VOL. E103-C NO. 3
MARCH 2020

A PUBLICATION OF THE ELECTRONICS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER *Special Section on Low-Power and High-Speed Chips*

Compiler Software Coherent Control for Embedded High Performance Multicore

Boma A. ADHI[†a)], **Tomoya KASHIMATA**[†], **Ken TAKAHASHI**[†], ***Nonmembers***, **Keiji KIMURA**[†], ***and*** **Hironori KASAHARA**[†], ***Members***

SUMMARY The advancement of multicore technology has made hundreds or even thousands of cores processor on a single chip possible. However, on a larger scale multicore, a hardware-based cache coherency mechanism becomes overwhelmingly complicated, hot, and expensive. Therefore, we propose a software coherence scheme managed by a parallelizing compiler for shared-memory multicore systems without a hardware cache coherence mechanism. Our proposed method is simple and efficient. It is built into OSCAR automatic parallelizing compiler. The OSCAR compiler parallelizes the coarse grain task, analyzes stale data and line sharing in the program, then solves those problems by simple program restructuring and data synchronization. Using our proposed method, we compiled 10 benchmark programs from SPEC2000, SPEC2006, NAS Parallel Benchmark (NPB), and MediaBench II. The compiled binaries then are run on Renesas RP2, an 8 cores SH-4A processor, and a custom 8-core Altera Nios II system on Altera Arria 10 FPGA. The cache coherence hardware on the RP2 processor is only available for up to 4 cores. The RP2's cache coherence hardware can also be turned off for non-coherence cache mode. The Nios II multicore system does not have any hardware cache coherence mechanism; therefore, running a parallel program is difficult without any compiler support. The proposed method performed as good as or better than the hardware cache coherence scheme while still provided the correct result as the hardware coherence mechanism. This method allows a massive array of shared memory CPU cores in an HPC setting or a simple non-coherent multicore embedded CPU to be easily programmed. For example, on the RP2 processor, the proposed software-controlled non-coherent-cache (NCC) method gave us 2.6 times speedup for SPEC 2000 "equake" with 4 cores against sequential execution while got only 2.5 times speedup for 4 cores MESI hardware coherent control. Also, the software coherence control gave us 4.4 times speedup for 8 cores with no hardware coherence mechanism available.

key words: *multicore, software coherence control, parallelizing compiler, shared memory, cache, soft core*

1. Introduction

Shared memory Symetric Multi Processing (SMP) machine or coherent cache architecture is one of the most widely used computer architecture, from the simplest IoT devices, smartphones, real-time embedded systems, gaming PCs, servers in the cloud to the record-breaking HPCs. Caches, or in general hierarchical memory concepts, are commonly utilized in an SMP system to hide the latency of larger and slower memory. But, a private cache in a share memory multicore has an inherent cache coherency problem.

Typically a hardware-based mechanism, either a snoopy-based or directory-based, is utilized to manage the cache coherence; it keeps every change made into a shared line in one processor's private cache always reflected to all other processors' cache line. Snoopy-based mechanism tends to be simpler to implement in hardware, but its working principle relies on a shared bus to connect all the CPU cores [1], which is not scalable for a large number of CPU cores. Directory-based approach was later proposed to address this scalability issues [2] but at the cost of additional storage for the directory, extra latency, and increased interconnect traffic as the mechanism keeps the directory updated. For today's multicore processor, the directory-based cache coherence mechanism scales well [3], e.g. Intel Xeon Phi [4], Tilera Tile64 [5]. But, despite its common usage among current generation multicore processors, this kind of hardware eventually becomes very complex, hot, and expensive for the upcoming hundreds to thousands of core massively parallel multicore systems. Also, for a simple embedded multicore CPU, its complexity makes designing and verifying this mechanism difficult [6], further increasing the development cost. FPGA based multicore also benefits as a hardware-based coherence consumes extra FPGA resources.

To eliminate such complexity, hardware-based cache coherency can be replaced with a simpler and more scalable software-based solution. The research on non-coherent cache shared memory system has been started as early as the late '80s. At that time, the effort was about eliminating or simplifying the notorious complexity of hardware-based cache coherence by using a software approach. One of the notable early works was [7], which proposed a fast selective invalidation scheme and version control scheme for compiler directed cache coherence. And later on, they showed that their proposal was capable of maintaining comparable performance to directory-based scheme [8]. Data Flow algorithm was also used to detect stale data reference, thus improving temporal and spatial locality [9]. Several other approaches were to simplify the hardware cache coherence mechanism by a timestamp-based method [10], by reducing complicated memory sharing pattern [11], or by using a self-invalidation scheme [12]. Another approach was to enable compiler support for automatically inserting cache management instruction into an OpenMP parallel program to solve stale data problem [13].

On the other side of the story, back in 2007, we developed an 8 cores multicore CPU [14] which instead of consisting a single 8 core SMP in a single coherency domain, due to a limited budget and development time it has two sets

Manuscript received July 1, 2019.
Manuscript revised October 29, 2019.

[†]The authors are with Department of Computer Science and Engineering, Waseda University, Tokyo, 169–8555 Japan.
a) E-mail: boma@kasahara.cs.waseda.ac.jp

DOI: 10.1587/transele.2019LHP0008

of 4-cores SMP cluster with each own separate coherency domain. To utilize all 8 cores as a single SMP system, the software has to manage the cache coherency. We were unable to find any off-the-shelf solution which could work for our purpose.

The research effort on the OSCAR Compiler has been started since 35 years ago. OSCAR Compiler is a source-to-source multi-grain parallelizing compiler. It is capable of automatically generating parallelized code for an SMP cache-coherent machine with loop optimization, cache optimization, power management, and other optimization [15]. In this research, we extend the capability of the OSCAR Compiler to automatically generate a parallel code for non-cache-coherent SMP system. We integrated several elemental compilation techniques to solve two main problems in a non-cache coherent multicore architecture; false sharing by data alignment, array expansion, array padding, and non-cacheable buffer, and stale data by self-invalidation and synchronization, into our OSCAR Compiler. The new compiler module utilizes the parallelized-sections data and def-use data from the compiler framework to solve both fundamental non-cache-coherent architecture problems.

We first introduced our concept and the preliminary evaluation of this research in IEEE COMPSAC 2017 [16]. Our approach is the first to integerate an automatic parallelizing compiler which generates code for non-cache-coherent platform by solving both stale data and false sharing problem. In the current paper, we further explained the algorithm. In addition to the RP2 Multicore CPU, we added more hardware platforms, the Nios II based soft CPU SoC on FPGA. We ran some benchmarks and obtained better or comparable performances compared to the hardware-based approach. This research enables automatic parallelization with a simple, easy to program and efficient Non-Cache-Coherent (NCC) many cores processor. Right now we start our effort from a small embedded CPU and soft CPU for FPGA, but the same principle could be applied to a larger system more CPU cores.

2. Problems in a Non-Cache-Coherent Architecture

In a multicore SMP system, usually, a dedicated hardware mechanism takes care of each CPU core's private cache coherency. In the absence of such mechanisms, the software should maintain coherency by itself. Writing a correctly executing program for a non-cache-coherent machine by hand is a daunting task for a programmer. To achive a compiler controlled cache coherence, the compiler has to solve the following two fundamental problems.

2.1 Stale Data Problem

A hardware-based cache coherence ensures every change made to the data in one of the CPU core's cache line is propagated to other cores and each copy of this data in other cores are then invalidated. The process of notifying the other processor in a snoopy-based cache coherence may impact the performance of the processor. With a directory-based mechanism, simultaneous access to the directory may become a performance bottleneck. Meanwhile, without any hardware cache coherence, these bottlenecks do not exist, but the compiler should manually manage access to stale data.

In the absence of hardware cache coherence, stale data reference should be avoided. An update made by one of the processor core is invisible to other cores until the changes are written back to memory or all copies of the same data in other cores' cache are invalidated.

A trivial example can be seen in Fig. 1. Assume 3 integer-typed global variables are declared in the main memory, a, b, and c. All of them are stored in a single memory line and shared by both core 0 and core 1. At first, core 1 assigns b to a. Then core 1 fetches the data into its cache and starts updating the data, b = a. Please note that because a, b, and c share the same line, all of them are copied into core 1's cache. And because of that, both a and b are zero,

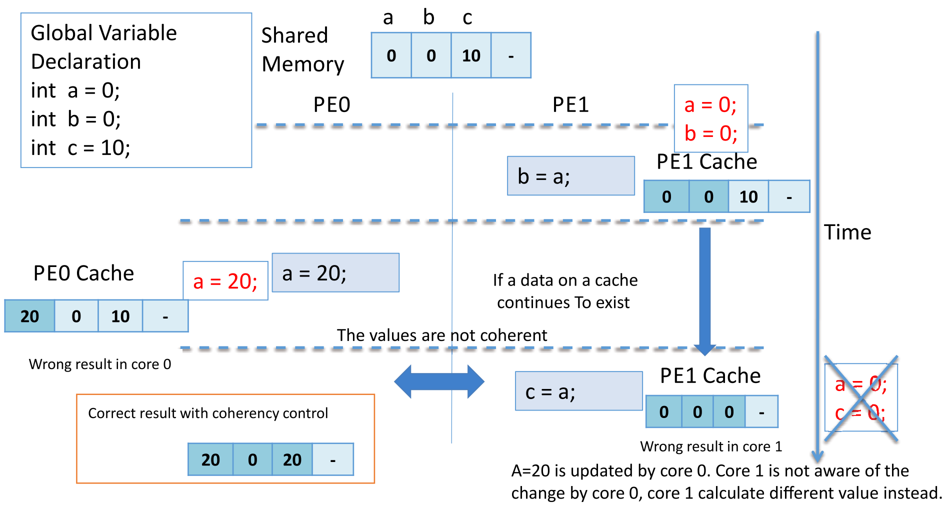

Global Variable Declaration
int a = 0;
int b = 0;
int c = 10;
Shared Memory
a b c
0 0 10 -
PE0
PE1
a = 0;
b = 0;
b = a;
PE1 Cache
0 0 10 -
Time
PE0 Cache
a = 20;
a = 20;
20 0 10 -
If a data on a cache continues To exist
The values are not coherent
Wrong result in core 0
c = a;
PE1 Cache
0 0 0 -
a = 0;
c = 0;
Correct result with coherency control
20 0 20 -
Wrong result in core 1
A=20 is updated by core 0. Core 1 is not aware of the change by core 0, core 1 calculate different value instead.

Fig. 1 The stale data problem.

so no variable is updated. Then, core 0 would like to change the value of a to 20. Core 0 fetches all the variable into its cache and start to change the value of a. In the absence of hardware coherence control, the change is not reflected in the main memory nor core 1, hence cache coherence is not maintained. Next, core 1 would like to change c equal to a. Here we can see that the change made by core 0 is not reflected in the core 1's cache. Therefore, c would be updated with the stale or out of date data from the core 1 cache instead of the correct data from core 0. In the end, none of the core has the correct calculation result. The values of a, b, and c are different in both core 0 and core 1. None of them are correct.

2.1.1 Stale Data Handling

Based on the coarse grain scheduling result, to manage the stale data problem, the compiler generates explicit cache manipulation instructions to the processor, i.e., writeback, self-invalidate, and purge. Writeback command tells the processor to write the modified cache line to the main memory. The self-invalidate is a command for invalidating the line of the cache memory. The purge command executes the self-invalidate after the writing back (writeback) of the data stored in the line of the cache memory.

The cache manipulation code is inserted into the code segment in which communications occur between tasks running in different cores. Figure 2 is an example of the compiler-generated code to prevent stale data reference. Core 0 defines a new value of a shared variable A. The compiler automatically inserts a writeback instruction and a synchronization flag on core 0's code. The compiler also inserts a self-invalidate instruction on core 1 right after the synchronization flag. For this purpose, OSCAR Compiler uses static scheduling. The compiler then optimize the schedules for the task in a way to minimize the delay caused by the synchronization. Besides, if multiple cores retain the same data at the same time, the compiler schedules all cores in a way to prevent the data from being simultaneously updated.

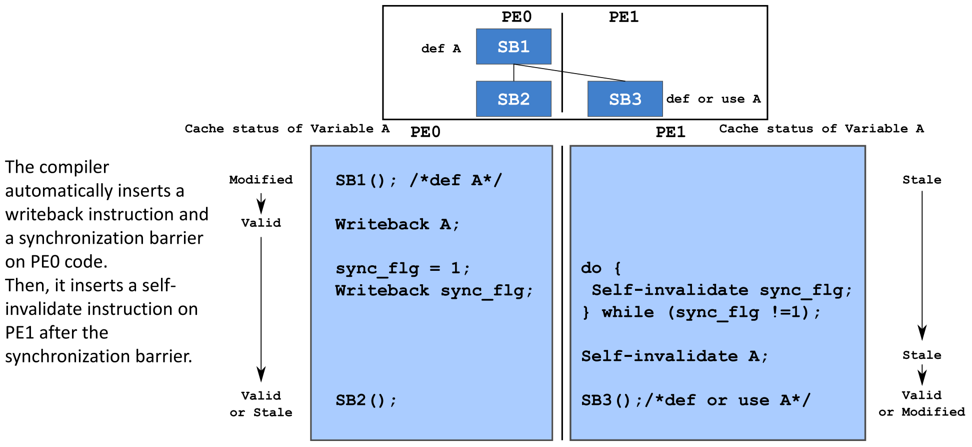
These behaviors are originally part of the OSCAR Compiler default behaviors [17], but those behaviors also improve the performance of the NCC architecture. These cache manipulation instructions are inserted only for Read-after-Write and Write-after-Write data dependence. Meanwhile, for Write-after-Read, only synchronization instruction is inserted. The OSCAR Compiler also manages the memory consistency of the program similar to OpenMP. So, in this case, it guarantees that the memory is consistent at the end of each macro task [18]. By using this approach, stale data can be avoided. Moreover, the overhead caused by the transmission of invalidating packets associated with hardware-based mechanism can be eliminated.

2.1.2 Selective Cache Operation in Loop Parallelization

DOALL loop and reduction loop typically has no inherent stale data problem because of the fact that there is no data dependency between iterations [19]. On the other hand, DOACROSS loop has dependency between each iteration [19], hence the possibility of stale data problem. Running such DOACROSS loop in a multicore NCC architecture requires insertion of cache invalidation instruction in each inner loop iteration. While this is possible, it effectively disables the function of the cache as all data should be fetched from the shared memory. To mitigate this problem, the compiler analyzes the access pattern of the loop. Stale data problem only exists if the array element which is accessed by parallelized iteration shares the same cache line [20]. Hence, the compiler inserts the cache invalidation instruction only when required.

2.2 False Sharing Problem

False sharing is a condition in which two or more independent data share a single cache line. Whenever one of those data is updated, inconsistency may occur. This is due to the granularity of the cache writeback mechanism which works on line level instead of a single byte or word.


PE0
PE1
def A
SB1
SB2
SB3
def or use A
Cache status of Variable A
PE0
PE1
Cache status of Variable A
The compiler automatically inserts a writeback instruction and a synchronization barrier on PE0 code. Then, it inserts a self-invalidate instruction on PE1 after the synchronization barrier.
Modified
Valid
Valid or Stale
Stale
Stale
Valid or Modified

```
SB1(); /*def A*/

Writeback A;

sync_flg = 1;
Writeback sync_flg;

SB2();
```

```
do {
 Self-invalidate sync_flg;
} while (sync_flg !=1);

Self-invalidate A;

SB3();/*def or use A*/
```

Fig. 2 Cache control code inserted by the compiler to prevent reference to stale data.

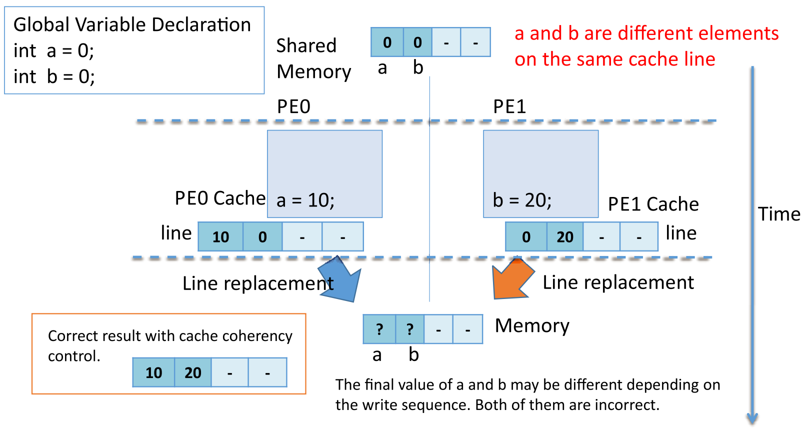

Global Variable Declaration
int a = 0;
int b = 0;
Shared Memory
0 0 - -
a b
a and b are different elements on the same cache line
PE0
PE1
PE0 Cache
a = 10;
line 10 0 - -
b = 20;
PE1 Cache
0 20 - - line
Time
Line replacement
Line replacement
Correct result with cache coherency control.
10 20 - -
? ? - -
Memory
a b
The final value of a and b may be different depending on the write sequence. Both of them are incorrect.

Fig. 3 The false sharing problem.

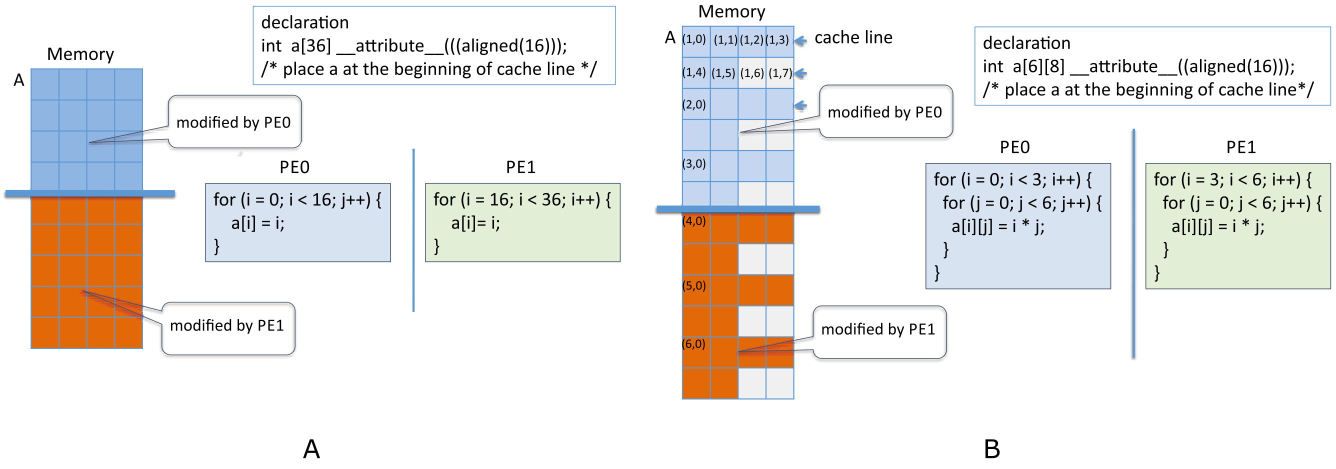

Memory
A
modified by PE0
modified by PE1
declaration
int a[36] __attribute__(((aligned(16)));
/* place a at the beginning of cache line */
PE0
for (i = 0; i < 16; j++) {
a[i] = i;
}
PE1
for (i = 16; i < 36; i++) {
a[i]= i;
}
Memory
A
(1,0) (1,1) (1,2) (1,3)
(1,4) (1,5) (1,6) (1,7)
(2,0)
(3,0)
(4,0)
(5,0)
(6,0)
cache line
modified by PE0
modified by PE1
declaration
int a[6][8] __attribute__((aligned(16)));
/* place a at the beginning of cache line*/
PE0
for (i = 0; i < 3; i++) {
for (j = 0; j < 6; j++) {
a[i][j] = i * j;
}
}
PE1
for (i = 3; i < 6; i++) {
for (j = 0; j < 6; j++) {
a[i][j] = i * j;
}
}
A
B

Fig. 4 (A) Cache aligned loop decomposition is applied to a one-dimension matrix to avoid false sharing. (B) Array padding is applied to a two-dimention matrix to avoid false sharing.

A simple example is provided in Fig. 3. Assume there are 2 global variables declared in the shared main memory, a and b. Both a and b are not related, but both share a single cache line due to their small size. Assume a parallelized code which requires 2 different cores to update a and b independently at the same time. Therefore, both processor cores have the same line in their respective cache. While core 0 only updates a and core 1 only updates b, core 0 is not aware of the change made by core 1 on b and vice versa. Therefore, without hardware cache coherence scheme, the final value of a and b is always incorrect. The final result depends on the line replacement sequence of both cores. A simple data alignment to the beginning of each cache line usually solves the problem, but it is not always an ideal solution. There are several other kinds of false sharing especially in arrays that should be handled differently.

2.2.1 Variable Alignment and Array Expansion

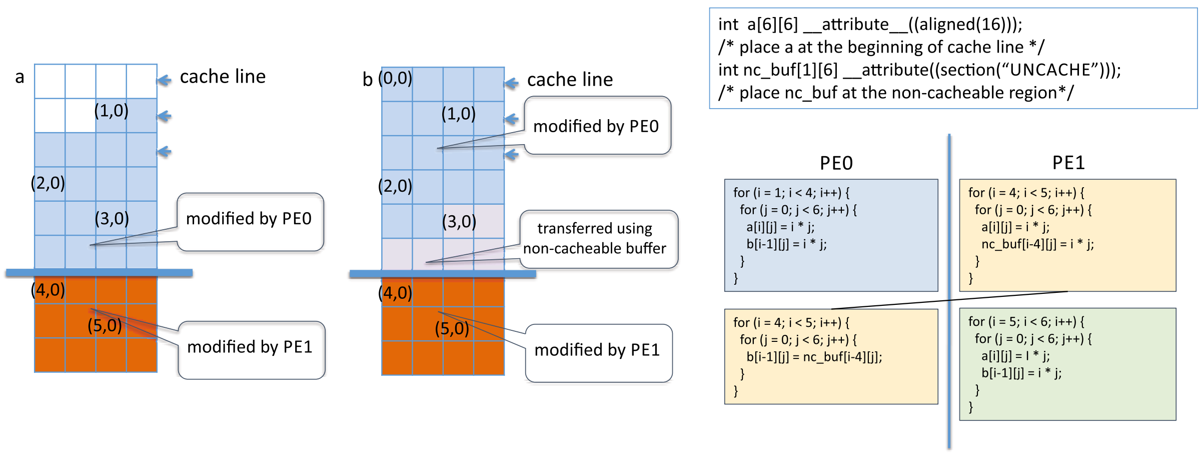
To prevent unrelated variables from sharing a single cache line, the compiler aligns each variable to the beginning of a cache line. Not only for scalar variables, but this approach is also applicable for small sized one-dimension array. The array can be expanded so that each element is stored in a single cache line. While not very efficient due to potentially wasting cache space, this approach effectively prevents false sharing. Data alignment works best for one-dimension array whose size is smaller than the number of cache line in all available processor cores. It also works well for indirect access array where the compiler has no information regarding the access pattern of the array.

2.2.2 Cache Aligned Loop Decomposition

Loop decomposition distributes a loop inside a certain task to several partial tasks which are then run on different processor cores. Instead of equally distribute the number of iteration to each partial task, the compiler decomposes a loop with respect to cache line size as seen in Fig. 4 (A).

2.2.3 Array Padding

A two-dimension array is not always possible to be splited cleanly along the cache line. This is due to the lowest dimen-



```
int  a[6][6] __attribute__((aligned(16)));
/* place a at the beginning of cache line */
int nc_buf[1][6] __attribute((section("UNCACHE")));
/* place nc_buf at the non-cacheable region*/
```

PE0

```
for (i = 1; i < 4; i++) {
  for (j = 0; j < 6; j++) {
    a[i][j] = i * j;
    b[i-1][j] = i * j;
  }
}
```

```
for (i = 4; i < 5; i++) {
  for (j = 0; j < 6; j++) {
    b[i-1][j] = nc_buf[i-4][j];
  }
}
```

PE1

```
for (i = 4; i < 5; i++) {
  for (j = 0; j < 6; j++) {
    a[i][j] = i * j;
    nc_buf[i-4][j] = i * j;
  }
}
```

```
for (i = 5; i < 6; i++) {
  for (j = 0; j < 6; j++) {
    a[i][j] = I * j;
    b[i-1][j] = i * j;
  }
}
```

Fig. 5 Non-cacheable buffer is used to avoid false sharing.

sion of the array is not an integer multiply of the cache line size. In this case, the compiler inserts a padding to the end of the array to match the cache line size. This approach is depicted in Fig. 4 (B). It should be noted that this approach, while insignificant, may also waste cache space.

2.2.4 Data Transfer Using Non-Cacheable Buffer

In the case that cache aligned loop may cause a significant imbalance or array padding is not preferred due to cache space restriction or due to the structure of the program none of the former approach cannot be applied, a non-cacheable buffer may be used. The compiler utilizes a small area in the main memory that should not be copied to the cache along the border between area modified by different processor core. Figure 5 depicts the usage of non-cacheable buffer.

3. Software Cache Coherent Control by Parallelizing Compiler

In this paper we extend the capability of the OSCAR Compiler. A new compiler module is written. It takes benefits of the existing extensive OSCAR Compiler framework to handle non-cache-coherent architecture. The OSCAR Compiler is a multi-grain parallelizing compiler. The compiler generates a parallelized C or Fortran program using OSCAR API [15] that allows us to automatically generate parallel multicore code relying on a sequential compiler for any shared-memory multicore targets available in the market. The OSCAR Compiler currently is capable of automatically generating parallelized code for an SMP cache-coherent machine with loop optimization, cache optimization, power management, and other optimization.

The OSCAR compiler starts the compilation process by dividing the source program into three types of coarse-grain tasks, or Macro Tasks (MTs): Basic Blocks (BBs), Repetition Blocks (RBs), and Subroutine Blocks (SBs). RBs and SBs are hierarchically decomposed into smaller MTs if coarse-grain task parallelism still exists within the task. Then, as all MTs for the input program are generated, they are analyzed to produce a Macro Flow Graph (MFG). An MFG is a control flow graph among the MTs having the data dependence edges. Based on this information, the compiler detects and data usage patterns, inserting self invalidation for any suitable def-use pattern. Then A Macro Task Graph (MTG) is generated by analyzing the Earliest Executable Condition (EEC) of every MT and tracing the control dependencies and data dependencies among MTs on the MFG [15], [18].

The general idea of the proposed method is that a parallelizing compiler analyzes and decomposes the program based on its control flow and data dependence, then it partitions the data according to the size of the cache and checks for false sharing possibility, then it automatically inserts cache manipulation instruction to the program. To achieve the goals, we made some changes to the compilation process of OSCAR Compiler. Figure 6 depicts the proposed compilation process. The grayed boxes are new steps introduced to handle cache coherency.

As a parallelizing compiler, after figuring out the EEC and creating coarse grain schedule, the OSCAR Compiler automatically creates parallel sections. It also collects def-use pattern by tracing the control and data dependency in addition to array distribution & array access analysis and pointer analysis [18]. Based on both informations, the proposed compiler module analyzes write sharings between parallel sections. All variables or arrays shared between two or more parallel sections at the same time are marked. For each of those shared variables, a simple decision tree is taken. Depending on the kind of the shared array, a different approach is taken to minimize the impact of false sharing. A simple scalar variable is aligned to the cache line. Also, a small 1-dimensional array is aligned and expanded. The decomposition of the loop considers the size of the array to prevent false sharing. For two-dimensional array, depending on its innermost dimension, it is padded to fit into the cache line. If all effort fails, Non-cacheable buffer is used. Then, for each read-after-write conflicts between simultaneous parallel sections the compiler inserts a self-invalidate and synchronization. The process is described in Fig. 7.

```
input: simultaneousParallelSections, defUseList
output: data structure changes and API Call

foreach simultaneousParallelSections p
    foreach defUseList l
        find sharedVariables
        find RAWConflicts

foreach sharedVariables s
    if s is scalar then align to cache
    else if s is array then
        if s is single dimensional array then
            if number of elements < number of cache line
              then align array elements to cache line
            else
              change loop division to integer multiply of cache line width
        else
          if innermost dimension != integer multiply of cache line width
            then pad innermost dimension to cache line size
          change loop division to integer multiply of cache line width / innermost dimension (+ pad)
    else use non-cacheable buffer

foreach simultaneousParallelSections p
    foreach RAWConflicts r
        insert self invalidation and synchronization
```

Fig. 7 Pseudocode for detecting and mitigating stale data and false sharing.

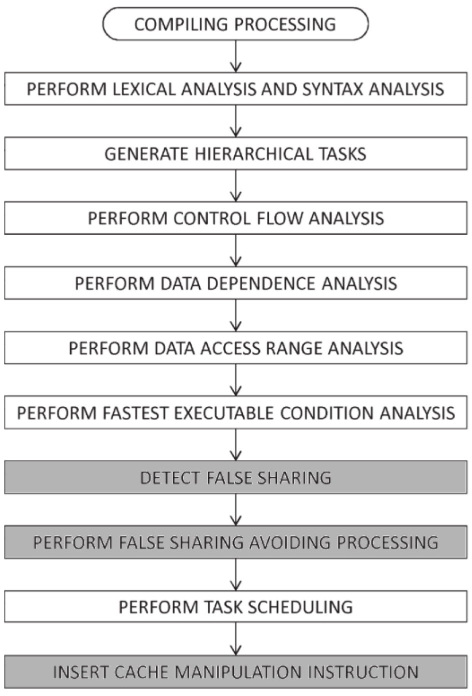

COMPILING PROCESSING
PERFORM LEXICAL ANALYSIS AND SYNTAX ANALYSIS
GENERATE HIERARCHICAL TASKS
PERFORM CONTROL FLOW ANALYSIS
PERFORM DATA DEPENDENCE ANALYSIS
PERFORM DATA ACCESS RANGE ANALYSIS
PERFORM FASTEST EXECUTABLE CONDITION ANALYSIS
DETECT FALSE SHARING
PERFORM FALSE SHARING AVOIDING PROCESSING
PERFORM TASK SCHEDULING
INSERT CACHE MANIPULATION INSTRUCTION

Fig. 6 Proposed compilation sequence.

4. Target Architecture

The target architecture of the proposed method is a non-coherent cache architecture multicore processor which is depicted in Fig. 8. All processors share a single shared memory. The system should have a private cache for each core with no hardware coherency scheme. Each of the cache line has 2 flags, "Valid" and "Dirty", which is just like an ordinary processor. The states of each line can be explicitly manipulated by software. The compiler prevents multiple processor to enter "Modified" state at the same time. The compiler also forbids load/store operation during "Stale" state.

One of the example of such architecture is the RP2 which is 8 cores multicore processor developed by Renesas Electronics, Hitachi Ltd., Waseda University and supported by NEDO Multi core processors for real-time consumer electronics project [14]. Other than the RP2, we also evaluated our proposed method on a simple multicore Nios II CPU on Altera FPGA. A more detailed information about these processors is available in the performance evaluation section.

Our proposed method can be applied to almost any kind of inter-processor networking as our method uses the main shared memory for synchronization and does not rely on communication between CPU cores.

5. The OSCAR Compiler API for Non-Cache-Coherent Architecture

The OSCAR compiler has an API to support its operation [15]. The OSCAR compiler takes a sequential C programs and generates a parallelized C code with OSCAR API directives. The API consists of many directives for different purposes, like power management, cache operation, and so on. Some of the directives are subset of OpenMP directives such as `parallel section`, `flush` and `critical`.

To support the NCC architecture, several new directives are added to the API [15].

- `noncacheable`: to indicates that a variable must not be stored in cache.
- `aligncache`: to indicates that a variable should be aligned to the beginning of the cache line.
- `cache_writeback`: to writeback dirty cache line to the memory.
- `cache_selfinvalidate`: to change the state of the

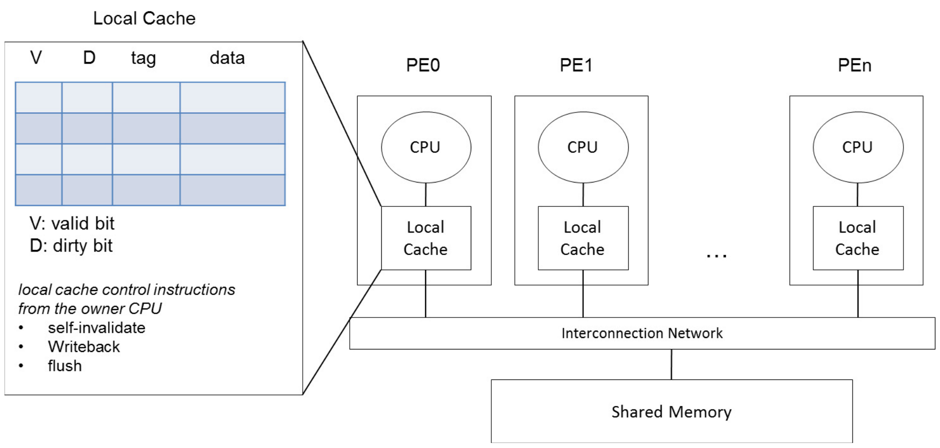


Fig. 8 Proposed cache coherency scheme.

current cache line in a local core to invalid.

- `complete_memop`: to mark the end of cache operation.

The OSCAR Compiler generated code with OSCAR API annotation is then processed by an API Analyzer to convert the directives to a platform specific driver level function call such as `alt_dcache_flush` on NiosII platform to implement OSCAR API whole data cache flush directives `flush`. Another example is, on RP2 platform, the OSCAR API `aligncache` directive is converted into a proper backend compiler API call on Renesas SuperH C Compiler to align the cache while taking the actual configured cache line size into account.

6. Performance Evaluation

This section shows the performance of the proposed method on an embedded multicore the Renesas RP2, also soft CPU core Nios II and an Intel Xeon SMP cache coherent machine for benchmark programs from SPEC, NAS Parallel Benchmark and MediaBench.

6.1 The RP2 Processor

The Renesas RP2 is an 8-core embedded processor configured as two 4-core SH-4A SMP clusters, with each cluster having MESI protocol, jointly developed by Renesas Electronics, Hitachi Ltd., Waseda University and supported by METI/NEDO Multicore Processors for Real-time Consumer Electronics Project in 2007 [14]. Each processor core has its own private cache. However, there is no hardware coherence controller between the cluster for hard real-time applications like automobile engine control; hence, to use more than 4 cores across the cluster, a software based cache coherency must be used. The MESI hardware coherence mechanism can be disabled completely. The RP2 board as configured for this experiment has 16kB of data cache with 32-byte line size and 128MB shared memory. The local memory, which was provided for hard real-time control application was not used in this evaluation. The RP2 processor supports several native instructions in NCC mode: write-back operation (`OCBWB` instruction), cache invalidate (`OCBBI` instruction), cache flush (`OCBP` instruction).

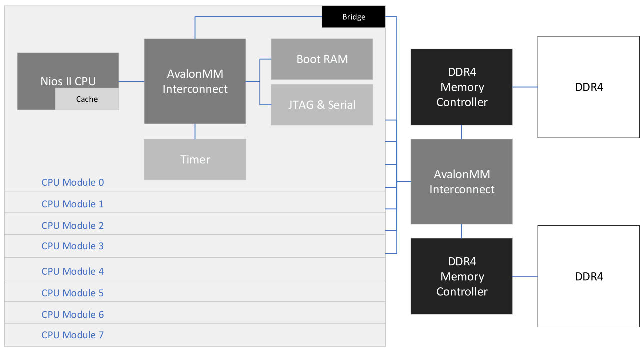


Fig. 9 Diagram of the 8-core Nios II SoC.

6.2 Eight-Core Nios II System on FPGA

To test the scalability of our system, we designed our multicore system based on Altera Nios II soft CPU core. Figure 9 is a simplified block diagram of the multicore system. The system consists of eight Nios II CPUs with 16kB data cache and a separate instruction cache for each cpu core. The caches are configured as a direct-mapped, 32 bytes wide, write-back cache. All 8 cores share two banks of 1 GB DDR4 1066 MHz main memory connected over two Altera External Memory Controllers. The CPU cores are connected through a series of Avalon bus, adapter, clock bridge and Altera Platform Designer generated interconnect. No special interconnect is required. Also, no any hardware-based cache coherency mechanism is implemented. The system is designed and sythesized on Altera Quartus II 18.1 and implemented on Altera Arria 10 SoC Board Development Kit.

Each of the Nios II CPU has its own memory sections to run the Altera HAL which provides basic C library support. Then the benchmark programs and data were run from the main shared memory. The Nios II CPU has several native instructions to manage its data cache manually; cache flush (`flushd` instruction) and cache invalidate (`initd` instruction).

6.3 Intel Xeon Based SMP Cache Coherent Machine

To further investigate the performance impact of the proposed false sharing mitigation on an SMP cache coherent system, we ran the same OSCAR Compiler parallelized benchmark application on such system with the self invalidation method turned off. The SMP system used is Intel Xeon E5-2699 v3 CPU with 128 GB of DDR4 memory.

6.4 Benchmark Applications

To evaluate the performance of the proposed method, 10 benchmark applications from SPEC2000, SPEC2006, NAS Parallel Benchmark (NPB), and Mediabench II were used. The selection of the benchmark program is somewhat limited due to several factors. Our implementation runs on a bare-metal configuration as most operating systems currently rely on an SMP system to function correctly. Also, the Nios II SoC implementation currently does not have a proper file system, preventing SPEC benchmark and Mediabench from running. The RP2 board has only 128MB of shared memory, which also limits the data sets used in the benchmark. These benchmarks were written in C and converted to Parallelizable C [21] by hand, which is similar to MISRA-C [22] used in the embedded field.

Parallelizable C is a guideline to write a C program that allows a parallelizing compiler to extract the full potential of a conventional compiler framework for parallelization and data locality optimization, mainly targeting arrays and loops. Converting to Parallelizable C is very straight forward and trivial. For example, in the case of lbm benchmark, only 3 lines of pointer related modification are required, and the rest of the lbm benchmark can be parallelized automatically. Aside from lbm, most other benchmark program does not need to be changed at all.

Then, for the next step, these programs were compiled by the OSCAR Compiler. The output C program by the OSCAR compiler was compiled by the Renesas SuperH C Compiler (SH C) for the RP2 processor, by Nios II GCC for the eight-core Nios II CPU and GCC for the Intel Xeon platform. The SPEC benchmark programs were run in their default configuration and datasets except lbm, which were run with $100 \times 100 \times 15$ matrix. All NPB benchmarks were configured with CLASS S data size considering the size of the RP2 processor board main shared memory size (128 MB). But for the evaluation on Intel Xeon machine, we used Class A and Class B data size in order to measure the running time reliably.

6.5 Experimental Results and Analysis

Figures 10, 11, and 12 show the speedups by multiple cores of the proposed method on RP2 Processor and Nios II multicore system. The blue bars show the baseline performance on a RP2 Symmetric Multiprocessor (SMP) cluster with MESI hardware coherence control. The orange bars show the performance of RP2 processor with the proposed software coherence control method on NCC architecture. The single core performance on SMP machine was selected as the baseline. Meanwhile, the green bars show the performance of Nios II multicore systems. The Nios II system performances are measured relative to its respective single thread performance.

As shown in Fig. 10, the software coherence provides a similar speedup compared to hardware-based coherence. Moreover, the compiler-controlled coherency allows 8 core CPU execution, which is formerly impossible due to the lack of hardware coherence mechanism. It also gives a

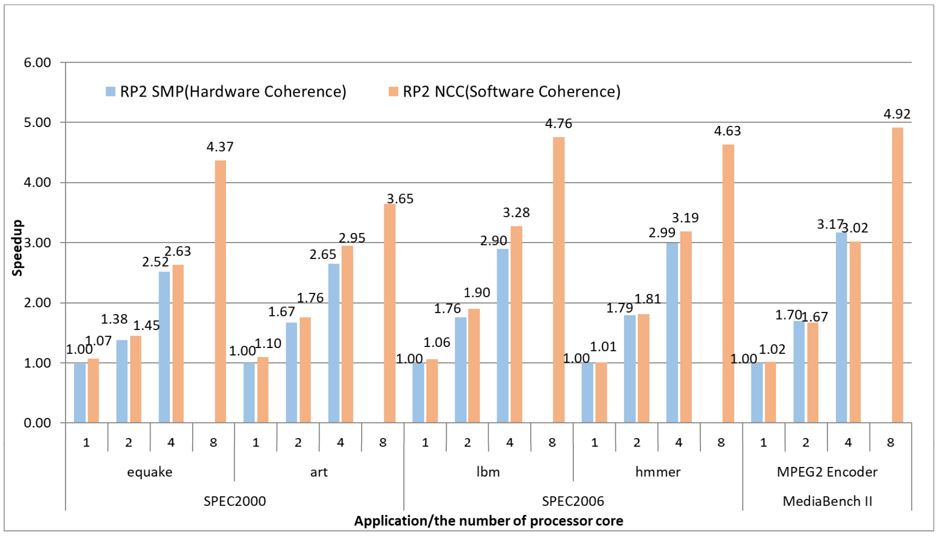


Fig. 10 The performance of the proposed method on RP2 Processor for SPEC Benchmark and MediaBench.

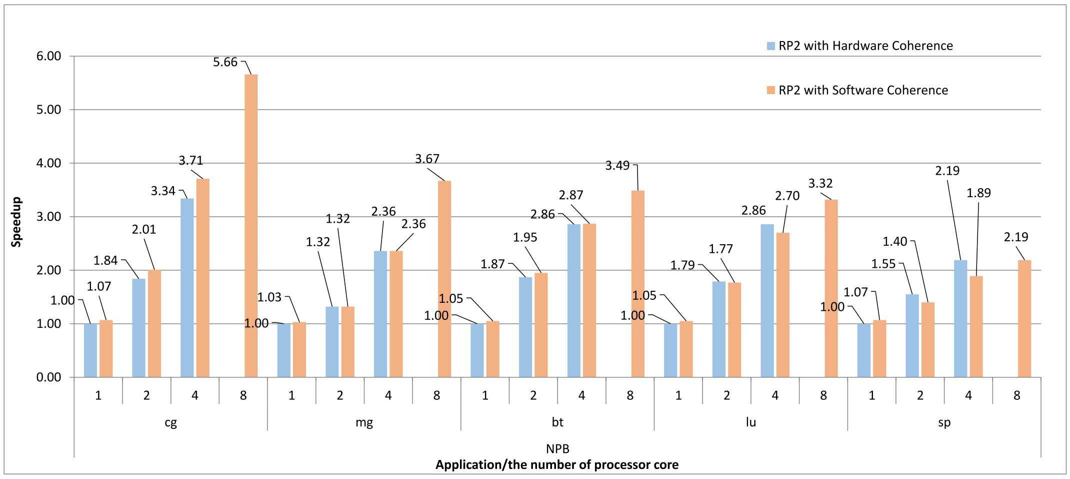


Fig. 11 The performance of the proposed method on RP2 Processor system for NAS Benchmark.

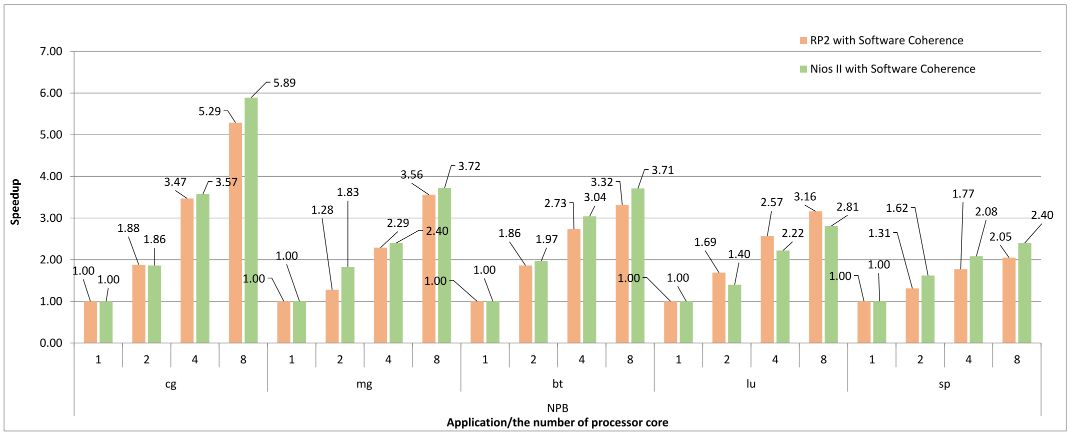


Fig. 12 The performance comparison of RP2 Processor and Nios II system for NAS Benchmark.

respectable speedup compared to the 4 core performance. The performance of the proposed software cache coherence method give us roughly 4%–14% better performance compared to hardware based coherence. With hardware based coherence, an overhead is imposed due to frequent transmission of invalidation packet between processor cores via the interconnection bus. On the other hand, the software does not require the transmission of such packet as the compiler will insert self-invalidate instruction to the required processor core. For "art", "quake" and "lbm" benchmark, is positively affected by this performance benefit of software based coherence. In Fig. 11, we see similar performance gain for NPB, albeit not as strong as the SPEC benchmark. NPB cg is a conjugate-gradient calculation with many DOACROSS loops. Selective cache operation allows it to have a better performance by reducing the number of self-invalidation. In Fig. 12, we can see the Nios II soft CPU SoC could run the benchmark with respectable speed up automatically. Without the compiler support, writing parallel a program with a good speedup for this platform is difficult.

Then, we would like to see the impact of each mitigation method on the performance regardless the usefulness of the computation. We varied the combination of the method used to run the benchmark. Figure 13 depicts the performance impact of each proposed methods: **SMP** is a normal shared memory architecture with native hardware based coherence. This is selected as the baseline of the measurement. **Stale Data Handling with Hardware Coherence:** stale data handling method with hardware based coherence control still turned on. We can see here that the performance is negatively impacted. This is expected since stale data handling method is just wasting CPU clock and adding unnecessary delay to the program with hardware coherence control still active. But we can see here the effect of the stale data handling negatively impacted the performance of "lbm". **False Sharing with Hardware Coherence:** false sharing handling which comprises data alignment, cache line aligned data decomposition, and other layout transformation with hardware coherence control still turned on. We can see here that there is almost no significant performance impact. The cache line wasting effect is insignificant. In certain benchmarks, most notably "lbm", this approach improves the performance. This is to be expected since false sharing is also bad even for hardware based cache coherence

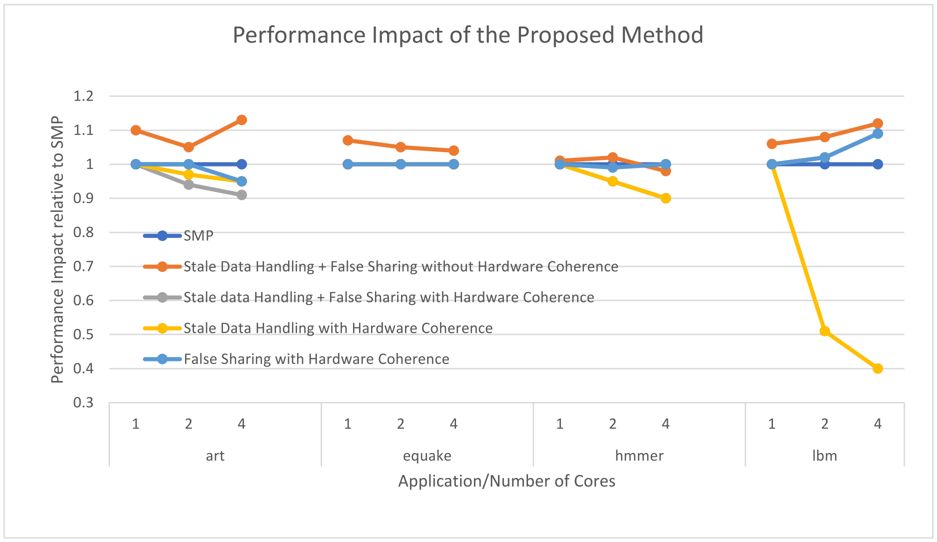


Fig. 13 The performance impact of software cache coherence.

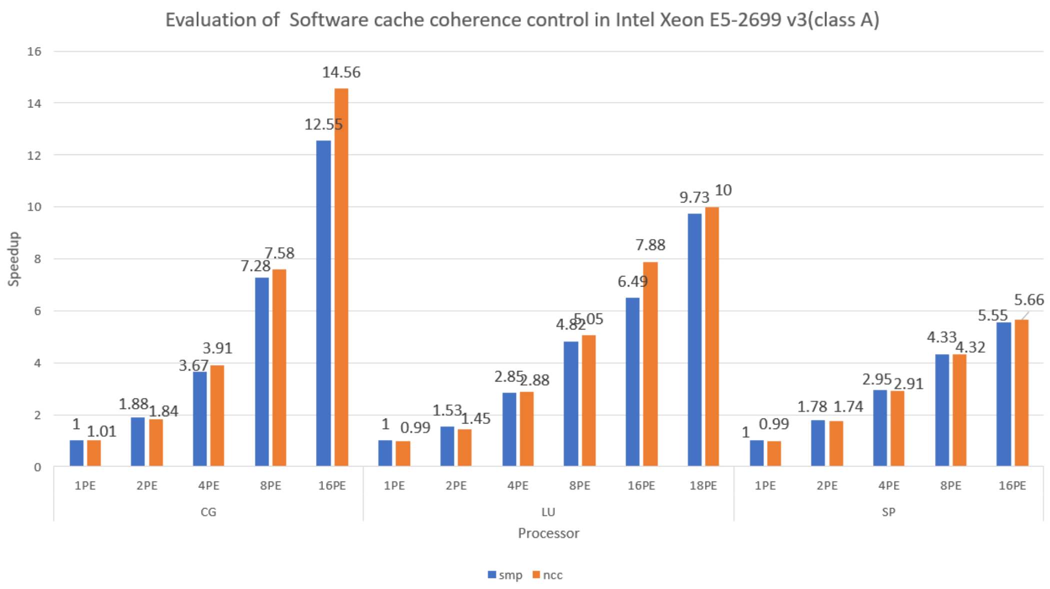


Fig. 14 The performance benefit of false sharing avoidance in Intel SMP cache coherent machine on NAS Parallel Benchmark class A data size.

control. Removing false sharing problem will improves the performance of a hardware based coherence control. **Stale data Handling + False Sharing with Hardware Coherence:** this graph measures the overhead of both proposed method for handling stale data and false sharing with hardware coherence still active. **Stale Data Handling + False Sharing without Hardware Coherence:** this graph shows the performance of the proposed method with hardware coherence control completely turned off.

Even on an SMP cache coherent machine, while the self invalidation scheme is not useful, the false sharing prevention still helps to improve the performance as seen on Figs. 14 and 15. But we can observe a slowdown on class B data size due to a reduction in cache utilization.

In this paper we proposed several method to avoid false sharing. Based on the usage statistic on Fig. 16, a simple array alignment is deemed sufficient by the compiler. Array padding only consist of about 2% and non-cacheable buffer is almost never used. This is due to the relative size of the array in the benchmark program is usually much larger compared to the cache-line size, hence false sharing happened rarely only at the edge case.

7. Conclusions

This paper proposes a method to manage cache coherency by an automatic parallelizing compiler for non-coherent cache architecture. The proposed method incorporates control dependence, data dependence analysis and automatic parallelization by the compiler. Based on the analyzed

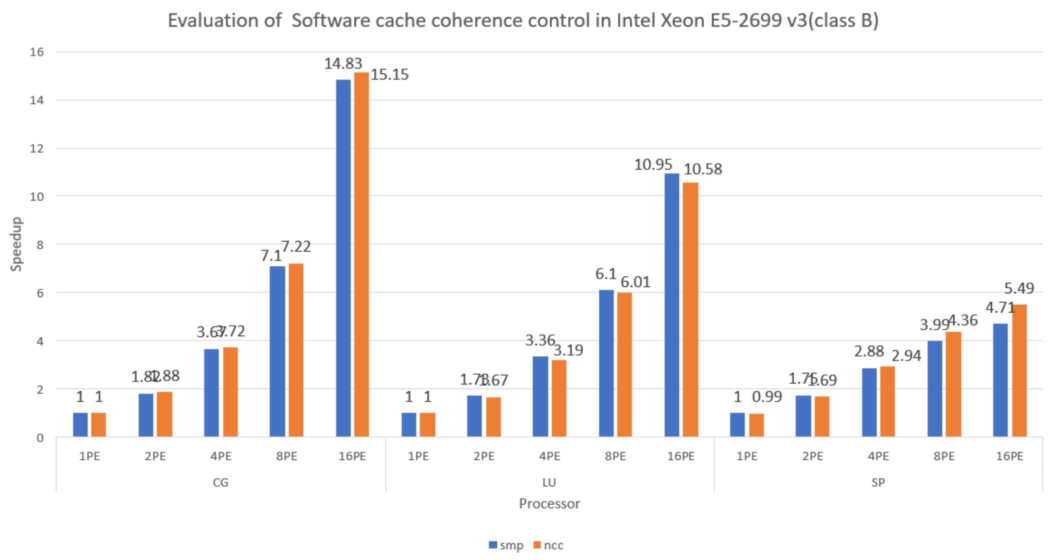


Fig. 15 The performance benefit of false sharing avoidance in Intel SMP cache coherent machine on NAS Parallel Benchmark class B data size.

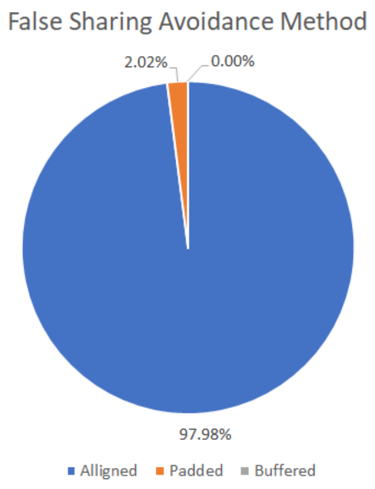


Fig. 16 Typical False Sharing Mitigation Method Usage average on NPB.

stale data, any possible false sharing is identified and resolved. Then, software cache control code is automatically inserted. The proposed method is evaluated using 10 benchmark applications from SPEC2000, SPEC2006, NAS Parallel Benchmark and MediaBench II on Renesas RP2 8 core multicore processor and a custom 8-core Nios II multicore processor on Altera FPGA. The performance of the NCC architecture with the proposed method is similar or better than the hardware-based coherence scheme. For example, the hardware coherent mechanism using MESI protocol gave us 2.52 speedup on 4 cores RP2 against one core SPEC2006 "equake", 2.9 times speedup on 4 cores RP2 for SPEC2006 "lbm", 3.34 times speedup on 4 cores RP2 for NPB "cg", 3.17 times speedup on 4 cores RP2 for MediaBench II "MPEG2 Encoder". On the other hand, the proposed software cache coherence control method implemented on OSCAR Multigrain Parallelizing Compiler gave us 2.63 times on 4 cores RP2, 4.37 times on 8 cores RP2 speedup for "equake", 3.28 times on 4 cores RP2 and 4.76 times on "lbm", 3.71 times on 4 cores RP2 and 5.66 times on 8 cores RP2 for "cg", 3.02 times on 4 cores RP2 and 4.92 times on 8 cores RP2 for "MPEG2 Encoder".

The proposed method also allows us to automatically parallelize and easily run the benchmark program on 8-cores Nios II CPU which is not designed for cache coherent operation. Those result shows the proposed software coherent control method allow us to obtain comparative performance with the traditional hardware coherence control mechanism for the same number of processor cores. Furthermore, it provides a good speedup automatically and easily for any number of processor cores without the hardware coherent control mechanism, while so far application programmers had to spend huge development time to use the non-coherent cache architecture. Further research on optimizing data reuse on limited cache space with software-controlled cache coherence should be conducted. Also, a scalability evaluation for more CPU cores on FPGA should be further investigated.

Acknowledgments

Part of this paper is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO). Boma A. Adhi's PhD study is sponsored by Hitachi Global Foundation Scholarship.

References

[1] J. Archibald and J.-L. Baer, "Cache coherence protocols: Evaluation using a multiprocessor simulation model," ACM Trans. Computer Systems, vol.4, no.4, pp.273–298, 1986.
[2] J. Laudon and D. Lenoski, "The SGI origin: A ccNUMA highly scalable server," The 24th Annual International Symposium on Computer Architecture, pp.241–251, June 1997.

[3] M.M.K. Martin, M.D. Hill, and D.J. Sorin, "Why on-chip cache coherence is here to stay," Commun. ACM, vol.55, no.7, pp.78–89, July 2012.
[4] G. Chrysos, "Intel® Xeon Phi™ coprocessor—The architecture," Intel Whitepaper, 2014.
[5] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "TILE64 - Processor: A 64-core SoC with mesh interconnect," 2008 IEEE International Solid-State Circuits Conference, Digest of Technical Papers, pp.88–598, Feb. 2008.
[6] D. Abts, S. Scott, and D.J. Lilja, "So many states, so little time: Verifying memory coherence in the Cray X1," Proc. International Parallel and Distributed Processing Symposium, April 2003.
[7] H. Cheong and A.V. Veidenbaum, "Compiler-directed cache management in multiprocessors," Computer, vol.23, no.6, pp.39–47, June 1990.
[8] Y.-C. Chen and A.V. Veidenbaum, "Comparison and analysis of software and directory coherence schemes," Proc. 1991 ACM/IEEE Conference on Supercomputing, Supercomputing '91, New York, NY, USA, pp.818–829, ACM, 1991.
[9] L. Choi and P.C. Yew, "A compiler-directed cache coherence scheme with improved intertask locality," Proc. 1994 ACM/IEEE Conference on Supercomputing, Supercomputing '94, Los Alamitos, CA, USA, pp.773–782, IEEE Computer Society Press, 1994.
[10] X. Yuan, R. Melhem, and R. Gupta, "A timestamp-based selective invalidation scheme for multiprocessor cache coherence," Proc. 1996 International Conference on Parallel Processing, pp.114–121, Aug. 1996.
[11] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S.V. Adve, V.S. Adve, N.P. Carter, and C.-T. Chou, "DeNovo: Rethinking the memory hierarchy for disciplined parallelism," 2011 International Conference on Parallel Architectures and Compilation Techniques, pp.155–166, Oct. 2011.
[12] S. Kaxiras and A. Ros, "A new perspective for efficient virtual-cache coherence," Proc. 40th Annual International Symposium on Computer Architecture, ISCA '13, New York, NY, USA, pp.535–546, ACM, 2013.
[13] S. Tavarageri, W. Kim, J. Torrellas, and P. Sadayappan, "Compiler support for software cache coherence," 2016 IEEE 23rd International Conference on High Performance Computing (HiPC), pp.341–350, Dec. 2016.
[14] M. Ito, T. Hattori, Y. Yoshida, K. Hayase, T. Hayashi, O. Nishii, Y. Yasu, A. Hasegawa, M. Takada, M. Ito, H. Mizuno, K. Uchiyama, T. Odaka, J. Shirako, M. Mase, K. Kimura, and H. Kasahara, "An 8640 MIPS SoC with independent power-off control of 8 CPUs and 8 RAMs by an automatic parallelizing compiler," 2008 IEEE International Solid-State Circuits Conference, ISSCC 2008, Digest of Technical Papers, pp.90–598, Feb. 2008.
[15] K. Kimura, A. Hayashi, H. Mikami, M. Shimaoka, J. Shirako, and H. Kasahara, "OSCAR API v2.1: Extensions for an advanced accelerator control scheme to a low-power multicore API," 17th Workshop on Compilers for Parallel Computing, 2013.
[16] H. Kasahara, K. Kimura, B.A. Adhi, Y. Hosokawa, Y. Kishimoto, and M. Mase, "Multicore cache coherence control by a parallelizing compiler," 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), pp.492–497, July 2017.
[17] K. Ishizaka, M. Obata, and H. Kasahara, "Cache optimization for coarse grain task parallel processing using inter-array padding," Languages and Compilers for Parallel Computing, ed. L. Rauchwerger, Lecture Notes in Computer Science, vol.2958, pp.64–76, Springer Berlin Heidelberg, 2004.
[18] K. Kimura, M. Mase, H. Mikami, T. Miyamoto, J. Shirako, and H. Kasahara, "OSCAR API for real-time low-power multicores and its performance on multicores and SMP servers," International Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science, vol.5898, pp.188–202, Springer, 2009.
[19] Y. Solihin, Fundamentals of Parallel Multicore Architecture, 1st ed., Chapman & Hall/CRC, 2015.
[20] Y. Kishimoto, M. Mase, K. Kimura, H. Kasahara, et al., "Evaluation of software cache coherency control method by an automatic parallelizing compiler," IPSJ SIG Technical Report (HPC), vol.2014, no.19, pp.1–7, 2014 (in Japanese).
[21] M. Mase, Y. Onozaki, K. Kimura, and H. Kasahara, "Parallelizable C and its performance on low power high performance multicore processors," Proc. 15th Workshop on Compilers for Parallel Computing, 2010.
[22] MISRA-C: 2012: Guidelines for the Use of the C Language in Critical Systems, HORIBA MIRA, 2019.

Boma A. Adhi received his Bachelor and Master of Electrical Engineering from University of Indonesia in 2010 and 2013 respectively. He is now a Ph.D. student of Computer Science and Engineering of Waseda University. His research interests include non-cache coherent architecture and system on a chip.

Tomoya Kashimata received his B.E. in Computer Science and Engineering from Waseda University in 2018. He is now a master course student of Computer Science and Engineering of Waseda University. His research interests include hardware accelerators for irregular memory access.

Ken Takahashi received his B.E. in Computer Science and Engineering from Waseda University in 2018. He is now a master course student of Computer Science and Engineering of Waseda University. His research interests include hardware accelerators for irregular memory access.

Keiji Kimura received the B.S., M.S. and Ph.D. degrees in electrical engineering from Waseda University in 1996, 1998, 2001, respectively. He was an assistant professor in 2004, associate professor of Department of Computer Science and Engineering in 2005, and professor in 2012 at Waseda University. He was also a department head of CSE from 2015 to 2016 and an assistant dean of FSE from 2016 to 2017. He is a recipient of 2014 MEXT (Ministry of Education, Culture, Sports, Science and Technology in Japan) award. His research interest includes microprocessor architecture, multiprocessor architecture, multicore processor architecture, and compilers. He is a member of IPSJ, ACM and IEEE. He has served on program committee of conferences such as ICCD, ICPADS, ICPP, LCPC, IISWC, ICS, IPDPS, and PACT.

Hironori Kasahara is IEEE Computer Society (CS) 2018 President and a senior executive vice president of Waseda University. He received a BS in 1980, a MSEE in 1982, a Ph.D. in 1985 from Waseda University, Tokyo, joined its faculty in 1986, and has been a professor of computer science since 1997 and a director of the Advanced Multicore Research Institute since 2004. He was a visiting scholar at University of California, Berkeley, in 1985 and the University of Illinois at Urbana-Champaign's Center for Supercomputing R&D from 1989 to 1990. Kasahara received the IEEE Fellow in 2017, CS Golden Core Member Award in 2010, IEEE Eta Kappa Nu Professional Member in 2017, IFAC World Congress Young Author Prize in 1987, IPSJ Fellow in 2017 and Sakai Special Research Award in 1997, and a Science and Technology Prize of the Japanese Minister of Education, Culture, Sports, Science and Technology in 2014. He led Japanese national projects on parallelizing compilers and embedded multicores, and has presented 216 papers, 176 invited talks, and 48 international patents. His research on multicore architectures and software has appeared in 608 newspaper and Web articles. He has served as a chair or member of 258 society and government committees, including the CS Board of Governors; Executive Committee; Planning Committee; chair of CS Multicore STC and CS Japan chapter; associate editor of IEEE Transactions on Computers; vice PC chair of the 1996 ENIAC 50th Anniversary International Conference on Supercomputing; general chair of LCPC 2012; PC member of SC, PACT, and ASPLOS; board member of IEEE Tokyo section; and member of the Earth Simulator committee.