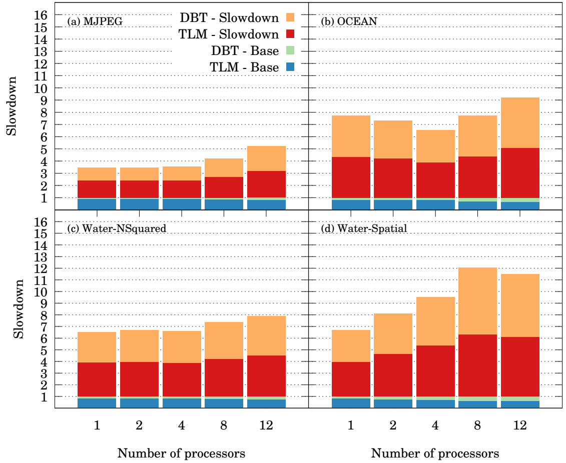
Virtual prototype

- Hardware
  - Rabbits simulator with enhanced trace capture
  - Processors : up to 16 Cortex-A9
- Software
  - Parallel MJPEG/Splash-2 (all `pthread` based)



Trace generation

- Simulation slowdown due to trace generation

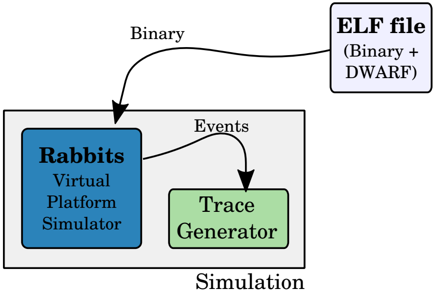


Trace analysis

- Detect and correct cache coherence violations

Approach

- Iterative process
  1. Identify accesses to “decayed” data
  2. Correct by acting on the local cache

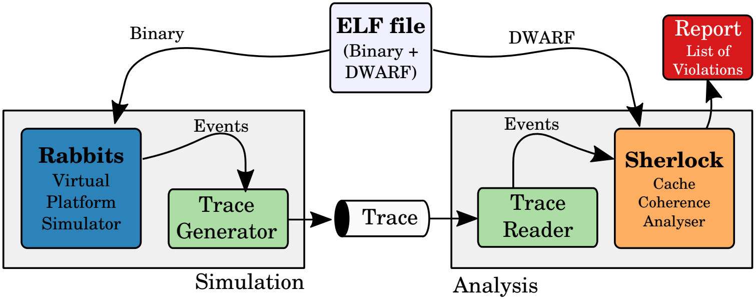


Trace analysis

- Detect and correct cache coherence violations

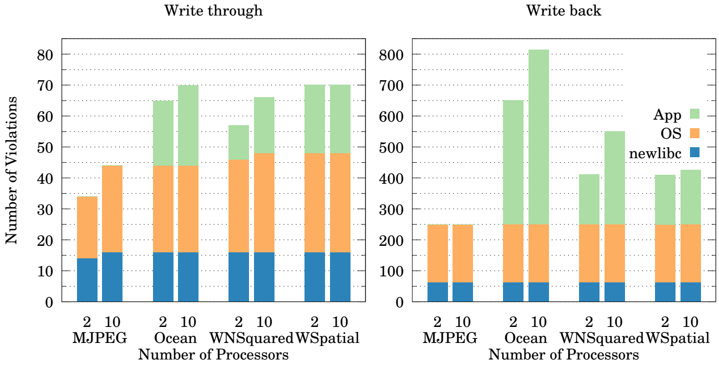
Approach

- Iterative process
  1. Identify accesses to “decayed” data
  2. Correct by acting on the local cache



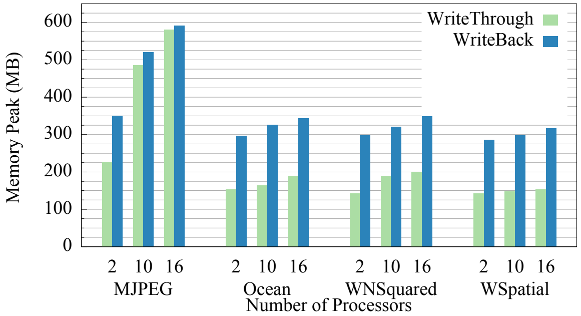
Cache coherence analysis: Violations

- Number of violations detected and corrected per program (Initial hypothesis: hardware coherent shared memory)



Cache coherence analysis: Complexity

- Analysis time is $O(k * |E|)$ with $k \ll |E|$
- Analysis is done online: Peak memory usage limited



Conclusion

VP produced execution traces:

- Require lots of resources
  - Take time to be generated
  - Need huge disk space to be stored
  - Need time and memory to be analysed
- But are very useful
  - Allow to obtain traces with relations between events
  - Simplifies analysis greatly:
    NP hard consistency model violation problem becomes linear with read/write mapping
  - Permit online analysis for some problem