Software Cache Coherent Control by Parallelizing Compiler

Boma A. Adhi(✉), Masayoshi Mase, Yuhei Hosokawa, Yohei Kishimoto, Taisuke Onishi, Hiroki Mikami, Keiji Kimura, and Hironori Kasahara

Department of Computer Science and Engineering, Waseda University, Tokyo, Japan
{boma, onishi, hiroki, kimura, kasahara}@kasahara.cs.waseda.ac.jp

Abstract. Recently multicore technology has enabled development of hundreds or thousands core processor on a single chip. However, on such multicore processor, cache coherence hardware will become very complex, hot and expensive. This paper proposes a parallelizing compiler directed software coherence scheme for shared memory multicore systems without hardware cache coherence control. The general idea of the proposed method is that an automatic parallelizing compiler parallelize coarse grain task, analyzes stale data and line sharing in the program, then solves those problems by simple program restructuring and data synchronization. The proposed method is a simple and efficient software cache coherent control scheme built on OSCAR automatic parallelizing compiler and evaluated on Renesas RP2 with 8 SH-4A cores processor. The cache coherence hardware on the RP2 processor is only available for up to 4 cores. The cache coherence hardware can also be turned off for non-coherence cache mode. Performance evaluation was performed using 10 benchmark programs from SPEC2000, SPEC2006, NAS Parallel Benchmark (NPB) and MediaBench II. The proposed method performed as good as or better than hardware cache coherence scheme while still provided correct result as the hardware coherent mechanism. For example, the proposed software cache coherent control (NCC) gave us 2.63 times speedup for SPEC 2000 equake with 4 cores against sequential execution while got only 2.52 times speedup for 4 cores MESI hardware coherent control. Also, the software coherence control gave us 4.37 speed up for 8 cores with no hardware coherent mechanism available.

1 Introduction

For many years, cache coherent SMPs have been widely used as the core component of all classes of machines, from smartphones, IoTs, PCs, and embedded systems all the way to HPC systems. Typically, a hardware cache coherence mechanism, either snoopy or directory based, is employed to ensure every change made into a shared line in one processor's private cache is always reflected in the content of all private cahces so that coherency is maintained. Hardware cache coherence mechanism scales well for current generation multicore processor [1],

L. Rauchwerger (Ed.): LCPC 2017, LNCS 11403, pp. 17–25, 2019.
https://doi.org/10.1007/978-3-030-35225-7_2

e.g. Intel Xeon Phi [2], Tilera Tile64 [3]. However, despite its common usage among current generation multicore processor, this kind of hardware will too complex, hot and expensive for the upcoming hundreds to thousands core massively parallel multicore system to avoid the complexity of the hardware based cache coherency.

Research on software controlled started in the late 80's. One of the prominent contributions is [4] which proposed fast selective invalidation scheme and version control scheme for compiler directed cache coherence. More recent research [5] proposes a compiler support for software based cache coherency. A practical and ready to use solution for software based coherence is yet to be proposed.

This paper proposes a new software coherent control scheme to guarantee coherency by avoiding stale data and false sharing. This method is novel, simple, powerful and give us delivers the same performance as the hardware implementation of cache coherency. Next, we present an overview of OSCAR's parallelization strategy followed by a discussion of the techniques to handle sate data and false sharing.

2 Software Cache Coherent Control by Parallelizing Compiler

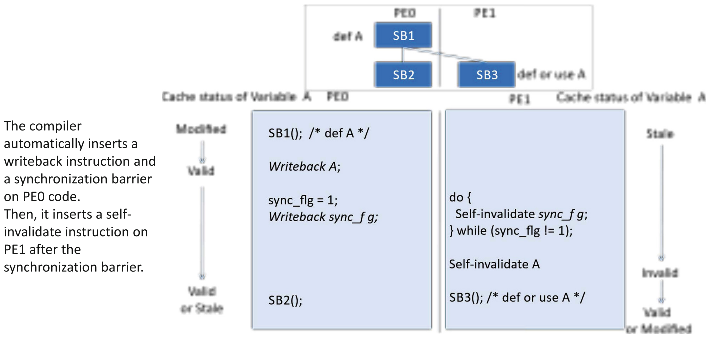
The proposed method is built into the OSCAR parallelizing compiler, which analyzes and decomposes programs into tasks using control flow and data dependence. Based on the data access range of each task, the compiler addresses stale data and false sharing. Our proposed method may be applied to almost any kind of interprocessor networking as our method uses the main shared memory for synchronization and does not rely on communication between CPU cores. Next, we present an overview of OSCAR's parallelization strategy followed by a discussion of the techniques to handle sate data and false sharing.

2.1 Coarse-Grain Task Parallelization

The OSCAR compiler is a multi-grain parallelizing compiler. The compiler generates C or Fortran program extended with invocations to OSCAR API [6] routines in this way, OSCAR compiler generated parallel multicore code that can be compiled for any shared memory multicore available in the market using a conventional compiler. 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. A Macro Task Graph (MTG) is generated by analyzing the earliest executable condition of every MT and tracing the control dependencies and data dependencies among MTs on the MFG. Based on this information, the compiler generates appropriate cache coherence control code [7].

2.2 Handling the Stale Data Problem

A hardware based cache coherence ensures information on changes made to the data in one of the CPU cores cache line is propagated to other cores so that each copy of this data in other cores can be invalidated. The process of notifying the other processors in a snoopy based cache coherence may impact the performance of the processor. With directory based mechanism, simultaneous access to directory may become a performance bottleneck. Meanwhile, without any hardware cache coherence, these bottlenecks do not exist, but access to stale data should be manually managed by the compiler.



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.

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

Writeback A;

sync_flg = 1;
Writeback sync_f g;

SB2();
```

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

Self-invalidate A

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

Fig. 1. Cache control code inserted by the compiler to prevent reference to stale data.

Based on the coarse grain scheduling result, to manage 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.

Figure 1 is an example of the compiler generated code to prevent stale data reference. Core 0 defines a new value for a shared variable, A. The compiler automatically inserts a writeback instruction and an assignment to a synchronization flag on core 0's code. The compiler also inserts a self-invalidate instruction on core 1 right after testing the synchronization flag. The compiler then schedules the task in a way that minimize the delay caused by the synchronization. In addition, if multiple cores retain the same data at the same time, the compiler schedules all cores in way to prevent the data to be simultaneously updated. These cache manipulation instructions are inserted only for Read-after-Write data dependence. Meanwhile for Write-after-Read and Write-after-Write, only synchronization instruction is inserted. By using this approach, stale data can be avoided. Moreover, the overhead caused by the transmission of invalidate packets associated with hardware based mechanism can be eliminated.

2.3 Handling the False Sharing Problem

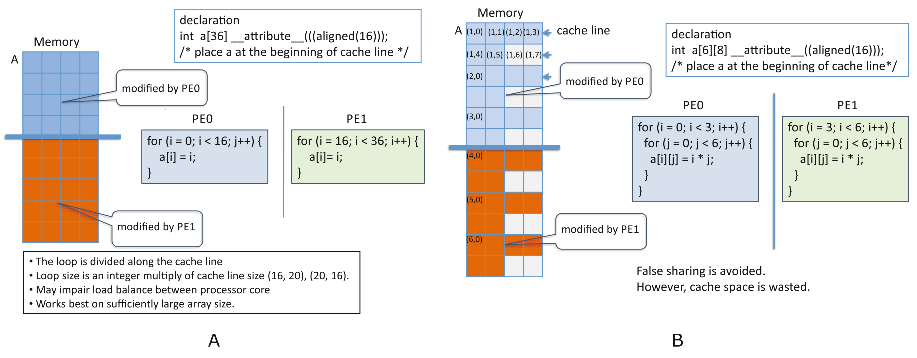
False sharing is a condition in which two or more data items share a single cache line. Whenever one of those data is updated, inconsistency may occur. This is due to the granularity of the cache writeback mechanism usually works with line instead of byte or word sized. To address this problem, OSCAR compiler uses one of the following four mechanisms:

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.

Cache Aligned Loop Decomposition. OSCAR compiler applies loop decomposition which consist in partitioning the iteration space of a loop to create several tasks. Instead of assigning the same number of iterations to each partial task, the compiler decomposes loops taking into account the cache line size as seen in Fig. 2(A).

Array Padding. It is not always possible to partition a two-dimension array cleanly along cache line boundaries. This happens when the lowest dimension of the array is not an integer multiply of the cache line size. In this case, OSCAR compiler inserts padding to the end of the array to match the cache line size. This approach is depicted in Fig. 2(B). It should be noted that this approach may also waste cache space.

Data Transfer Using Non-cacheable Buffer. When cache aligned loop causes a significant load imbalance or array padding consumes too much cache space or none of the former approaches cannot be applied, OSCAR compiler uses a non-cacheable buffer. The compiler designates a an area in the main memory that should not be copied to the cache and places the shared data in that area. Figure 3 depicts the usage of non-cacheable buffer.



```
declaration
int  a[36] __attribute__(((aligned(16)));
/* place a at the beginning of cache line */
```

```
for (i = 0; i < 16; j++) {
   a[i] = i;
}
```

```
for (i = 16; i < 36; i++) {
   a[i]= i;
}
```

- The loop is divided along the cache line
- Loop size is an integer multiply of cache line size (16, 20), (20, 16).
- May impair load balance between processor core
- Works best on sufficiently large array size.

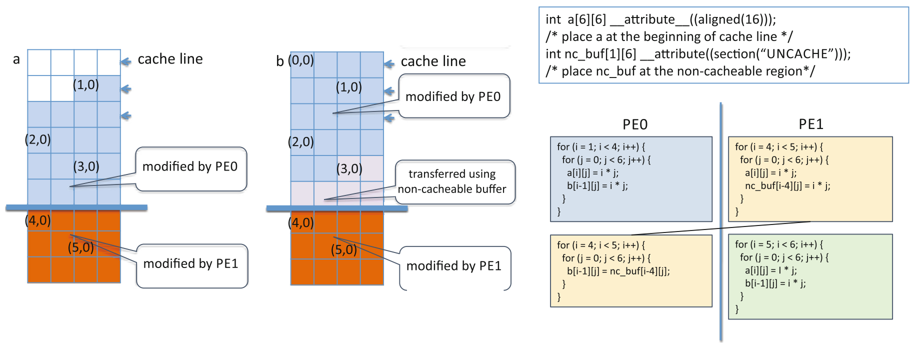
```
declaration
int  a[6][8] __attribute__((aligned(16)));
/* place a at the beginning of cache line*/
```

```
for (i = 0; i < 3; i++) {
  for (j = 0; j < 6; j++) {
    a[i][j] = i * j;
  }
}
```

```
for (i = 3; i < 6; i++) {
  for (j = 0; j < 6; j++) {
    a[i][j] = i * j;
  }
}
```

False sharing is avoided.
However, cache space is wasted.

Fig. 2. (A)Cache alligned 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.



```
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*/
```

```
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++) {
    a[i][j] = i * j;
    nc_buf[i-4][j] = i * j;
  }
}
```

```
for (i = 4; i < 5; i++) {
  for (j = 0; j < 6; j++) {
    b[i-1][j] = nc_buf[i-4][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. 3. Non-cacheable buffer is used to avoid false sharing.

3 Performance of the Software Coherent Control on Embedded Multicore

This section shows the performance of the proposed method on an embedded multicore the Renesas RP2 for benchmark programs from SPEC, NAS Parallel and MediaBench.

3.1 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. and Waseda University under support from the METI/NEDO Multicore Processors for Real-time Consumer Electronics Project in 2007 [8]. 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 16 kB 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: writeback operation (`OCBWB` instruction), cache invalidate (`OCBBI` instruction), cache flush (`OCBP` instruction).

3.2 Benchmark Applications

To evaluate the performance of the proposed method, we used 10 benchmark applications from SPEC2000, SPEC2006, NAS Parallel Benchmark (NPB) and Mediabench II. While the selection of the benchmark program is somewhat limited due to the main memory size of the current board, the selected benchmark represents several different types of scientific and multimedia application. These benchmarks were written in C and converted to Parallelizable C [9] which is similar to MISRA-C used in embedded field. Then these programs were compiled by the OSCAR source-to-source automatic parallelizing compiler. The output C program by the OSCAR compiler was compiled by the Renesas SuperH C Compiler (SH C) as the backend compiler as mentioned before. 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 small shared memory or main memory (128 MB) of the RP2 processor.

3.3 Experimental Results and Analysis

Figure 4 is a graph showing the speedups by multiple cores of the proposed method on RP2 Processor. The lighter bars show the baseline performance on a Symmetric Multiprocessor (SMP) cluster with MESI hardware coherence control. The darker bars show the performance of the proposed software coherence control method on NCC architecture. The single core performance on SMP machine was selected as the baseline.

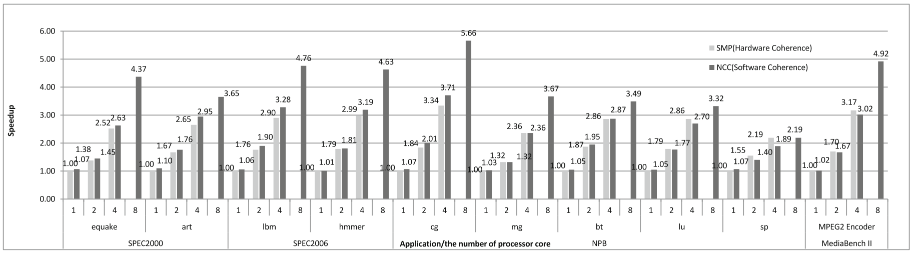


Fig. 4. The performance of the proposed method on RP2 Processor.

Figure 5 depicts the performance impact of each proposed methods. Five different plots are presented for four of the benchmark programs executing in 1, 2 and 4 cores: **SMP** is a normal shared memory architecture with native hardware based coherence. This is selected as the baseline of the measurement.

Stale data handling: stale data handling method with hardware based coherence control still turned on. We can see here that the performance is negatively impacted. This is to be expected since stale data handling method wastes CPU cycles since the hardware already handles this problem. But we can see here the effect of the stale data handling negatively impacted the performance of lbm.

False sharing avoidance: 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 control. Removing false sharing problem will improves the performance of a hardware based coherence control. **NCC (hardware coherence):** this graph measures the overhead of both proposed method for handling stale data and false sharing with hardware coherence still active. **NCC (software coherence):** this graph shows the performance of the proposed method with hardware coherence control completely turned off.

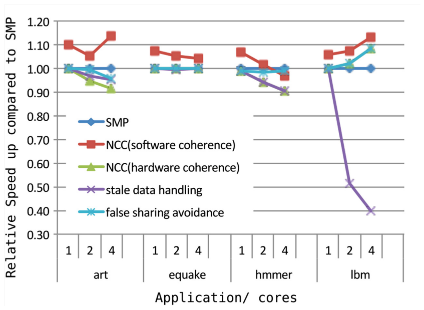


Fig. 5. The performance impact of software cache coherence.

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. The data structure of "lbm" is also unique that it has a lot of false sharing. We can see here that our proposed false sharing avoidance method improves the performance significantly.

While not offering huge performance benefit, compared to hardware based approach, the proposed method has enabled the usage of 8 cores in RP2 processor which does not have cache coherence mechanism. Before, using our proposed method, it was impossible to run an application with 8 cores without very complicated hand-tuned optimization.

4 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 stale data, any possible false sharing is identified and resolved. Then, software cache control code is automatically inserted. The proposed method was evaluated using 10 benchmark applications from SPEC2000, SPEC2006, NAS Parallel Benchmark and MediaBench II on Renesas RP2 8 core multicore processor. The performance of the NCC architecture with the proposed method was similar or better than the hardware based c herenc mple, the hardware coherent mechanism using MESI protocol gave us 2.52 speedup on 4 core against one core SPEC2006 "equake", 2.9 times speedup on 4 cores for SPEC2006 "lbm", 3.34 times speedup on 4 cores for NPB "cg", 3.17 times speedup on 4 cores for MediaBench II "MPEG2 Encoder". On the otherhand, the proposed software cache coherence control method implemented on OSCAR Multigrain Parallelizing Compiler gave us 2.63 times on 4 cores, 4.37 times on 8 cores speedup for "equake", 3.28 times on 4 cores and 4.76 times on "lbm", 3.71 times on 4 cores and 5.66 times on 8 cores for "cg", 3.02 times on 4 cores and 4.92 times on 8 cores for "MPEG2 Encoder". Those result shows the proposed software coherent control method allow us to obtain comparable performance with the MESI hardware coherence control mechanism for the same number of processor cores. Furthermore, it gives us good speedup automatically and quickly for many processor cores without the hardware coherent control mechanism although up until now application programmers had to spend huge development time to use the non-coherent cache architecture.

Acknowledgement. Masayoshi Mase and Yohei Kishimoto are currently working for Hitachi, Ltd. and Yahoo Japan Corp respectively. Their works contained in this paper were part of their study at Waseda University. Boma Anantasatya Adhi is part of Universitas Indonesia and currently a PhD student at Waseda University supported by Hitachi Scholarship.

References

1. Martin, M.M.K., Hill, M.D., Sorin, D.J.: Why on-chip cache coherence is here to stay. Commun. ACM **55**(7), 78–89 (2012)

2. Chrysos, G.: Intel & ®Xeon Phi Coprocessor-the Architecture. Intel Whitepaper (2014)
3. Bell, S., et al.: TILE64 - processor: a 64-Core SoC with mesh interconnect. In: 2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers, pp. 588–598, February 2008
4. Cheong, H., Veidenbaum, A.V.: Compiler-directed cache management in multiprocessors. Computer **23**(6), 39–47 (1990)
5. Tavarageri, S., Kim, W., Torrellas, J., Sadayappan, P.: Compiler support for software cache coherence. In: 2016 IEEE 23rd International Conference on High Performance Computing (HiPC), pp. 341–350, December 2016
6. Kimura, K., Hayashi, A., Mikami, H., Shimaoka, M., Shirako, J., Kasahara, H.: OSCAR API v2. 1 : extensions for an advanced accelerator control scheme to a low-power multicore API. In: 17th Workshop on Compilers for Parallel Computing (2013)
7. Kasahara, H., Kimura, K., Adhi, B.A., Hosokawa, Y., Kishimoto, Y., Mase, M.: Multicore cache coherence control by a parallelizing compiler. In: 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), vol. 01, pp. 492–497, July 2017
8. Ito, M.: An 8640 mips soc with independent power-off control of 8 cpus and 8 rams by an automatic parallelizing compiler. In: 2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers, pp. 90–598, February 2008
9. Mase, M., Onozaki, Y., Kimura, K., Kasahara, H.: Parallelizable c and its performance on low power high performance multicore processors (2010)