Efficient OpenMP Implementation and Translation For Multiprocessor System-On-Chip without Using OS

Woo-Chul Jeun and Soonhoi Ha
Seoul National University, Korea

2007.01.24

wcjeun@iris.snu.ac.kr

Contents

- Background
- OpenMP implementation for MPSoC configurations
- Our OpenMP implementation and translation for a target platform
- Conclusions & Future directions

Background

MPSoC (Multiprocessor System-on-Chip)

- **Attractive platform for computation-intensive applications such as multimedia encoder/decoder**
- **Can have various architectures according to target applications**
  - Memory: Shared, Distributed, and Hybrid
  - OS: SMP(symmetric multiprocessor) kernel Linux, small operating systems, and no operating system
- **No standard parallel programming model for MPSoC**
- **Not sufficient studies on various MPSoC architectures**

Parallel Programming Models

	Message-passing	Shared-address-space
Memory	Distributed memory (private memory)	Shared memory (same address space)
Comm.	Explicit message-passing	Memory access
De facto Standard	MPI (1994) (message-passing interface)	OpenMP (1998)
Programming easiness	Difficult	Easy
Performance	Optimize data distribution, data transfer, and synchronization manually	Complicate issues depend on OpenMP implementation

- Manual optimization (MPI) vs. Easy programming (OpenMP)

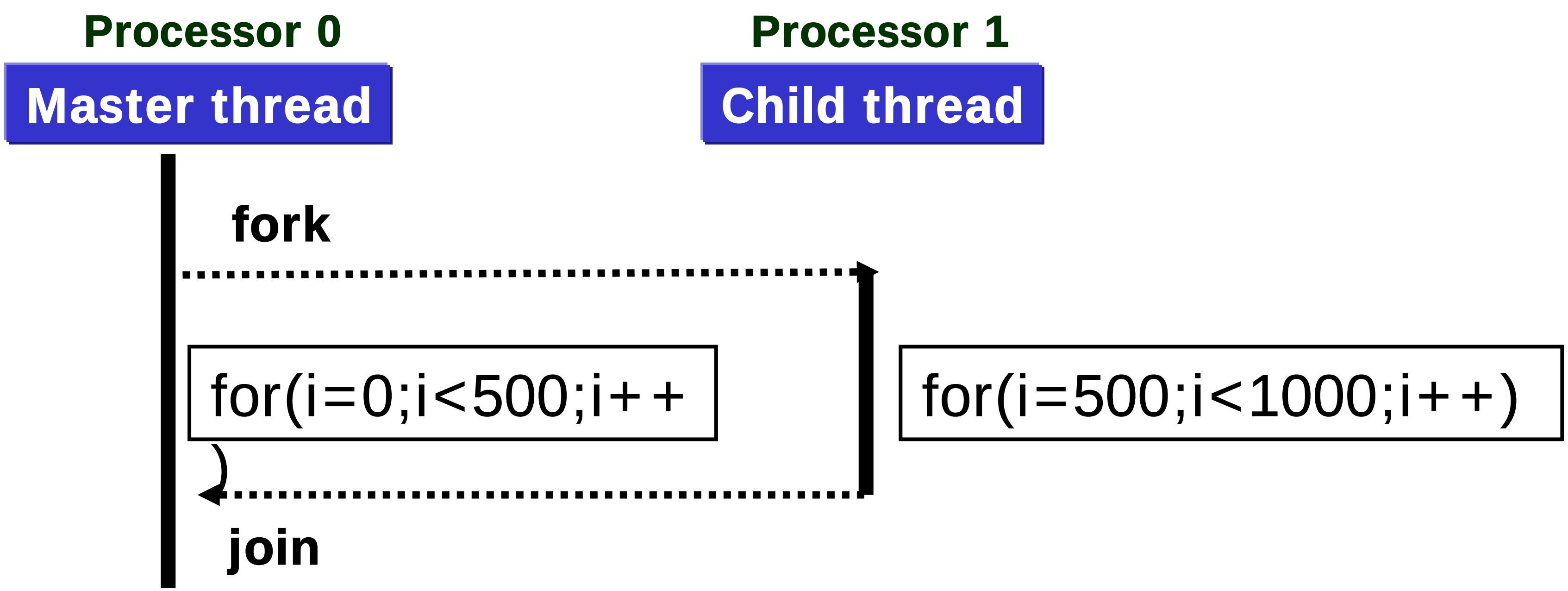
OpenMP Overview

- **Specification standard to represent programmer's intension with compiler directives (C/C++ and Fortran)**
- **OpenMP is an attractive parallel programming model for MPSoC because of easy programming.**
- **Programmers can write an OpenMP program by inserting OpenMP directives into a serial program.**

```
#pragma omp parallel for
for( i = 0 ; i < 1000 ; i++) {
        data[i] = 0;
}
```

OpenMP execution model : fork/join

```
#pragma omp parallel for
for( i = 0 ; i < 1000 ; i++) {
        data[i] = 0;
}
```



- **All threads divide and execute its own computation workload.**

OpenMP programming environment

- OpenMP does not define how to implement OpenMP directives on a parallel processing platform.

- OpenMP runtime system
  : OpenMP directive implementation with libraries on a target platform

- OpenMP translator (for C language)
  : converts an OpenMP program into the codes (C codes) using the OpenMP runtime system

Hybrid execution model

- Separating parallel programming model from execution model
- Application programmers use OpenMP.
- OpenMP runtime system can use hybrid execution model of message passing model and shared address space model for the performance improvement.
- Easy programming and High performance

Motivation

- OpenMP runtime system depends on a target platform.
- OpenMP translator is customized to the OpenMP runtime system.
- To get high performance on various platforms, it is necessary to research on efficient OpenMP runtime system and OpenMP translator for each target platform.

Terminologies

- **Barrier synchronization**
  - Every processor (or thread) waits until all processors (or threads) arrive at a synchronization point.
- **Reduction operation**
  - integrates operations of all processors (or threads) into one operation.

Processor 0	Processor 1	Processor 2
Sum: 0	Sum: 0	Sum: 0
sum+=a;	sum+=b;	sum+=c;

Reduction operation

sum+= (a+b+c);

Sum: a+b+c	Sum: a+b+c	Sum: a+b+c

OpenMP implementation on MPSoC configurations

Possible OpenMP implementation on MPSoC configurations

	Distributed memory	Shared memory
OS with thread library	Thread programming + SDSM : fault handler	Thread programming + Shared memory (Yoshihiko et al.)
Without OS	Processor programming + SDSM : message passing	Processor programming + Shared memory (Feng Liu et al., Ours)

- SDSM (software distributed shared memory) vs. Shared memory
- Thread programming vs. Processor programming

Shared memory + OS with thread library

	Distributed memory	Shared memory
OS with thread library	Thread programming+ SDSM : fault handler	Thread programming+ Shared memory
Without OS	Processor programming+ SDSM : message passing	Processor programming+ Shared memory

- **No need for memory consistency protocol**
- **Similar to thread programming in a SMP machine (Ex. dual processor PC)**
- **Yoshihiko Hotta et al., [EWOMP'2004]**
  - SMP(symmetric multiprocessor) kernel Linux and POSIX thread library
  - Similar to OpenMP implementation and translation in a SMP machine (dual processor PC)
  - They focused on power optimization

Shared memory + No OS

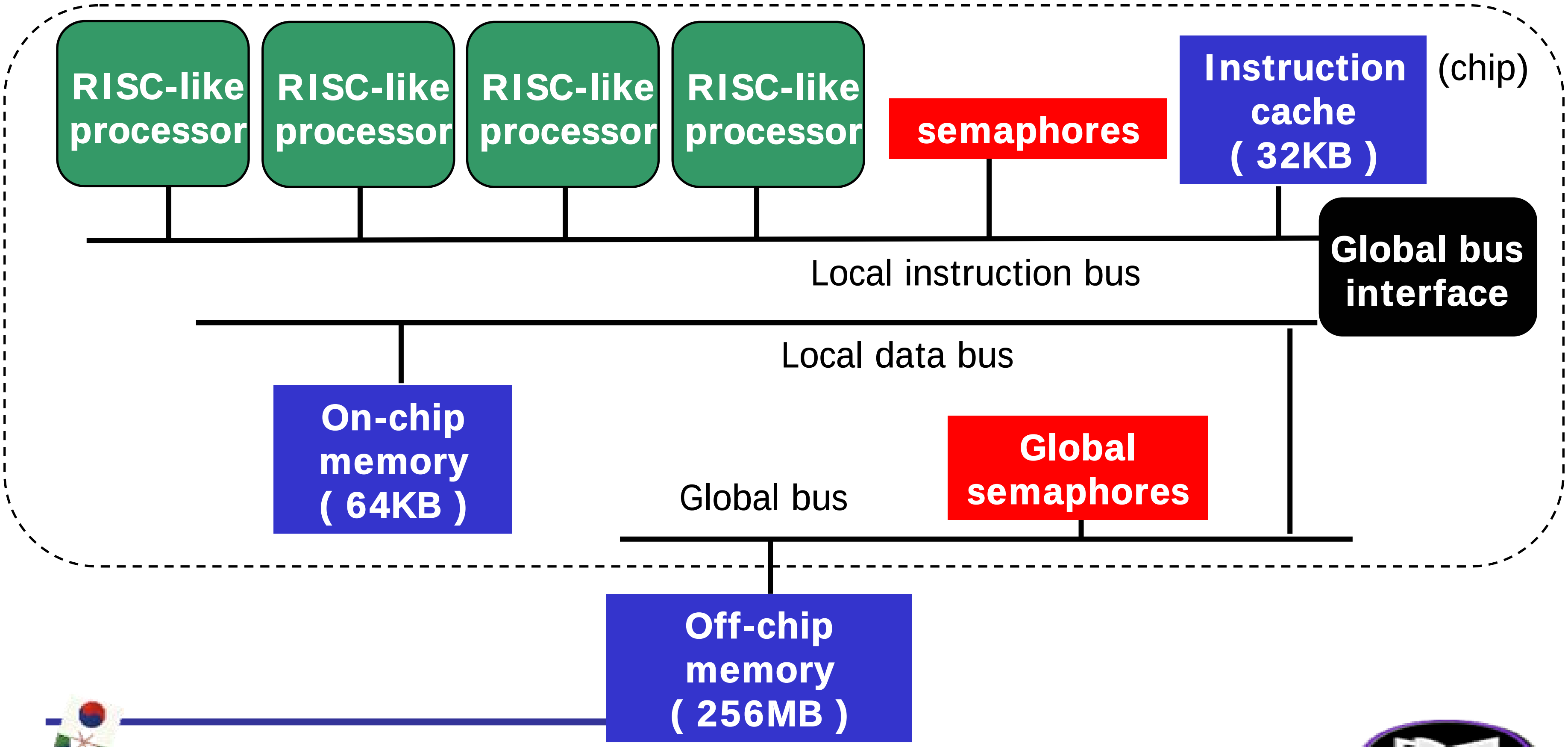
	Distributed memory	Shared memory
OS with thread library	Thread programming+ SDSM : fault handler	Thread programming+ Shared memory
Without OS	Processor programming+ SDSM : message passing	Processor programming+ Shared memory

- **No need for memory consistency protocol**
- **Make processors run a program in parallel (load and initiate processors)**
- **Feng Liu et al., [WOMPAT'03, ICPP'2003]**
  - No operating system
  - Their own OpenMP directive extension for DSP
  - OpenMP directive extension for special hardware on CT3400
  - Harmful barrier synchronization implementation

Our OpenMP implementation and translation on a target multiprocessor system-on-chip platform

CT3400 Architecture (Cradle Technologies, Inc.)

- 230MHz processor, hardware semaphores (32 local, 64 global)
- Shared memory; No operating system and no thread library



Initialization

- OpenMP translator extracts original main function to a function. (app_main())
- Make new main function call original main
- Initialization procedure loads program codes on other processors and initiates them before application starts (initializer())

Original main

```
int main(...)
{
}
```

OpenMP translation

New main on master node

```
int main(...) {
    initializer(...);
    app_main(...);
}
```

```
int app_main(...) {
}
```

Parallelization

- OpenMP translator extracts a parallel region to a function
- All processors execute the function. (cf. thread)
- Master processor executes serial region and other processors wait until master processor arrives at a parallel region.

Parallel region

```
#pragma omp parallel for
for(i=0;i<1000;i++){
    data[i]=0;
}
```

OpenMP translation

New main

```
parallelize(..., parallel_region_0)
```

```
void parallel_region_0(...) {
for(i=start;i<end;i++){
    data[i]=0;
}
```

Translation of global shared variables

- '*cragcc*' C compiler on CT3400 can process global variables efficiently
- OpenMP translator can translate global shared variables with two memory allocation methods.
- Static allocation
  - int data[100]; , global data area (0%~31% better)
  - OpenMP translator can inform the OpenMP runtime system that the variables are in global data area.
- Dynamic allocation
  - int *data; data = allocate_local(...); , heap area
  - OpenMP runtime system cannot know whether the variables are global variables.

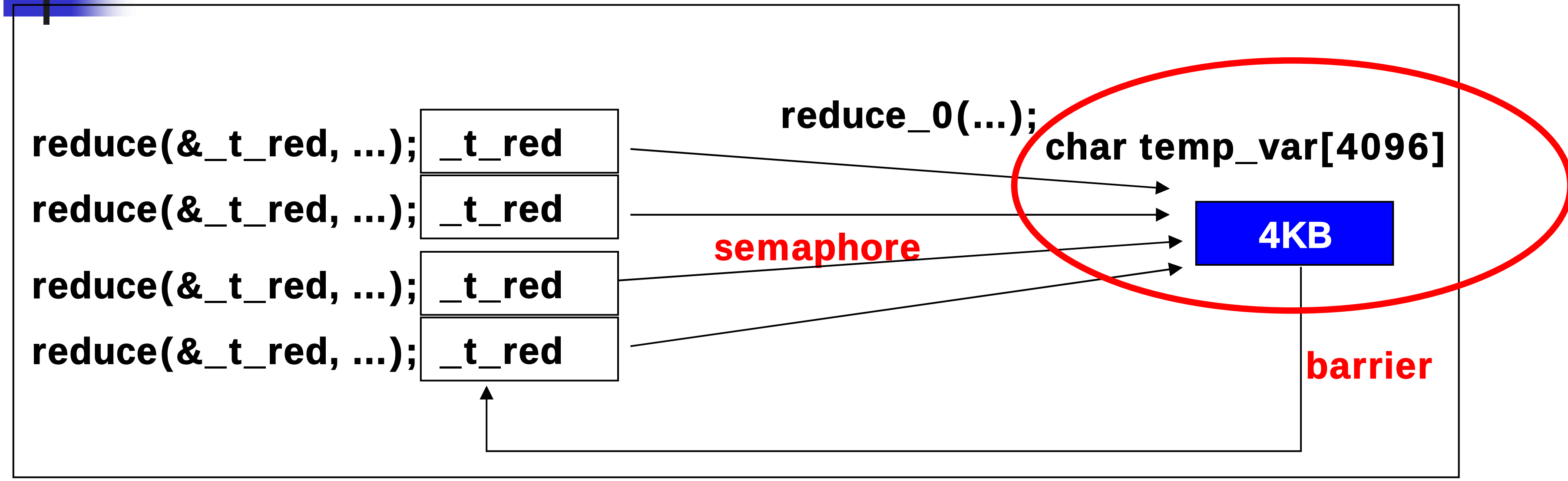
24*24 Matrix multiplication (cycles)

- On the cycle-accurate simulator "Inspector" provided by Cradle technologies, Inc.

Processors	1	2	4
Serial	3,664,513	N/A	N/A
Parallel (hand-written)	3,653,761	1,827,537	914,127
OpenMP, Dynamic	5,221,225	2,622,901	1,320,474
OpenMP, Static	3,674,336	1,845,050	933,549

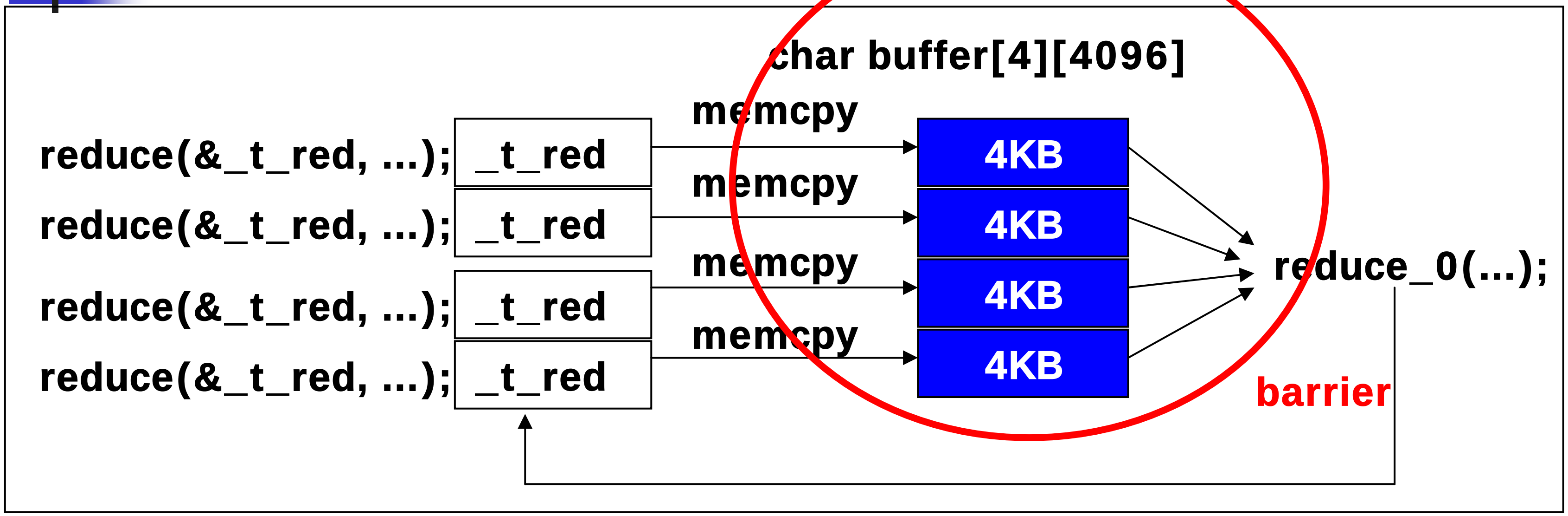
- **Static memory allocation is 31% better than dynamic memory allocation on CT3400.**

Reduction (using temporary variable)



- Uses a temporary variable. (temp_var)
- Similar to thread programming
- Each processor updates the temporary variable with semaphore.
- All operations are serialized.

Reduction (using temporary buffer array)



- Uses a temporary buffer array (buffer).
- Each processor updates its own element of the array without semaphore.
- All operations can be executed in parallel.

EPCC OpenMP micro-benchmark (cycles)

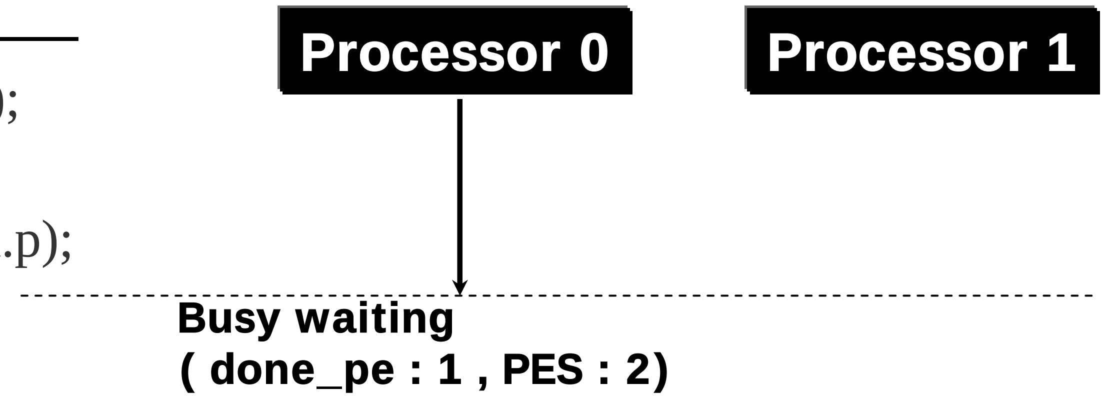
- On the cycle-accurate simulator "Inspector" provided by Cradle technologies, Inc.

Processors	1	2	4
Reduction, temporary variable	1,713	8,790	14,028
Reduction, temporary buffer array	1,713	7,805	12,631

- **Temporary buffer array method is 10% better than temporary variable method on CT3400.**

Previous harmful barrier synchronization implementation (example error case)

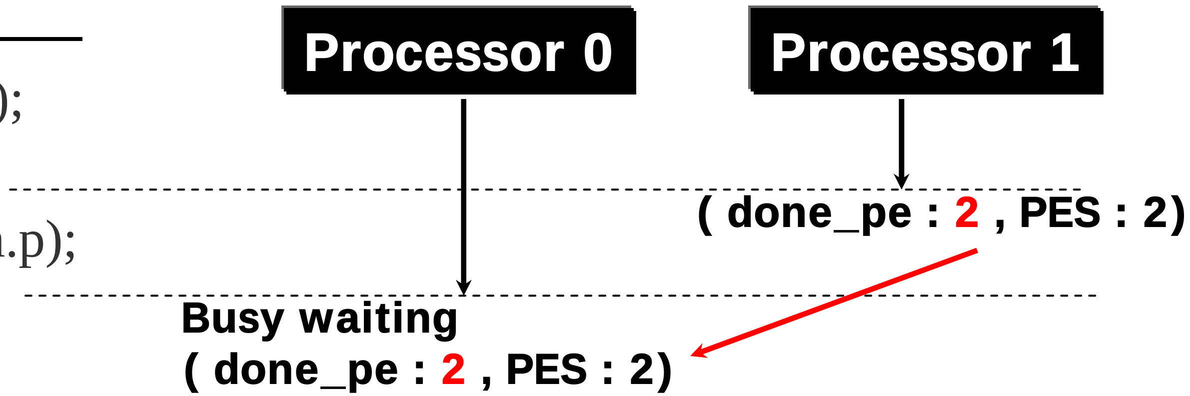
```
semaphore_lock(Sem.p);
done_pe++;
semaphore_unlock(Sem.p);
while( done_pe < PES )
    _pe_delay(1);
if( my_peid = = 0 )
    done_pe = 0;
```



- **PES (Number of processors) : 2**
- **done_pe (counter variable for synchronization)**
- **my_peid (processor ID)**
- **Processor 0 increases 'done_pe' to 1 with semaphore and does busy waiting.**

Previous harmful barrier synchronization implementation (example error case)

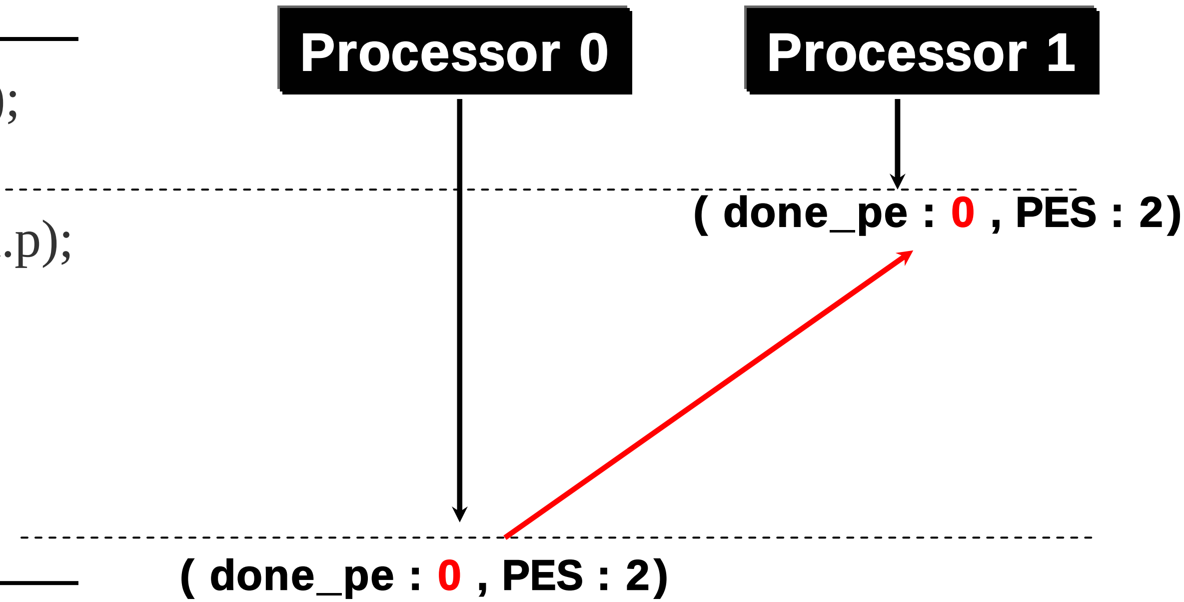
```
semaphore_lock(Sem.p);
done_pe++;
semaphore_unlock(Sem.p);
while( done_pe < PES )
     _pe_delay(1);
if( my_peid = = 0 )
     done_pe = 0;
```



- **Processor 1 increases the counter.**
- **Processor 0 can exit the busy waiting loop.**

Previous harmful barrier synchronization implementation (example error case)

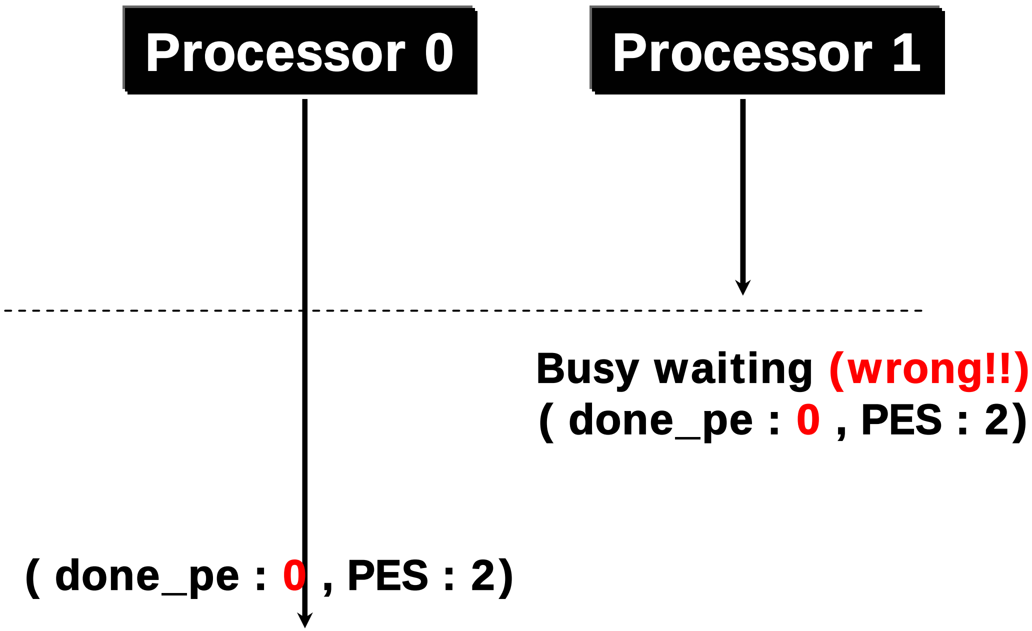
```
semaphore_lock(Sem.p);
done_pe++;
semaphore_unlock(Sem.p);
while( done_pe < PES )
     _pe_delay(1);
if( my_peid = = 0 )
     done_pe = 0;
```



- **Processor 0 initializes the counter variable before processor 1 checks the value of the counter variable.**

Previous harmful barrier synchronization implementation (example error case)

```
semaphore_lock(Sem.p);
done_pe++;
semaphore_unlock(Sem.p);
while( done_pe < PES )
    _pe_delay(1);
if( my_peid = = 0 )
    done_pe = 0;
```



- **Processor 1 cannot exit the loop and it fails for synchronization (wrong)**
- **Wrong assumption of this implementation : last processor is always processor 0 and it initializes the counter of current barrier**

Our barrier implementation

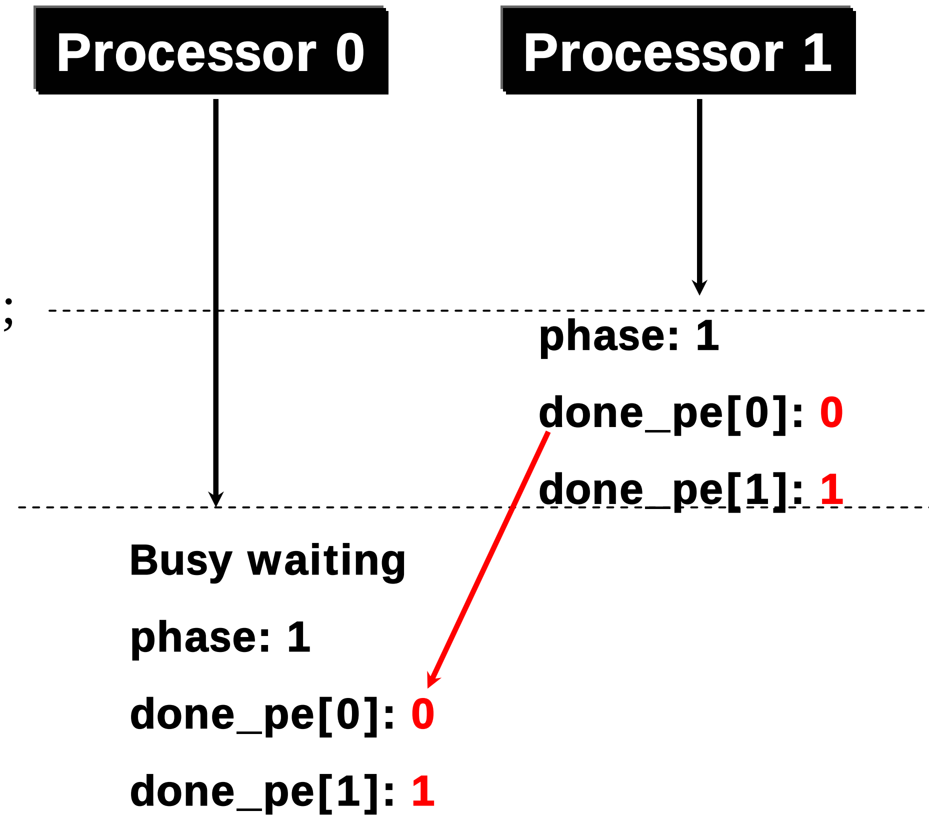
```
semaphore_lock(Sem.p);
done_pe++;
semaphore_unlock(Sem.p);
while( done_pe < PES )
    _pe_delay(1);
if( my_peid = = 0 )
    done_pe = 0;
```

```
semaphore_lock(Sem.p);
 phase = (phase + 1) % 2;
if( done_pe[phase] + 1 = = PES )
    done_pe[( phase + 1 ) % 2 ] = 0;
done_pe[phase]++;
semaphore_unlock(Sem.p);
while( done_pe[phase] < PES )
    _pe_delay(1);
```

- **Introduce a phase variable and toggle phase of barrier to discriminate consequent barriers**
- **Initialize the counter of next barrier**

Our barrier implementation

```
semaphore_lock(Sem.p);
 phase = (phase + 1) % 2;
 if( done_pe[phase] + 1 = = PES )
     done_pe[( phase + 1 ) % 2 ] = 0;
done_pe[phase]++;
semaphore_unlock(Sem.p);
while( done_pe[phase] < PES )
     _pe_delay(1);
```



- Initialize the counter variable of next barrier and keep the counter variable of current barrier at the same time.

Conclusions & Future directions

- When we translate global shared variables, static memory allocation is **31% better** than dynamic memory allocation.
- For the reduction implementation, temporary buffer array method is **10% better** than temporary variable method.
- We fixed previous harmful barrier synchronization implementation.
- Future directions
  - MPSoC with Distributed Memory
  - MPSoC with Heterogeneous processors (Ex. DSP)