Studies on Compiler Controlled Cache Coherency for Multicore Processors

マルチコアプロセッサのためのコンパイラによるキャッシュコヒーレンシー制御に関する研究

March 2020

Boma Anantasatya ADHI

Studies on Compiler Controlled Cache Coherency for Multicore Processors

マルチコアプロセッサのためのコンパイラによる
キャッシュコヒーレンシー制御に関する研究

March 2020

Waseda University

Graduate School of Fundamental Science and Engineering

Department of Computer Science and Communications Engineering,
Research on Advanced Computing Systems

Boma Anantasatya ADHI

Abstract

Most of the modern multicore processors use a hardware-based mechanism to maintain cache coherency. However, on a larger scale multicore, a hardware-based cache coherency circuitry gets exceedingly complex, thus generating more heat and challenging to verify. The complexity also drives the development cost and time up. A hardware-based cache coherence may not scale well to a large number of cores expected to be found on future SMP machines. On the other side of the spectrum, adding a hardware-based mechanism sometimes cannot be justified due to limited resources or specific requirements, such as in a real-time embedded system or soft processor cores implemented in FPGA Without a hardware-based cache coherency mechanism, writing a program for such a system is overwhelmingly difficult. Therefore, in this thesis, a compiler controlled cache coherency based on OSCAR Compiler is proposed. The compiler parallelizes the coarse grain task, then analyzes for true-sharing and false-sharing in the program. The compiler then solves true-sharing problems with self-invalidation and data synchronization. Likewise, it avoids false-sharing problems with simple data restructuring. Ten benchmark programs from SPEC2000, SPEC2006, NAS Parallel Benchmark, and MediaBenchII are compiled with the proposed method. The compiled binaries then are run on Renesas RP2, an 8-cores SH-4A processor, a custom 8-cores Altera Nios II SoC, and 8-core RISC-V SoC on Altera Arria 10 FPGA. The proposed method successfully runs benchmark programs on those multicores and achieve similar performance as the hardware-based coherence mechanism. For example, NAS Parallel Benchmark “cg” obtains 3.71 times speedup on RP2 without hardware coherence support versus 3.34 times speedup with hardware coherence turned on. Also, the same benchmark program runs with 5.66 times speedup on 8-cores RP2, 5.89 times on 8-cores Nios II SoC, and 3.68 times

on 4-cores RISC-V SoC, which all are otherwise impossible to run without hardware cache coherency.

Contents

1 Introduction **9**

1.1 Background 10

1.2 Related Works & Proposal 12

1.3 Thesis Organization 15

2 OSCAR Compiler and OSCAR API **19**

2.1 OSCAR Compiler 20

2.2 The OSCAR API 23

3 Compiler-Controlled Cache Coherence for Multicore CPU **27**

3.1 Cache Coherency 28

3.2 Stale Data Problem 29

3.2.1 Stale Data Handling 30

3.2.2 Selective Cache Operation in Loop Parallelization 33

3.3 False Sharing Problem 33

3.3.1 Variable Alignment and Array Expansion 38

3.3.2 Cache Aligned Loop Decomposition 38

3.3.3 Array Padding 38

3.3.4 Data Transfer Using Non-cacheable Buffer 38

3.4 Software Cache Coherent Control by OSCAR Compiler 40

4 Development of Non-Cache Coherent Architecture Evaluation Platforms **45**

4.1 The RP2 Processor 47

4.2 Eight-Core Nios II System on FPGA 49
4.3 Quadcore RISC-V Based Non-Cache Coherent SoC 51

5 Performance Evaluation **57**
5.1 Benchmark Applications 58
5.2 Experimental Results and Analysis 59
5.2.1 Relative Speedup 63
5.2.2 Performance Impact of the Proposed Method 64
5.2.3 Performance Benefits of False Sharing Avoidance on SMP Machine 69

6 Conclusions **71**
6.1 Summary of Works 72
6.2 Future Works 73

List of Figures

1-1 The Future of Computing. Image from [31] 10

2-1 Macro Flow Graph and Macro Task Graph Example. Image from [18] 21

2-2 OSCAR Architecture. Image from [23] 25

3-1 The stale data problem. Image from [20] 30

3-2 Cache control code inserted by the compiler to prevent reference to stale data. Image from [20] 31

3-3 Coherency Protocol Managed by the Compiler 32

3-4 False Sharing Detection Process. Image from [21] 35

3-5 The false sharing problem. Image from [20] 37

3-6 (A)Cache aligned loop decomposition is applied to a one-dimension matrix to avoid false sharing. (B)Array padding is applied to a two-dimensional matrix to avoid false sharing. Images from [20] 39

3-7 Non-cacheable buffer is used to avoid false sharing. Image from [20] . 41

3-8 Proposed compilation sequence. Image from [21] 42

3-9 Pseudocode for detecting and mitigating stale data and false sharing. Image from [1] 43

4-1 Target Architecture. Image from [20] 48

4-2 Renesas RP2 8-core Embedded Multicore Processor. Image from [38] 50

4-3 Diagram of the 8-core Nios II SoC. Image from [24] 52

4-4 Diagram of the 4-core RISV-V SoC showing only the data connections 56

5-1 The performance of the proposed method on RP2 Processor for SPEC Benchmark and MediaBench. Image from [1] 60

5-2 The performance of the proposed method on RP2 Processor system for NAS Benchmark. Image from [1] 61
5-3 The performance comparison of RP2 Processor, Nios II and RISC-V multicore for NAS Benchmark. 62
5-4 The performance impact of software cache coherence. Image from: [1] 65
5-5 The performance benefit of false sharing avoidance in Intel SMP cache coherent machine on NAS Parallel Benchmark class A data size. Image from [1] 67
5-6 The performance benefit of false sharing avoidance in Intel SMP cache coherent machine on NAS Parallel Benchmark class B data size. Image from [1] 68

List of Tables

2.1 List of OSCAR API Directives 23

4.1 Custom CSR for Interacting with the Cache Subsystem 54

5.1 List of Benchmark Programs 58

Chapter 1

Introduction

1.1 Background

Nowadays, processor clock speed increase has been slowing down, marking the imminent end of Moore's law[31]. As a result, instead of pursuing a higher frequency single-core processor, the industry has shifted towards a multicore processor. The multicore processors do not have run as fast as the single-core processor, but they perform more works through parallelism.[12].

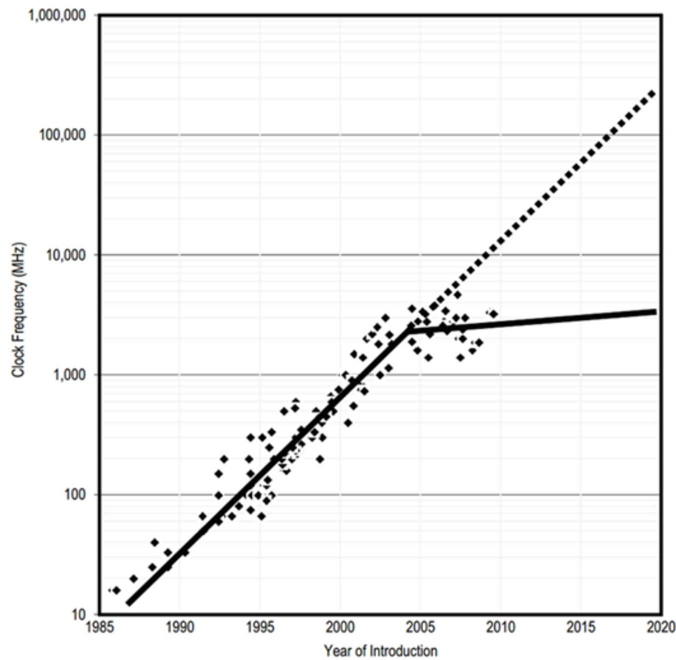


Figure 1-1: The Future of Computing. Image from [31]

Most modern multicore processors today are based on a shared memory system. Shared memory is a low-level communication paradigm that allows multiple processor cores to work and communicate through shared memory space. Shared memory multiprocessor is easier to program compared to other multiprocessing paradigms, such as distributed memory systems, and due to this fact, shared memory machines are ubiquitous[5], from tiny IoT microcontrollers, smartphones, industrial and automotive realtime embedded systems, gaming PCs, to cloud servers. Even nowadays, most distributed memory systems in TOP-500 Supercomputers consist of a massive array of shared memory machines.

In a shared memory system, processor cores are usually equipped with a smaller

but faster memory, which is called a cache, to improve the read and write performance to the shared memory. If the processor requires some data from the main memory, the data will be fetched from the main memory and then stored in the cache for further use. If the processor requires more of that data, it will search for the cached content first before searching the main memory. For every change or update to the cached data, the cache controller is responsible for flushing the data back to the main memory. Frequently or recently used data typically reside in the cache, effectively hiding the latency of the slower and larger shared main memory. This way, the cache creates an illusion of large but fast shared memory[14].

In a multicore processor, the cache can be organized in many ways. The cache can be private for each core, shared between multiple cores or a combination of both. Private cache is better in terms of latency, especially when multiple cores share the same data. Meanwhile, a shared cache offers a better hit rate as there is no duplicate data stored in multiple caches belong to different cores. Multiple levels of caches are also commonly found. Usually, for a performance reason, a multicore processor has a private cache for each core, and if necessary, another level of caches shared between multiple cores[22].

Fundamentally, a private cache in a multicore processor has a cache inconsistency problem. It happens as multiple cores keep a copy of the same shared data in their private cache. If a core updates the value of the shared variable in its cache, the newly updated value will not be seen by other cores until the new value is written-back to the shared memory, and all other cores are notified of the change. Different processor cores may see a different value for the same variable on its private cache, thus causing a problem called cache inconsistency. In order to avoid such inconsistency, a cache coherency control mechanism is required. Most modern multicore processors utilize a hardware-based mechanism, either a snooping-based mechanism found in older multicore with a low core count or a more elaborate directory-based mechanism found in a larger multicore processor[26].

The hardware-based cache coherency currently dominates the market. It is rare for a modern processor relies on a software-based coherency outside the few specific applications. The main reason is, the hardware-based cache coherence is commonly

believed to provide better performance than the software-based approach. Another reason is compatibility, as the hardware-based approach is software transparent, thus writing a program for a hardware-based cache coherence multicore system is trivial compared to manually controlling cache coherency by hand. No intricate software layer or compiler is required, effectively maintaining compatibility with older single-core processor[26].

On a larger scale multicore, a hardware-based cache coherency circuitry gets exceedingly complex, thus generating more heat and reducing efficiency[10]. It is also challenging to verify[30]. The complexity also drives the development cost and time up. A hardware-based cache coherence may not scale well to a large number of cores expected to be found on future SMP machines [15]. On the other side of the spectrum, some applications require the software-based mechanism, for example, a realtime system, in which the hardware cache coherency mechanism may introduce undesired uncertainty[28][33] or soft processor cores implemented in FPGA. A multicore implemented in FPGA with a hardware cache coherence utilizes about 50% more logic area compared to their NCC counterpart[4].

Without a hardware-based cache coherency mechanism, writing a program for such a Non-Cache Coherent (NCC) system is overwhelmingly complex, as the programmer has to maintain the cache coherency manually. Improper manual cache coherency optimization may even lead to degraded performance compared to single-core application[13].

There are two fundamental problems in the NCC system that must be taken care of by the compiler. First, the true-sharing or stale-data problem. This problem occurs when multiple cores share the same data in their cache. The second is the false-sharing problem. This problem happens when independent data shares a cache line, thus treated as a single entity by the cache update mechanism.

1.2 Related Works & Proposal

Researchers have long known the scalability and complexity problems of the hardware-based coherence. Therefore, the research on non-coherent cache shared memory sys-

tem has been started since the late '80s through the '90s, although less research can be found from the late '90s through the 2000s due to the supremacy of hardware-based cache coherence[26]. From the late 2000s onward, the focus of the research shifted toward software cache coherency on massively parallel processors, distributed systems, processors to accelerators, and soft processor core on FPGA. Most of the research listed here successfully solved the stale-data problem, but almost none handles the false sharing problem, which may degrade the system performance.

The researches in the late '80s and early '90s tried to eliminate or reduce the complexity of hardware cache coherence by using a software approach, but a little to none of those research made their way to a real production processor. One of the pioneering research is [22], which proposed a fast-selective invalidation scheme and version control scheme for compiler directed cache coherence. It proposed a rule for cache coherency violations and used a dependency graph to detect the violation and to issue invalidation instruction. It required no communication between processors. The correctness of the proposed method was proven, but it was not implemented on hardware. The research was latter continued [9], which proposed the fast-selective invalidation scheme and version control scheme for compiler directed cache coherence. Furthermore, they showed that their proposal was capable of maintaining comparable performance to a directory-based scheme [7].

Another interesting early research is [39], which proposes a timestamp-based method to allow simpler hardware cache coherence to be implemented in a processor. At that time, hardware coherence was very well known and broadly implemented. However, for some multiprocessor architecture, namely, Dance-hall architecture, both snoopy based and directory-based scheme could not be employed due to lack of broadcast mechanism between processors.

The research by [25] emulated directory-based coherence in software. It proposed a mechanism to maintain and propagate directory information between processors. The information includes the list of page readers and writers, and also the state of the page. This structure, which is called the coherent map, is stored at the home node. This structure is uncached, and the latency for accessing this structure is hidden with each lock operation.

Another alternative approach was by using the data-flow algorithm to detect stale data reference[8], and subsequent work to improve temporal and spatial locality [11]. It improved the task locality by using an epoch timer, and each cache data has a tag of its creation time. It then proposed a set of rules based on an epoch-flow graph to mark if a particular entry is stale or not. The catches are the following; it required a special epoch counter, and it treated an array as a single entity, which means an entire array is invalidated even if only part of it is modified.

One of the latest research on NCC [36] utilized the polyhedral model to identify the read-write pattern at cache line granularity precisely. Moreover, for non-regular programs, it employed an inspector-executor paradigm to maintain coherency. It showed a comparable performance against the hardware-based cache coherence scheme on simulator. This research handles false sharing well. However, the polyhedral method employed only considers a single loop nest at a time.

Based on the previous research above, none of the research handles the two problems in NCC architecture well and implemented in hardware. This thesis explains the work to extend the capability of the OSCAR Compiler to generate parallel code for a non-cache-coherent SMP system automatically. Several elemental compilation techniques are integrated into OSCAR Compiler in order to solve two main problems in a non-cache coherent multicore architecture. The stale data problem is solved by self-invalidation and synchronization; meanwhile, false sharing is avoided by data alignment, array expansion, array padding, and non-cacheable buffer and stale data by self-invalidation and synchronization. A new compiler module is created for this purpose. The new module utilizes the parallelized-sections data and def-use data from the OSCAR Compiler framework to solve both fundamental non-cache-coherent architecture problems.

The concept of this research was granted a US patent[21]. The idea and the early evaluation of the proposed method were presented in IEEE COMPSAC 2017[20]. In LCPC 2017[2], the algorithm was elaborated further, and additional analysis was done. In this thesis, more hardware platforms are developed, namely, the Nios II and RISC-V based softcore SoC on FPGA in addition to the Renesas RP2 multicore. The newly developed hardware platform was designed without a cache coher-

ence mechanism yet still providing a respectable speed up. An additional test on the Intel machine was done to investigate the impact of the proposed method on a cache-coherent system. Several benchmark programs were run, and the result shows that the proposed method obtains better or comparable performances compared to the hardware-based approach. In conclusion, this research enables automatic parallelization with a simple, easy to program and efficient Non-Cache-Coherent (NCC) manycore processor. Right now the effort starts from a simple embedded multicore and soft processor core for FPGA, but the same principle could be applied to a larger system more cores.

1.3 Thesis Organization

This thesis is organized as follows:

Chapter 1, "Introduction", this current chapter. It provides the background and explains the objectives of this research. This chapter also outlines the significance of this research by comparing related works.

Chapter 2, "OSCAR Compiler and OSCAR API", introduces the OSCAR Compiler and OSCAR API. The OSCAR Compiler is a source to source automatic parallelizing compiler. The compiler takes a C program, and then it decomposes the program into coarse-grain tasks. It then analyzes the control flow and the data dependency between those tasks. Based on this information, the compiler parallelizes the task and, later on, inserts the cache manipulation code for the coherency control and finally outputs a parallelized code with API directives annotation. The OSCAR Compiler is supplemented with the OSCAR API that consists of many directives for different purposes, like platform-specific functions, power management, and cache operation. The OSCAR API converts directives in the annotated code into the correct functions or driver calls.

Chapter 3, "Compiler-Controlled Cache Coherence for Multicore processor", discusses two fundamental problems in NCC systems, "stale data/true sharing" and "false sharing". Stale data is a state when a processor core updates a shared data in its cache, but other processor core cannot get the updated value until the data is flushed back

to the main shared memory. In order to handle this problem, as one of the cores updates a variable in its cache, the compiler then inserts self-invalidation instruction into the code segment in which other cores access the variable. The self-invalidate is an instruction for invalidating a specific line in the cache, which forces other cores to get the latest value from shared memory, thus eliminating inconsistency. The second problem, “false sharing”, happens when two or more independent data reside in a single cache line. If one of those data is changed, inconsistency may occur; this is because the granularity of the cache writeback mechanism is at the line level instead of a single byte or word. The compiler performs several elemental data restructuring operations to prevent unrelated variables from sharing a single cache line. For a scalar or small-sized one-dimensional array, usually, cache alignment is sufficient. In the case of a multi-dimensional array, the compiler performs array expansion or array padding, and for the case in which all other methods are inapplicable, the compiler uses a non-cacheable buffer. Later on, this chapter discusses the necessary changes to implement the coherency control in the OSCAR Compiler and the OSCAR API.

Chapter 4, “Development of Non-Cache Coherent Architecture Evaluation Platform”, explains three different test platforms used in this research, the RP2 multicore processor and two custom SoC based on Nios II and RISC-V. The Renesas RP2 is an 8-core embedded processor which comprises of two 4-core SH-4A SMP clusters, with each cluster has its hardware coherency domain. A regular program can run with up to four cores, but a software approach is necessary for cache coherency beyond the 4-core cluster. This research was also partially motivated by this RP2 limitation. The “Nios II-based SoC” is a simple SoC generated entirely in Altera Platform Designer. Nios II is a soft processor core which is intended for single-core operation in Altera FPGA without any multicore or hardware cache coherency support. Designing multicore Nios II SoC is relatively easy, but writing a program for it is difficult without any coherency mechanism. RISC-V is a relatively new and promising open-source hardware instruction sets architecture. There are a lot of available RISC-V processor implementations in the market, but unfortunately, there is no synthesizable and functional multicore RISC-V SoC with hardware cache coherence in the market.

Chapter 5, “Performance Evaluation of Compiler-Controlled Cache Coherence”,

explains the performance evaluation of the Compiler Controlled Cache Coherency. Ten benchmark programs from three benchmark suites, NAS Parallel Benchmark, SPEC, and MediaBench, are compiled by the OSCAR Compiler with NCC support. The proposed method achieves similar performance as the hardware-based coherence mechanism. The proposed method also allows us to parallelize automatically and run the benchmark program on NCC 8-cores Nios II SoC and 4-cores RISC-V SoC as if they are a cache-coherent SMP machine. For example, NAS Parallel Benchmark "cg" obtains 3.71 times speedup on RP2 without hardware coherence support versus 3.34 times speedup with hardware coherence turned on. Also, the same benchmark program runs with 5.66 times speedup on 8-cores RP2, 5.89 times on 8-cores Nios II SoC, and 3.68 times on 4-cores RISC-V SoC, which all are otherwise impossible to run without hardware cache coherency.

Chapter 6, "Conclusion", concludes the thesis and discusses the future of this research.

Chapter 2

OSCAR Compiler and OSCAR API

This chapter provides an overview of the OSCAR Compiler and OSCAR API. The whole work described in this thesis is based on and built on top of the OSCAR Compiler.

OSCAR Compiler is a source-to-source multigrain parallelizing compiler[18]. It takes sequential C or Fortran programs, and then it generates a parallelized program with OSCAR API directives[23]. The OSCAR API converts directives in the annotated code into the correct functions or driver calls. Then the resulting code is compiled with any standard compiler to generate machine code. This way allows us to generate parallel multicore codes just using a sequential compiler for any shared-memory multicore available in the market.

2.1 OSCAR Compiler

OSCAR Compiler is a multigrain parallelizing compiler. It parallelizes sequential C or Fortran program in multiple levels of graininess: coarse-grain task parallelization between loops and function calls, near-fine-grain parallelization between statements inside the basic blocks, and the loop-level parallelization.

The compiler starts the compilation process by decomposing the source program into three types of coarse-grain tasks, or Macro Tasks (MTs), i.e., Basic Blocks (BBs), Repetition Blocks (RBs), and Subroutine Blocks (SBs). BBs are a straight line of codes that usually consist of simple assignments without any branch, RBs are loops, and SBs are function calls. The compiler will then hierarchically decompose RBs and SBs into simpler MTs if any coarse-grain task still exists within those tasks.

As all MTs for the source program are generated, the compiler generates a Macro Flow Graph (MFG). MFG is a variant of a control flow graph that explicitly represents both control flow and data dependencies among MTs. Figure 2-1 is an example of MFG. In this MFG(N, E, C), a set of nodes N contains all MTs. Edges E has two kinds of edges, i.e., $E_C F$ represents control flow shown as dotted edges, and E_D represents data dependence shown as solid edges. The set of C represents conditional branch inside each MT shown as a small circle inside MTs. The direction of the graph is always top to bottom.

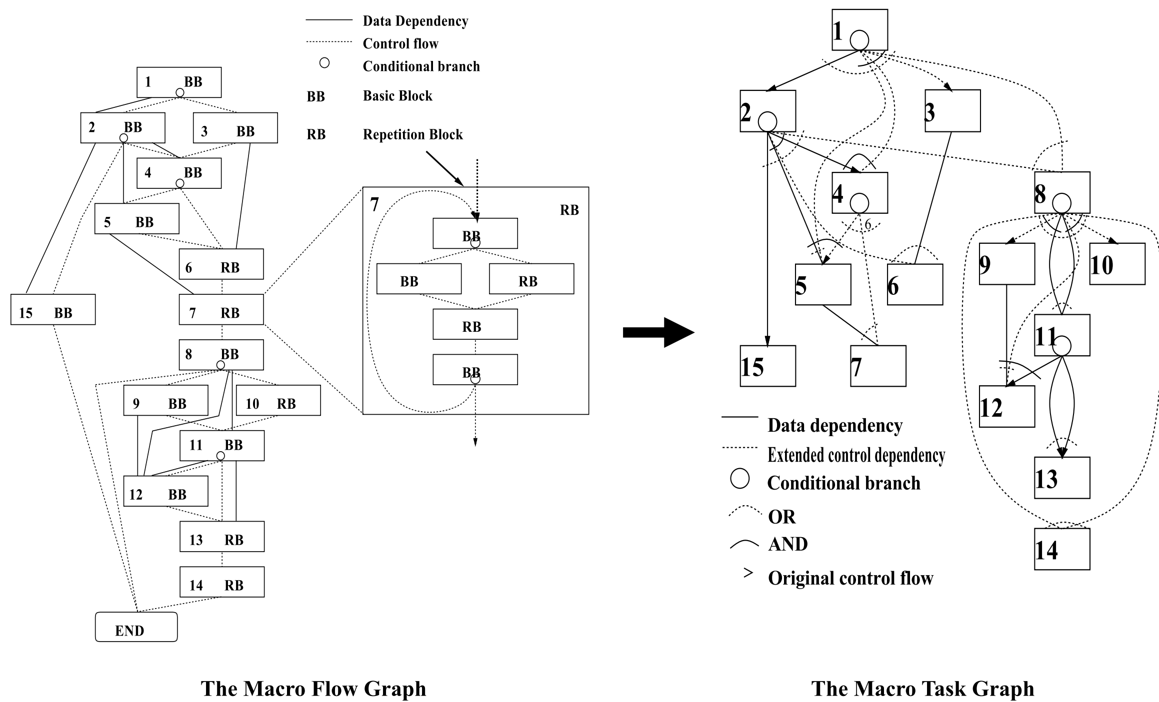


Figure 2-1: Macro Flow Graph and Macro Task Graph Example. Image from [18]

In order to extract maximum parallelization from the program, both control flow and data dependence must be taken into account. The Earliest Execution Condition (EEC) of an MT is a condition in which all preceding constraints caused by both control flow and data dependence are satisfied. A macro task i (MT_i) can be executed if the following conditions are satisfied: If MT_i has a data dependency on macro task j (MT_j), then MT_i can only be executed after MT_j finishes. If MT_i is on a control dependence on MT_j, then MT_i can be executed as soon as the branch direction on MT_j is known. This is possible because both MT_i and MT_j are executed on different cores. In general, ECC can be formulated as follows: (MT_j, on which MT_i has data-dependences, branches to MT_i) $\wedge$ ((every macro task on which MT_i has data dependencies, $\mathrm{MT}_k : 0 \leq k \leq |N|$ are completed) $\vee$ (MT_k will not be executed)).

As the EECs for all MTs are known, the compiler schedules all MTs either by static or dynamic scheduling. Typically MTs are statically scheduled to avoid run-time overheads caused by the scheduler. If the MTG has run-time uncertainty, i.e., conditional branching, I/O wait, or varying MT execution time, the MTs in it are dynamically scheduled. For this NCC research, OSCAR Compiler generates static scheduling to preserve the order of memory operation among all MTs, which will be further discussed in the next chapter[20]. Profiling data from each different processor architecture are also taken into account to improve the static scheduling performance. For most of the cases, the OSCAR Compiler adopts Critical Path/Data Transfer/Most Immediate Successors First (CP/DT/MISF) scheduling for its static scheduling mechanism. In addition to that, the compiler may choose CP/ETF/MISF, ETF/CP/MISF, and DT/CP/MISF scheduling[19]. The compiler then put together the codes for all MTs based on the scheduling result. It also inserts data transfer and synchronization to the required places. This way ensures the memory state is consistent at the end of each MT execution. Based on the scheduling result, the compiler then optimizes the code further, for example, by grouping execution of task working on the same data sets on one processor core, managing power gating for least used CPU cores, the NCC module, and many other purposes. In the end, the compiler generates the code for each processor core.

Table 2.1: List of OSCAR API Directives

List of Directives in OSCAR API				
Parallel Execution API	Data Transfer API	Power Control API	Accelerator API	Cache Control API
parallel sections	dma_transfer	fvcontrol	accelerator_task_entry	cache_writeback
flush	dma_contiguous_parameter	get_fvstatus	Hint directives for	cache_seflinvalidate
critical	dma_stride_parameter	Synchronization API	OSCAR Compiler	complete_memop
execution	dma_flag_check	groupbarrier	accelerator_task	noncacheable
Memory Mapping API	dma_flag_send	Timer API	oscar_comment	aligncache
threadprivate		get_current_time		
distributedshared				
onchipshared				

2.2 The OSCAR API

OSCAR API is designed to support OSCAR Compiler's operation. The API is designed based on the OpenMP subset to provide portability over many kinds of shared memory multicore systems[23]. The early version of OSCAR API (v 1.0) was designed for the OSCAR Architecture but worked with any shared memory system. The OSCAR Architecture consists of several multicore chips with external/off-chip Chip Shared Memory (CSM), and each chip also has an on-chip CSM. Each core has its own Local Program Memory, Local Data Memory (LDM) for private data, and Distributed Shared Memory (DSM) for flags and shared data. The architecture is depicted in Figure 2-2.

The OSCAR API v1.0 takes three directives from OpenMP, namely "`parallel sections`", "`flush`", and "`critical`". These directives allow single-level parallel thread execution with OSCAR API. In addition to those, one OpenMP directive was extended, and 12 new directives were created. The new directives control the memory mapping, DMA transfer, power control, synchronization, and timer.

OSCAR API v2.0 added accelerator directives, which included heterogeneous multicore and NCC architecture support. The NCC support comprises of 5 new directives:

- `cache_writeback`: This directive forces a writeback of a cache line to the main shared memory.
- `cache_selfinvalidate`: This directive changes the state of the current cache line in a local core to invalid.
- `complete_memop`: This directive generates a "memory fence" like instruction to preserve the order of the memory operation.

- `noncacheable`: This directive indicates that a variable must not be stored in the cache.
- `aligncache`: This directive indicates that a variable must be aligned to the beginning of the cache line.

The full list of API can be seen in Table 2.1. The implementation details of the API are discussed in the next chapter.

The OSCAR Compiler takes a sequential C program as input. Then the compiler analyzes the program and generates a parallelized C program annotated with OSCAR API directives. The API translator then converts those directives into the appropriate run-time library or driver calls. In some cases, the translator directly converts the directives into specific in-line assembly instructions for the required functionality. Depending on the target architecture, for platforms with OpenMP support, the API translator converts the directives directly to OpenMP directives, which is essentially the same, differing only in the prefix, or to other parallel execution techniques for platforms without OpenMP support. Finally, the translated code can be compiled with any available C compiler for the target platform.

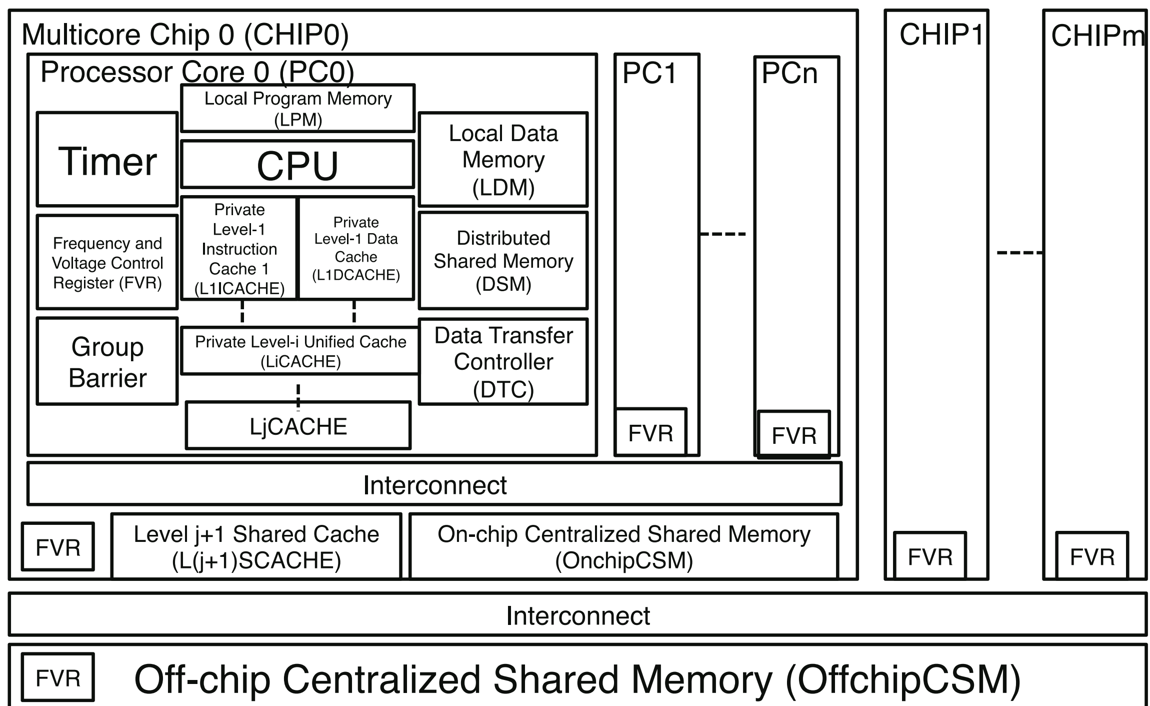


Figure 2-2: OSCAR Architecture. Image from [23]

Chapter 3

Compiler-Controlled Cache Coherence for Multicore CPU

This chapter discusses the cache coherency, problems in an NCC system, and changes made to the OSCAR Compiler to support the NCC architecture.

3.1 Cache Coherency

In a shared-memory multicore with a private cache, multiple copies of a shared variable may exist in the caches of several different cores. Cache coherency is a state which guarantees that any changes made to one of the copies are propagated to all other copies in the correct order.

One of the most commonly implemented cache coherence protocol is the MESI protocol[29] or any of its variants. It is invalidation-based, and it supports writeback cache. The acronym MESI represents four states of a cache line.

- "**Modified**" state represents a particular cache line that is dirty or modified by a local core and different from the value in the main memory. This cache line must be written back at one point in the future before other cores to read the outdate value in the main memory.
- "**Exclusive**" state represents a particular cache line that is exclusively only present in that local core's cache, and it is in a clean state. It will be changed to "Modified" if the core makes any change to it.
- "**Shared**" state represents a particular cache line that is stored in multiple cores' caches, and it is in a clean state. If another copy in other core's cache is modified, then it will be changed to "Invalid".
- "**Invalid**" state represents a particular cache line that is no longer valid because another core modifies its own copy of the same cache line.

Typically, a hardware mechanism maintains all those states transparently from the software point of view. Such a mechanism 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 snooping-based cache coherence may impact the performance of the

processor[32]. With a directory-based mechanism, the complexity of the design increases significantly for the many-cores processor[30]. Meanwhile, without any hardware cache coherence, these bottlenecks do not exist, but the compiler must manage access to stale data.

While most multicore processors in the market have a dedicated hardware cache coherency mechanism, a hardware-based cache coherence mechanism has its own drawbacks, as explained in Chapter 1. Without such a mechanism, the programmer is responsible for maintaining cache coherency. While it is doable for a small embedded application, but with any nontrivial program, writing a correctly executing program for a non-cache-coherent machine by hand is close to impossible.

3.2 Stale Data Problem

Stale data problem is a state in which an old value of shared data exists in one core cache. Assuming the MESI protocol[29] is supported by the cache, whenever a particular processor core writes to a cached data, the state of the data should become "Modified". Then, the cache coherence mechanism is responsible for notifying other cores with the same piece of data in their cache to mark that particular data as "Invalid". In the absence of such a mechanism, stale data reference should be avoided.

Figure 3-1 is a trivial example of a stale data problem. Assume there are three global integers in the main memory, namely, `a`, `b`, and `c` stored in a single memory line. All those three variables are shared by both core 0 and core 1. Core 1 reads the value of `a` then write it into `b`. As core 1 reads the value of `a`, the entire cache line is copied to core 1's cache. As variable `a` is 0, after the operation `b` becomes 0 too. After that, for example, core 0 writes to variable `a`, so the entire cache line is copied to core 0's cache, and the value of `a` is updated to 20. At this point, in a cache-coherent system, a hardware-based mechanism notifies core 1 that its copy of the same memory line is now invalid. Without such a mechanism, core 1 is unaware of the change made by core 0. Whenever core 1 executes the last statement `c = a;` then the old values of `a` is used causing the result to be incorrect. Therefore, the compiler should provide a mechanism to prevent stale data from continuing to exist

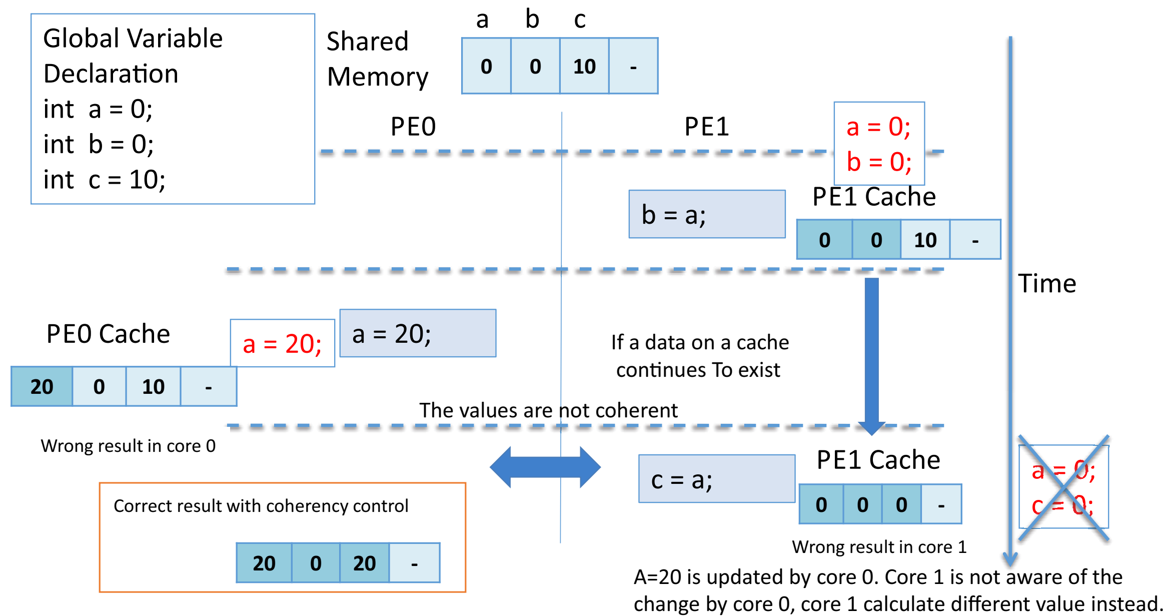


Figure 3-1: The stale data problem. Image from [20]

on other processors core.

3.2.1 Stale Data Handling

As explained in chapter two, the compiler decomposes the program into MTs, and then it analyzes the data dependency of each MT. Then it statically schedules those MTs according to their EEC on multiple cores. This way, the compiler keeps track of which part of the memory location that each MT reads and writes. In order to prevent stale data, first, the compiler tries to schedule all MTs which share the same data on the same core. Nevertheless, most of the time, data sharing between cores is unavoidable. After an MT makes a change to a shared variable, the compiler then inserts a writeback command followed by a synchronization mechanism on the writer core before continuing to the next MT. As the MTs are statically scheduled, the compiler keeps track of which cores that still have the data in its cache, and then it inserts self-invalidate instruction on those cores before starting a task that will consume the modified data.

Similar to the MESI protocol, the compiler manages the four states of MESI protocol. Figure 3-3 shows the compiler controlled coherency protocol. The red arrows show compiler induced action. The compiler inserts “writeback” and “self-

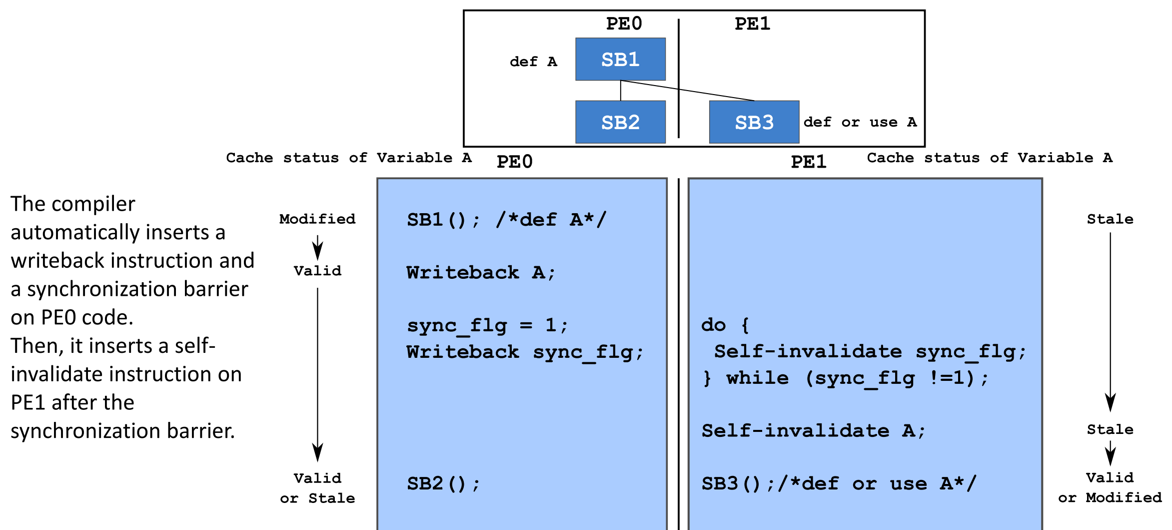


Figure 3-2: Cache control code inserted by the compiler to prevent reference to stale data. Image from [20]

invalidate" command to the cache to maintain the appropriate state for each cache line. The blue line is an automatic response from the caching mechanism.

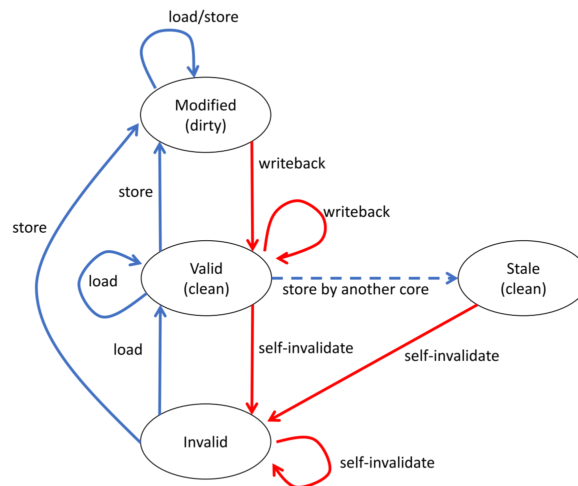


Figure 3-3: Coherency Protocol Managed by the Compiler

Figure 3-2 is an example of the code inserted by the compiler. A task SB1 running in core 0 modifies the value of `A`. At the end of the MT, the compiler inserts a writeback instruction for variable `A` to make sure the changes are written to the main memory. The compiler then updates a flag, which indicates that a writeback for variable A has been done. It again issues a writeback after updating the flag. Meanwhile, on core 1, a task SB3 uses variable `A`. In order to prevent core 1 from using a stale copy of variable `A`, the compiler inserts a busy loop synchronization point waiting for the flag value to be updated by core 0. If the flag is set, then the local copy of variable `A` is self-invalidated, effectively forcing core 1 to fetch the updated value of `A` from the main memory. Finally, task SB3 may run and use variable `A`. Notice that in the busy loop, for each iteration, core 1 also self-invalidates its copy of the flag.

The compiler also priorities the schedule to minimize the synchronization delay. Likewise, if multiple cores have the same copy of data at the same time, tasks working with those copies will not be scheduled at the same time because there is output dependency among these tasks, effectively preventing the particular cache line from being simultaneously updated. In other words, it prevents multiple processors from entering the "Modified" state at the same time, mimicking the behavior of a MESI coherence protocol. The compiler also forbids load/store operation during the "Stale"

state before the stale data is self-invalidated.

The compiler inserts this stale data prevention mechanism for both Read-after-Write and Write-after-Write type data dependency. Meanwhile, for the case of Write-after-Read data dependency, the compiler only inserts synchronization instruction. This way, the OSCAR Compiler guarantees the memory consistency at the end of each macro task, while also preventing stale data problem.

The proposed approach is more efficient compared to the snooping protocol, as there is no broadcast of the cache update packet flooding the shared bus.

3.2.2 Selective Cache Operation in Loop Parallelization

OSCAR Compiler supports several kinds of loop parallelizing techniques. In DOALL loops and reduction loops, which do not have any loop carried dependency, typically stale data is not a problem. However, with DOACROSS loops, it inherently has a loop-carried dependency in each of its iterations[34].

In an NCC architecture, while inserting the stale data problem handling mechanism described in the previous subsection solves the problem, but it may lead to an excessive number of synchronization. The cache invalidation instruction needs to be inserted on every iteration, effectively disabling the cache function. In order to reduce the impact of this problem, the compiler only inserts the stale data handling mechanism if the array element accessed by parallelized iteration shares the same cache line[24].

3.3 False Sharing Problem

False sharing is commonly overlooked in software-based cache coherence. Meanwhile, it is one of the most important factors in the performance of NCC architecture. Typically cache replacement mechanism works with line-level granularity. It cannot updates or replace an individual byte or word. False sharing happens when multiple cores share independent data stored in the same memory line. Whenever one of those data is updated, cache inconsistency may occur. Some of the previous approach [8][11], treated false sharing in an array as a true sharing that requires invalidating

the entire array. In this approach, several simple data restructuring methods are utilized to avoid false sharing without a significant performance impact.

Figure 3-5 is a trivial example of a false sharing problem. Let us assume there are two independent global integers stored in main memory, namely, `a` and `b`. As their size is smaller than the memory line, they are stored next to each other, sharing a single line. Then core 0 updates the value of `a`. This action results in the whole memory line copied into core 0's cache and the value of `a` in core 0's cache updated to 10. At the same time, core 1 updates the value of `b`, which again causes the entire memory line containing both `a` and `b` copied to core 1's cache and core 1 change the value of `b` in its cache. Core 0 only updates variable `a`, and core 1 only updates variable `b`; however, core 0 is not aware of the change made by core 1 on `b` and vice versa. Whenever both cache lines are replaced, inconsistency may occur. The final result in the main memory depends on the sequence of the line replacement.

If the arrays are not aligned to the cache line, false sharing will occur in all parts of the array. With all the beginning of each array is aligned to the boundary of the cache line, the compiler must also detect whether false sharing occurs inside of the array itself. False sharing can be detected after the compiler performs the memory access range analysis of each task in the MTG after the task division for parallel execution, especially for loops subjected to loop-division for parallel executions.

To understand the detection process, consider a sample program with several loops depicted in Figure 3-4 (a) and its MTG in Figure 3-4 (b). A single line indicates data dependence, and a double line indicates a portion that is likely to cause false sharing. The simple example program consists of four tasks: three DOALL loops and one sequential loop. In order to parallelize the program, each task is subjected to a loop division. Figure 3-4 (b) shows an MTG before the loop division. The loop is then divided into several partial tasks in which each of the tasks is executed in different processor cores, as shown in Figure 3-4 (c). Then, the compiler will analyze the access range based on the array and its index used by each partial task.

If the memory access range of each task does not overlap, and no cache line crosses the boundary of those access ranges, it can be guaranteed that false sharing will not occur. Meanwhile, false sharing is likely to occur if the lowest dimension of the array

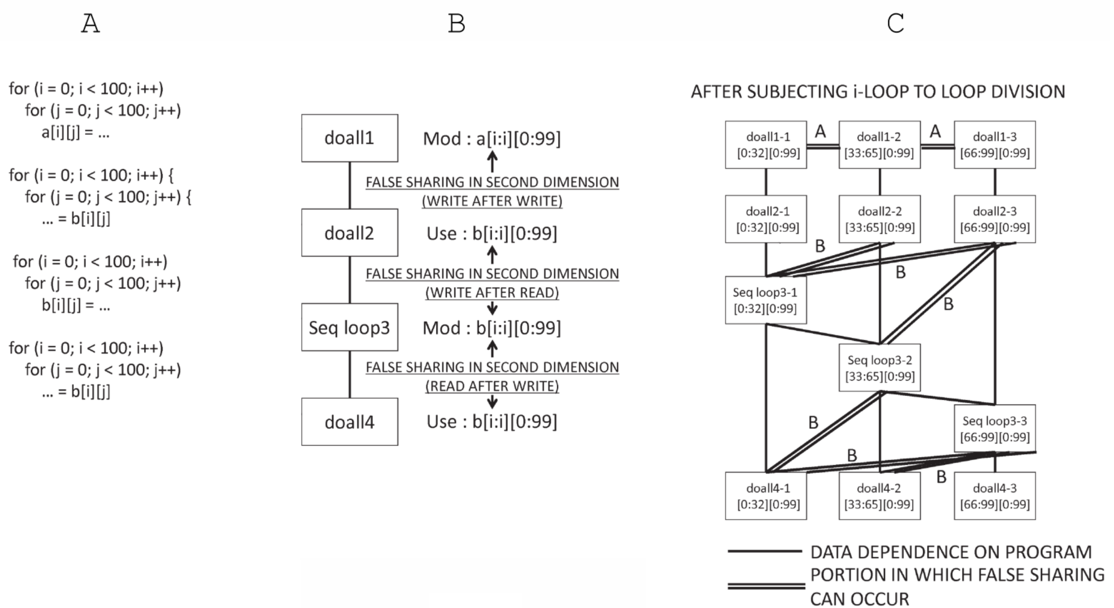


Figure 3-4: False Sharing Detection Process. Image from [21]

used by the above-mentioned partial task could not be fully divided by the cache line size. In that case, false sharing is likely to occur between partial tasks that write into the same array (write-after-write) or between different partial tasks that write into and other tasks that read from the same array (read-after-write). One possibility of false sharing can be observed in the DOALL1 loop. For the purpose of analyzing the possibility of false sharing, which occurred by dividing the loop in arbitrary size, the task is divided into the maximum possible number, which is the number of the loop iteration with one processor core handles exactly one iteration. Dividing the DOALL1 loop at one iteration per processor may cause Write-after-Write false sharing. The data written by a one-iteration partial task will likely take less space than the cache line size; therefore, more than one processors may write into the same memory address at the same time; hence, false sharing occurs. This is also reflected in the example. When the compiler analyzes the memory access range of DOALL1-1 and DOALL1-2 and DOALL1-3, the same cache line may be updated by multiple processor cores in each task at that time. False sharing is detected within the DOALL1 loop.

The other possibility of false sharing may occur between different tasks. When the compiler analyzes the memory access range of DOALL2-2 and LOOP3-1, the same cache line may be updated by multiple processor cores in each task at the same time; hence, false sharing occurs in this part. LOOP3-1 also updates the cache line read by DOALL2-2, so, an inverse dependency or Write-after-Read dependency due to false sharing occurs between LOOP3-1 and DOALL2-2. Furthermore, the cache line read by DOALL4-1 is updated by LOOP3-2, so, between LOOP3-2 and DOALL4-1, so, a flow dependency (Read after Write) due to false sharing also occurs between LOOP3-2 and DOALL4-1.

The compiler must apply data restructuring techniques to prevent false sharing. A simple variable alignment works well for most cases but is inefficient for specific cases. Several different data restructuring methods are required to prevent false sharing problem for each use case. If all effort failed, then the array will be processed sequentially.

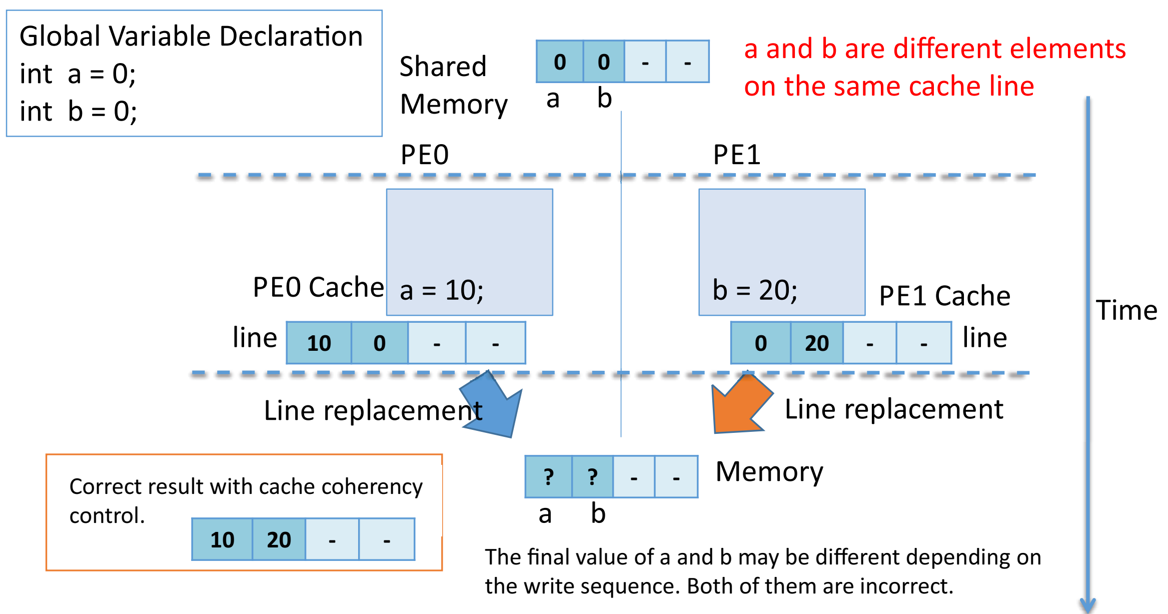


Figure 3-5: The false sharing problem. Image from [20]

3.3.1 Variable Alignment and Array Expansion

Aligning each variable to the beginning of each cache line prevents false sharing effectively. The same method works for small one-dimensional arrays with the number of elements less than the number of cache lines in all available processor cores. The array is expanded, so there is only one element stored in every cache line. For larger arrays, however, this method is not practical due to cache space wasting. This method is also the default behavior if the compiler cannot determine the access pattern of the array.

3.3.2 Cache Aligned Loop Decomposition

Oscar Compiler, in addition to the coarse-grain parallelization, also employs loop parallelization. One of the conventional methods is loop decomposition. Loop decomposition splits a loop in a task to several smaller tasks run on different cores. Usually, the compiler splits the loops equally by the number of available cores. However, to prevent multiple cores from sharing a cache line, the compiler splits the loop along the cache line, as seen in Figure 3-6(A).

3.3.3 Array Padding

It is not always possible to split a two-dimensional array cleanly along the cache line, especially if the innermost dimension is not an integer multiple of the cache line size. In this case, the compiler pads the innermost array to match the size of the cache line. This method is illustrated in Figure 3-6(B). This method also potentially waste some cache space.

3.3.4 Data Transfer Using Non-cacheable Buffer

Sometimes, due to the structure of the program, it is not always possible to apply any of the previous methods without a significant performance penalty. For example, in extreme cases, the cache aligned decomposition might introduce a significant imbalance, or the array padding wastes too much cache space, a non-cacheable buffer can

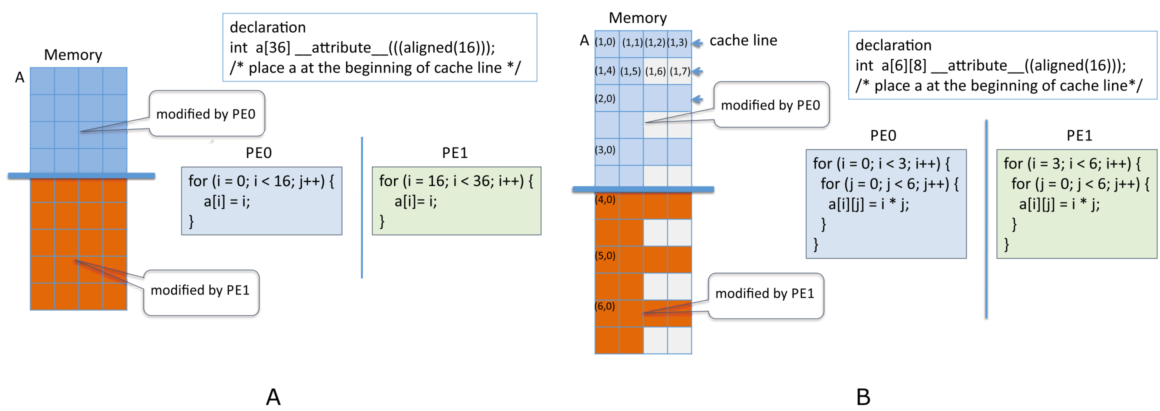


Figure 3-6: (A)Cache aligned loop decomposition is applied to a one-dimension matrix to avoid false sharing. (B)Array padding is applied to a two-dimensional matrix to avoid false sharing. Images from [20]

be used. Non-cacheable buffer is a small memory window in which the cache is bypassed. The buffer is used for cache writes along the border between area modified by different processor cores. Instead of the conflicting core writes directly to the shared line, the data are stored in the non-cacheable buffer, and then the cache line owner copies the data from the buffer to the appropriate location. Figure 3-7 describes the usage of the non-cacheable buffer.

3.4 Software Cache Coherent Control by OSCAR Compiler

In order to support NCC architecture on OSCAR Compiler, a new compiler module is created. The new NCC module leverages the OSCAR Compiler framework to handle both stale data and false sharing problems. Figure 3-8 depicts the proposed compilation process. The grayed boxes are new steps introduced to handle cache coherency.

As explained in Chapter 2, the OSCAR Compiler starts the compilation process by hierarchically decomposing the program into coarse-grain tasks. Then the compiler analyzes the control flow and data dependency between tasks. Based on these data, the compiler then determines the EEC for all tasks, generates the coarse-grain schedule, and automatically creates parallel sections. At this time, the compiler has enough information on which task run on which core and their def-use pattern. Moreover, it also collects an array access pattern and employs a pointer analysis to supplement the def-use pattern from the coarse-grain task scheduling. The new compiler module utilizes these data to generate cache control instructions for avoiding stale data problem and restructure the array to avoid false sharing.

In order to take care of the stale data problem, as explained in Section 3.2.1, the new compiler modules inserts required data transfer and synchronization in addition to self-invalidate instructions in between the tasks.

Meanwhile, in order to handle false sharing, the NCC module marks variables or arrays shared between two or more parallel sections at the same time. A simple

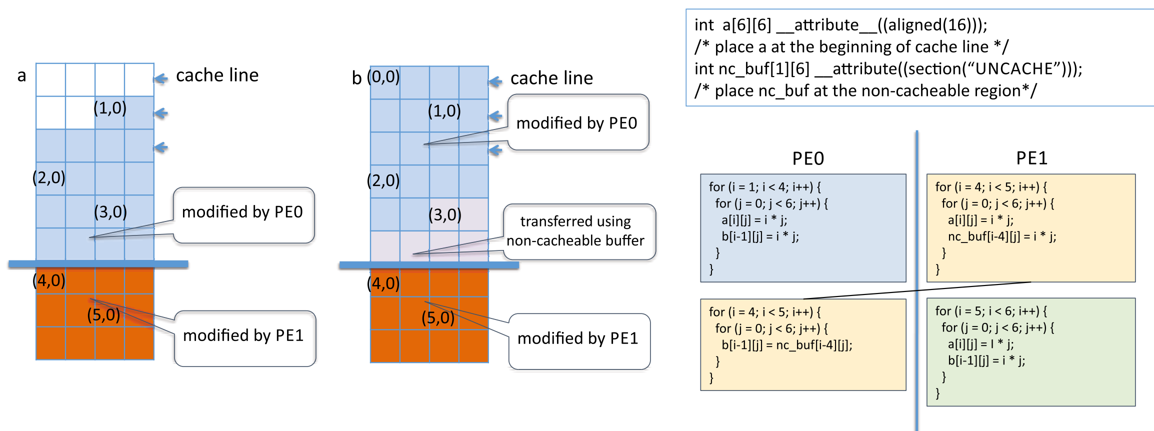


Figure 3-7: Non-cacheable buffer is used to avoid false sharing. Image from [20]

decision tree, which described in Figure 3-9, decides the compiler actions depending on the kind of those shared arrays. Several different approaches are taken to avoid false sharing: Scalar variables are aligned to the cache line. Small 1-dimensional arrays are aligned and expanded. The decomposition of the loop considers the size of the array to prevent false sharing. For two-dimensional arrays, depending on its innermost dimension, it is padded to fit into the cache line. If all effort fails, non-cacheable buffers may be used. After these data restructuring methods are taken, the compiler continues the normal compilation process.

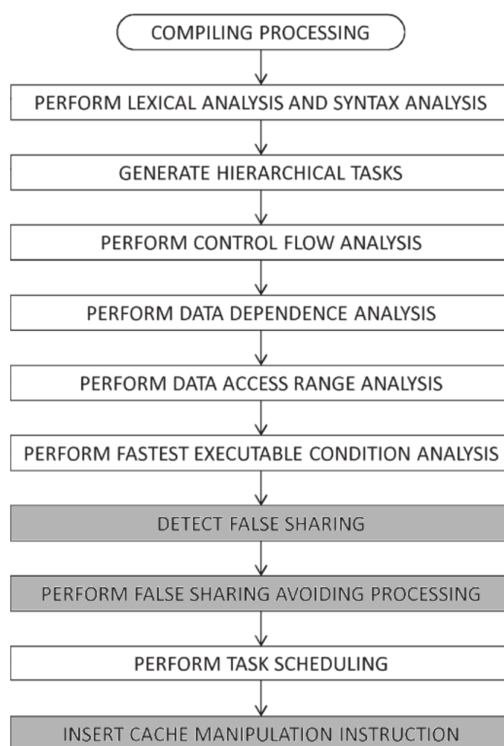


Figure 3-8: Proposed compilation sequence. Image from [21]

As the compilation process completed, the OSCAR API translates the NCC directives into the proper driver/run-time API call or appropriate in-line assembly instructions. Since the API implementation differs from platform to platform, it will be discussed more in the next chapter.

```
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
```

Figure 3-9: Pseudocode for detecting and mitigating stale data and false sharing. Image from [1]

Chapter 4

Development of Non-Cache Coherent Architecture Evaluation Platforms

This research aims to provide automatic parallelization for the NCC multicore system using the OSCAR Compiler. Therefore, the system should be at least targetable by the OSCAR API. Currently, most multicore processor systems are targetable by OSCAR API. However, there are some requirements imposed by the API and also the NCC extension of the OSCAR Compiler. The target platform is illustrated in Figure 4-1.

In order to use the NCC support on the OSCAR Compiler, the multicore system must have:

- Shared memory accessible by all cores.
- Private cache with each cache line has at least two flags, "Valid" and "Dirty".
- Software-controllable cache operation for at least "invalidate" and "writeback".
- Atomic store instruction or "fence" like instruction for architecture with weaker memory consistency.

The OSCAR API mandates the multicore system to have at least a shared memory accessible by all cores, which is just like an ordinary multicore processor available on the market. Furthermore, not only limited to off-the-shelf multicore processors, but the API also supports manycore processors, soft processor cores on FPGA, or industrial real-time multicore microcontroller without cache coherency mechanism. The 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.

The second requirement is a private cache with at least two flags. The NCC module does not require a full MESI protocol support, but a two-state cache line is sufficient as the compiler is responsible for maintaining the cache state. Tag support is also optional at the cost of lower performance due to the necessity of flushing the entire cache instead of line-by-line flush/writeback.

The next requirement is software controllable cache operation. This requirement is fundamental as the compiler has to control the cache by software. The NCC module at least needs two instructions, "Flush" or "Writeback", and "Self-Invalidate". Most

architectures support this kind of cache operation instructions. RISC-V is one of a notable architecture which mandates for a software transparent cache architecture. In other words, the cache is not software-controllable. This limitation means the vanilla version of the RISC-V processor cannot be used with the OSCAR Compiler. Required hardware modification on RISC-V will be discussed later in this chapter.

The last requirement is atomic write or fence like instruction to preserve the order of write to the memory. On architecture with a firmer memory consistency model, this is not an issue, but other architectures with weaker memory consistency need to those instructions to ensure the synchronization flag is not set before the data it guards are fully written. This is important because some out-of-order RISC-based soft processor cores are designed only for a single-core operation and do not support atomic write instructions.

In order to evaluate the performance of the proposed method, four different platforms are used, i.e., Renesas RP2, Nios II SoC, RISC-V SoC, and Intel Xeon. The following sections detail each of the evaluation platforms.

4.1 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 snooping-based hardware cache coherency. It was jointly developed by Renesas Electronics, Hitachi Ltd., Waseda University, and supported by METI/NEDO Multicore Processors for Real-time Consumer Electronics Project in 2007[17].

Each processor core has a private cache with hardware coherence support for up to four cores, as seen in Figure 4-2. However, there is no hardware coherence controller between the clusters. A software-based cache coherency must be employed to utilize more than four cores across the cluster. For hard-real-time applications, the MESI hardware coherence mechanism can be disabled completely. The RP2 board, as configured, 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 applications, was not used in this evaluation.

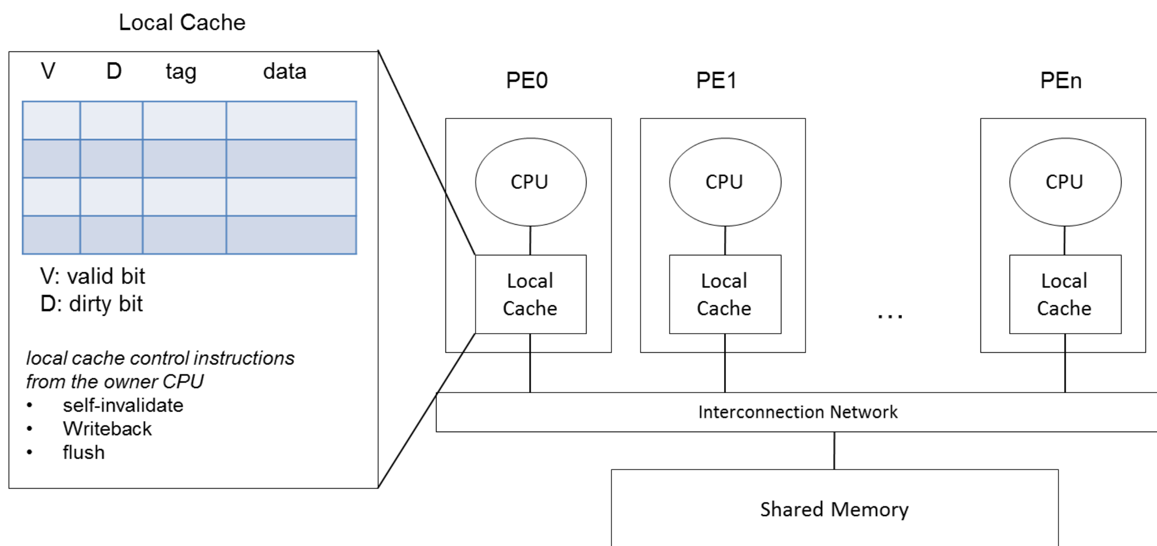


Figure 4-1: Target Architecture. Image from [20]

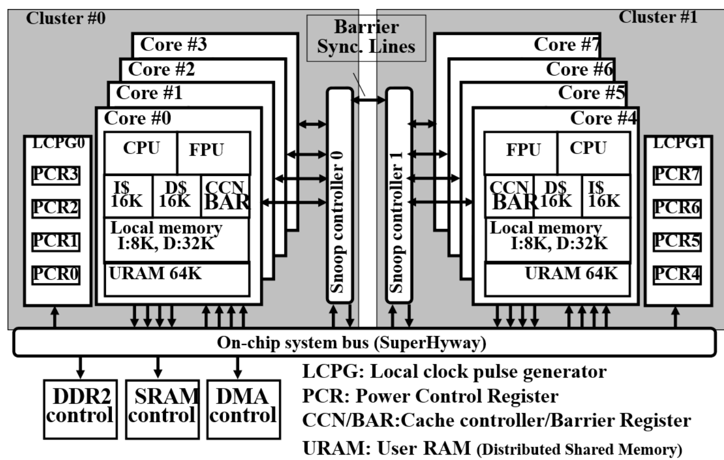
The RP2 processor supports several native instructions in NCC mode: write-back operation (`OCBWB` instruction), cache invalidate (`OCBBI` instruction), cache flush (`OCBP` instruction). The OSCAR API directly translates the NCC directives to these instructions.

4.2 Eight-Core Nios II System on FPGA

A custom multicore system based on Altera Nios II soft CPU[16] core was created for testing the scalability of the proposed method.

The Altera (now Intel) Nios II processor is a 32-bit embedded processor architecture for Altera FPGA. It is has a RISC based design. In this research, the Nios II/f core is chosen. This variant has six pipeline stages, hardware-based single-cycle multiplier and divider, dynamic branch prediction, and a custom instruction port, which is used to connect an external FPU module. The Nios processor does not come with hardware-based cache coherence support. However, it has several instructions for cache manipulation, providing support for software-based cache coherence.

Figure 4-3 shows a block diagram of the multicore system. The system consists of up to eight Nios II CPUs with 16kB data cache for each core. The caches are configured as a direct-mapped, 32 bytes wide, writeback cache. All eight cores share two banks of 1 GB DDR4 1066 MHz main memory connected over an Altera External Memory Controller through the Arria 10 Hard Processor System (HPS) memory window. The system is designed in regards to OSCAR Architecture. The system is configured as 8 CPU modules. Each CPU module has its own timer, JTAG, and an on-chip memory acting as both DSM for synchronization flag, and LDM for private data. The CPU modules are connected to the main memory and peripherals through a series of Avalon bus, adapter, clock bridge, and Altera Platform Designer generated interconnect. Each of the CPU module also has its own boot memory to run the bootloader and Altera HAL, which provides necessary C library support. Meanwhile, the benchmark programs and data were loaded and run from the main shared memory. An additional external reset sequencer is added to control the booting process of the system. The system has no hardware-based cache coherence mechanism. The system



"**A 4320MIPS Four-Processor Core SMP/AMP with Individually Managed Clock Frequency for Low Power Consumption**", Proc. of 2007 **IEEE International Solid-State Circuits Conference**(ISSCC2007), Feb. 2007.

Figure 4-2: Renesas RP2 8-core Embedded Multicore Processor. Image from [38]

is designed and synthesized entirely on Altera Quartus II 18.1 and implemented on Altera Arria 10 SoC Board Development Kit.

The Nios II CPU has several native instructions to manage its data cache manually; cache flush (`flushd` instruction) and cache invalidate (`initd` instruction). The Altera HAL API[16] wraps some of the cache management functionality into several easy-to-use functions. The OSCAR API directly translates the directives into the appropriate Altera HAL API call.

The Nios II platform does not have native parallel execution support. Therefore, the OSCAR API Translator must convert the OpenMP parallel sections into separate code for each Nios II core to run. As explained in Chapter 2, OSCAR Compiler generates an OpenMP-like single-level parallel thread execution. On this multicore implementation, each Nios II core is assigned a unique CPUID. Instead of generating OpenMP Parallel sections, the API Translator generates a giant switch case wrapping each core's code with the CPUID of each processor as the selector. This way, all CPU cores can share a reset vector while executing their respective sections. The API Translator then assigns the memory segments for each thread private variable, synchronization flags, and shared data to appropriate segments. It also aligns the data according to the cache line size.

4.3 Quadcore RISC-V Based Non-Cache Coherent SoC

RISC-V[37] is the only open Instruction Set Architecture (ISA), which was first introduced in 2010 and is currently maintained by the RISC-V Foundation. The foundation members come from both academic institutions and industries, including software, systems, semiconductor, and IP. The ISA, especially the base instructions and approved extensions, has been frozen since 2017. Being an open ISA, everyone can optimize its design for power consumption, performance, security, and other specific requirements. It also supports custom instructions for applications that require particular acceleration or specialty functions. The RISC-V ISA has a BSD

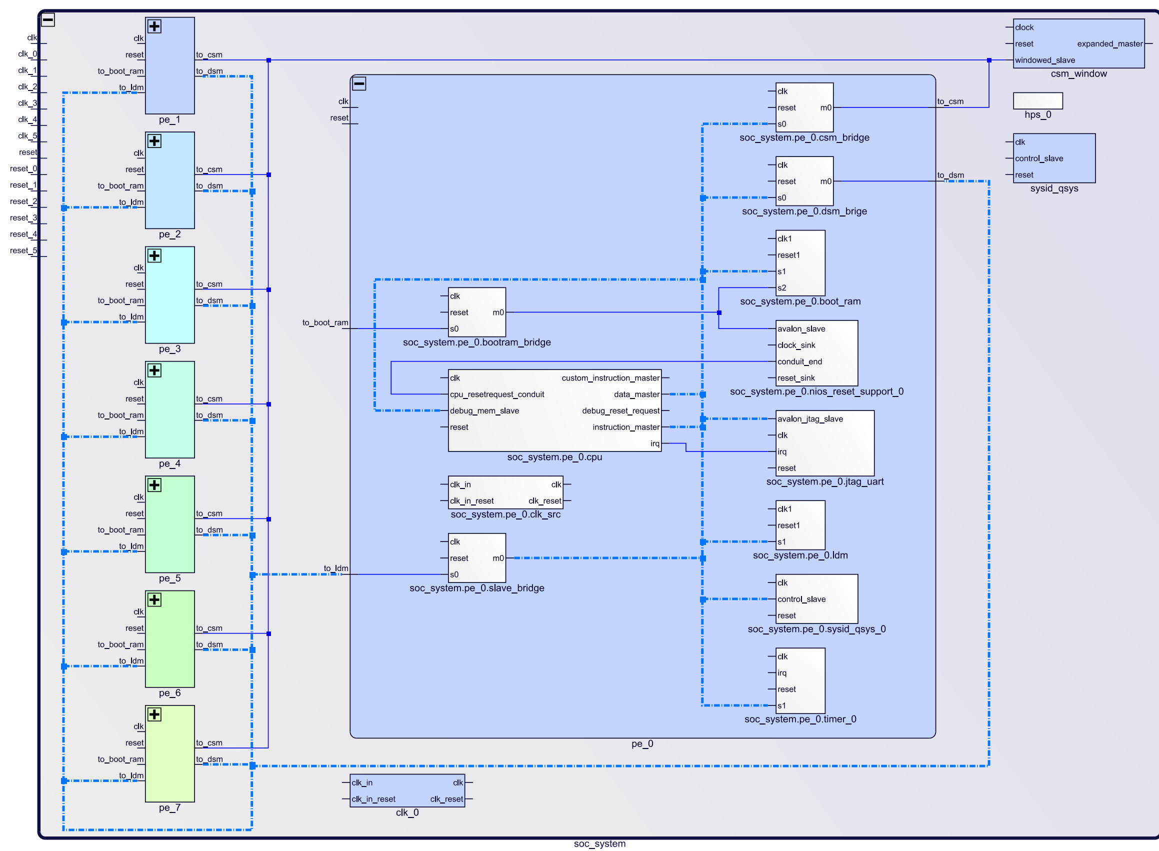


Figure 4-3: Diagram of the 8-core Nios II SoC. Image from [24]

license[6]; therefore, now there are several implementations of the RISC-V processor, both opensource and commercial licensed.

A RISC-V multicore, which is based on VexRISCV[35] by SpinalHDL, is developed specifically for this research. The implementation is written in Scala, and it uses a plugin system for writing CPU extension, which makes writing CPU extensions relatively easier. SpinalHDL framework compiles Scala code to either VHDL or Verilog. The generated HDL is also FPGA friendly and does not use any vendor-specific primitives. Also, as the target FPGA is an Altera's Arria 10 FPGA, it is very convenient that SpinalHDL can generate Tcl scripts for easy integration with Altera Platform Designer.

The first iteration of SoC used VexRISC Briey SoC reference design as the base model. It used RV32IM core without floating-point support. VexRISCV is a five-pipeline-staged single-issue in-order CPU. The CPU core itself was later changed to a more capable CPU core with floating-point support. The data cache subsystem is a custom design, based on VexRISCV DBusCachedPlugin configured as a single level 4 way 16kb private cache for each core. Each processor core is organized as a processor module consists of an RV32IM core, an on-chip boot memory, a timer, and a series of bridges for external AvalonMM interfaces. Each processor module has access to the external memory controller and a shared UART. A NiosII-based subsystem is used to help with the boot process and IO as the JTAG and UART drivers were not working correctly. The overall arrangement of the multicore is very similar to the Nios II multicore explained in Section 4.2.

The latest RISC-V ISA specification, version 2.1, was revised in December 2019[37]. Unfortunately, as the document states, cache management instructions are not part of standard instructions but may be expanded in the future revision. Therefore, a mechanism for cache management is required. For this research purpose, a custom cache controller plugin is developed based on VexRISCV's DBusCachedPlugin plugins. The plugin has an interface that can be controlled from user-level CSRs using CSRRW/CSRRWI instruction, as seen in Table 4.1. The CSR interface is based on the VexRISCV CSR module. The flush and invalidate operation can be initiated for a single memory pointer or the whole cache area. The memory argument sent to

Table 4.1: Custom CSR for Interacting with the Cache Subsystem

Address	**Format**	**Description**
0x800	32 bit address	writeback/flush an address
0x801	000000000000000000000000000000ib	flush status/read only i = 1 if request received b = 1 if busy/not done
0x802	0000000000000000000000000000000f	set f bit for flushing the whole cache
0x810	32 bit address	invalidate an address
0x811	000000000000000000000000000000ib	invalidate status/read only i = 1 if request received b = 1 if busy/not done
0x812	0000000000000000000000000000000f	set f bit for invalidating the whole cache

the CSR must be cache aligned address; otherwise, the cache controller may fail to determine the exact cache line number. The typical delay for a writeback and invalidate delay is about 13 cycles more than a regular cache miss in both directions. A simple driver is implemented to wrap the CSR write and a busy loop to check if the operation is accomplished. The OSCAR API Translator converts the NCC directives to this driver call.

RISC-V specification mandates weak memory ordering, but the current implementation of VexRISCV is strictly in-order; hence no memory fence or atomic operations are necessary. Also, all memory access and writes generated by OSCAR compiler with NCC option are always aligned, effectively preventing the VexRISCV write back module from generating multiple writes for miss-aligned variables.

The VexRISCV platform also does not have native parallel execution support. Similar to the Nios II platform, the OSCAR API Translator must convert the OpenMP parallel sections into a suitable format. Similar to other implementation platforms, the OSCAR Compiler generates a specific code for each core. Each of these codes is then compiled with the RISC-V toolchain. Then a custom phyton script is used to prepare the generated hex files and merge them into a single memory image file with respect to the start vector of each core. Due to incomplete JTAG driver implementation, a Nios-II-based subsystem helps the bootstrapping process by copying

the generated code to the main memory and starting all the VexRISCV cores. This subsystem also handles the serial input-output of the multicore system.

Similar to the Nios II platform, the API Translator also assigns the memory segments for each thread private variable, synchronization flags, and shared data to appropriate segments. It also aligns the data accordingly.

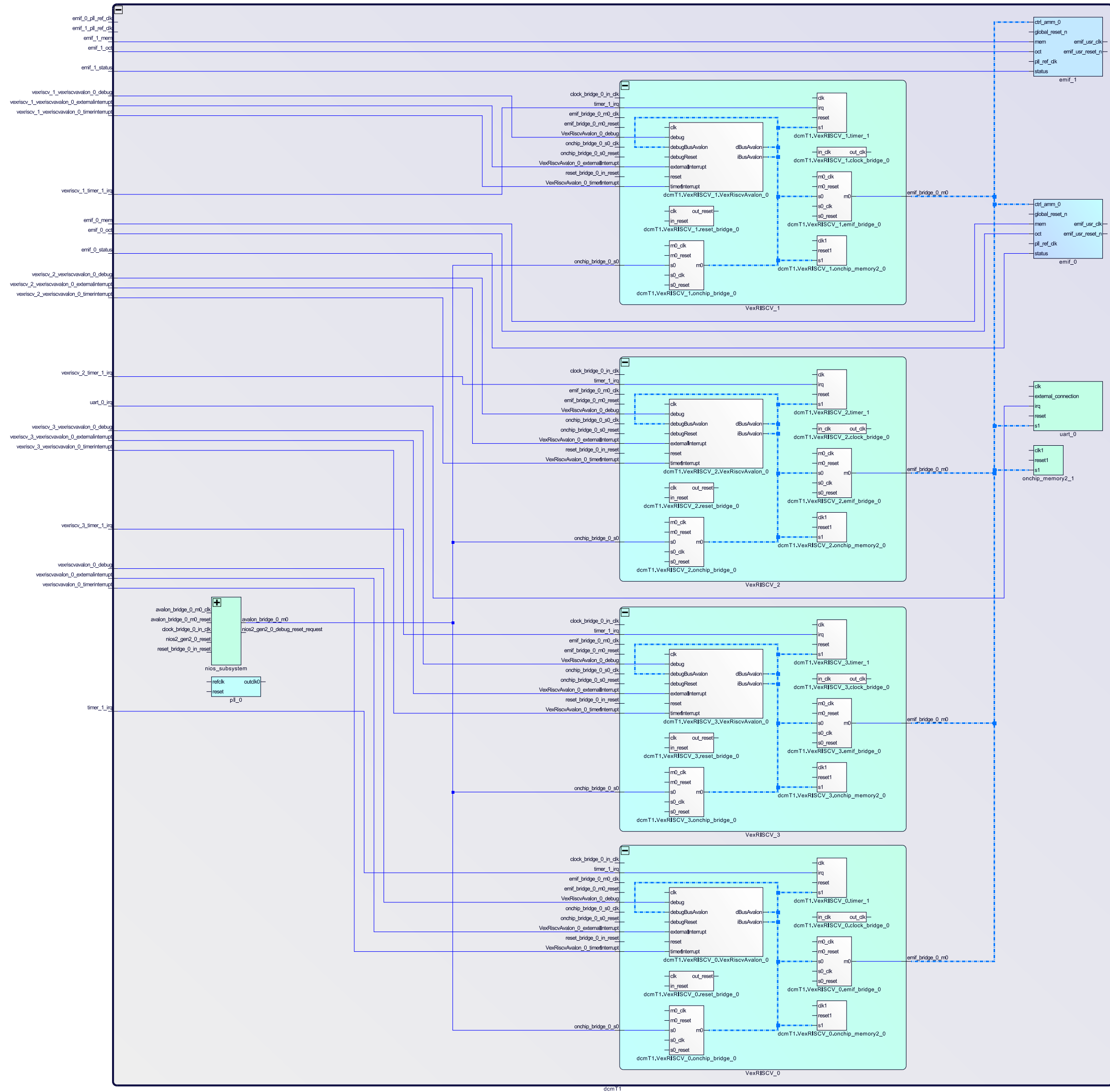


Figure 4-4: Diagram of the 4-core RISV-V SoC showing only the data connections

Chapter 5

Performance Evaluation

Table 5.1: List of Benchmark Programs

Benchmarks Programs			
SPEC2000	equake	NAS Parallel Benchmark	cg
	art		mg
SPEC2006	lbm		bt
	hmmer		lu
MediaBenchII	MPEG2 Encoder		sp

The hardware-based cache coherence scheme has been considered to have a better performance than a software-based approach. In this chapter, the performance of the compiler controlled cache coherency scheme is tested and compared against the hardware-based mechanism. Most of the previous acts evaluated their proposed software coherence scheme on simulator [8][11][36]. In this research, however, real benchmark applications are tested on practical hardware implementation.

5.1 Benchmark Applications

There are several factors in deciding the benchmark programs to be used here. Unfortunately, the selection of the benchmark program is somewhat limited due to several limitations. One of the main factors is the absence of the Operating System (OS) support. Most OSes in the market are designed for the Symmetric Multi-Processing (SMP) shared memory system. Therefore, most OSes cannot run correctly on all of our evaluation platforms, and as a result, all benchmarks are run bare-metal without any operating system. The lack of OS also complicates the usage of file systems, which limits further the benchmark selection.

Ten benchmark programs from SPEC2000, SPEC2006, NAS Parallel Benchmark (NPB), and Mediabench II were run to test the performance of the proposed method on the Renesas RP2 platform. And, due to the lack of file system support on Nios II and RISC-V SoC implementation, both SPEC benchmarks and Mediabench cannot run correctly. Therefore, only NPB successfully on all four architectures.

All benchmark programs are written in C. In order to allow for a more efficient parallelization, some SPEC benchmark programs, namely “lbm” and “hmmer”, and MediaBench MPEG II Encoder were converted to Parallelizable C[27] by hand. Par-

allelizable C is a guideline to write a C program that allows a parallelizing compiler to extract the full potential of parallelization and data locality optimization, mainly targeting arrays and loops. It is similar to MISRA-C[3], which is commonly used in the embedded system. Conversion to Parallelizable C is trivial and straight forward. As an example, the “lbm” benchmark requires only three lines of pointer related modification, and the rest of the code can be parallelized automatically. Minor changes are also necessary for “art”. Aside from those, most other benchmark program does not require any changes.

The converted benchmark programs were then compiled using OSCAR Compiler and processed by the OSCAR API translator. The parallelized programs are then fed to the backend compiler for each platform. For the RP2 platform, Renesas SuperH C Compiler (SH C) is used. For the Nios II SoC, the benchmark programs are compiled with nios2-elf-gcc v7.3.1 against Altera HAL 18.1. For the RISC-V platform, riscv-gcc v8.3 included in the RISC-V toolchain is used. Meanwhile, for the Intel platform, gcc v7.4 is used.

The SPEC benchmark programs were run in their default configuration and datasets except for 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 off-chip main shared memory size, which is only 128 MB.

5.2 Experimental Results and Analysis

As the benchmark programs are successfully compiled, the performance data on each platform is measured. Figure 5-1, 5-2, and 5-3 show the speedups by multiple cores of the proposed method on RP2 Processor, Nios II, and RISC-V multicore system.

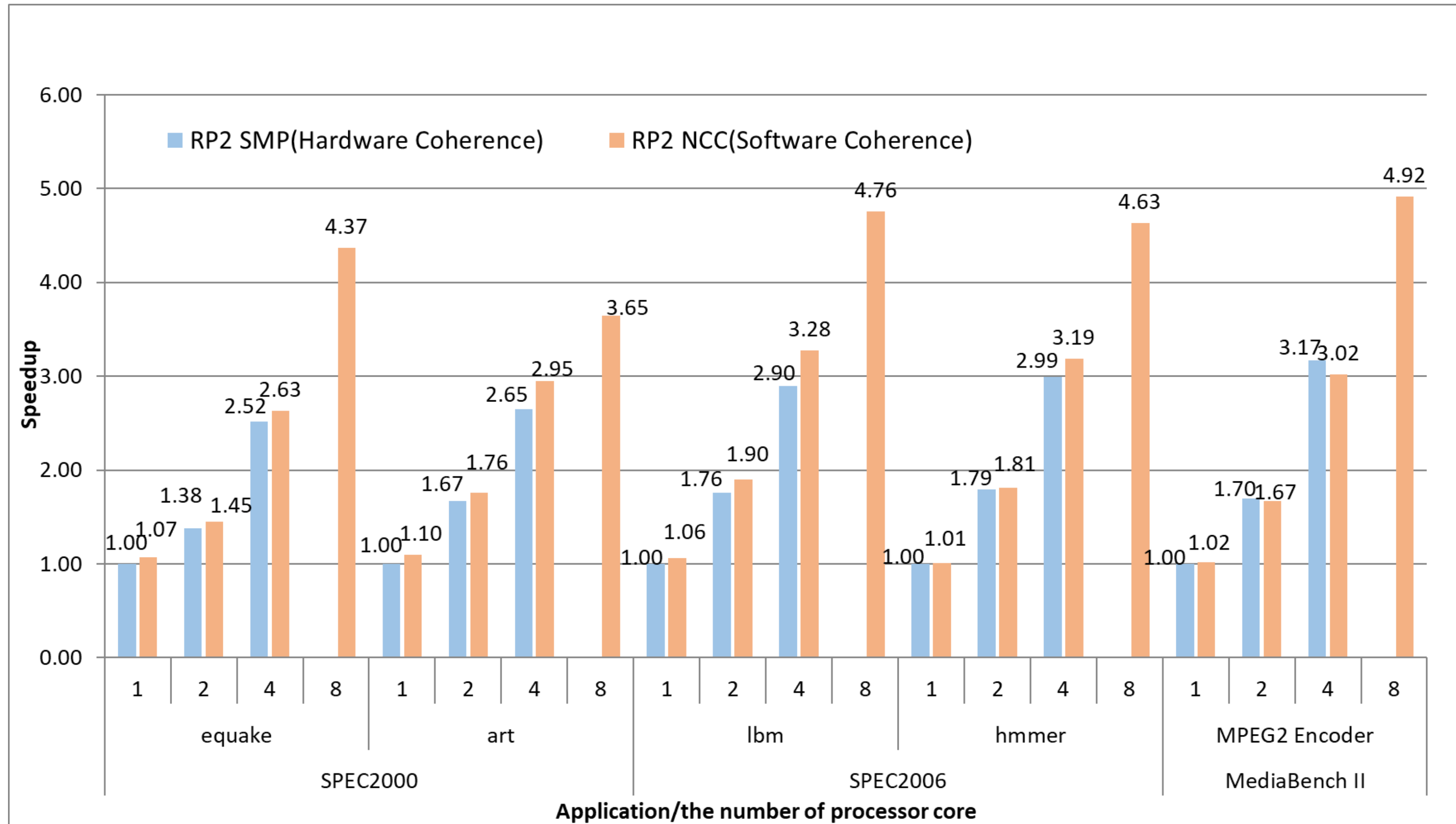


Figure 5-1: The performance of the proposed method on RP2 Processor for SPEC Benchmark and MediaBench. Image from [1]

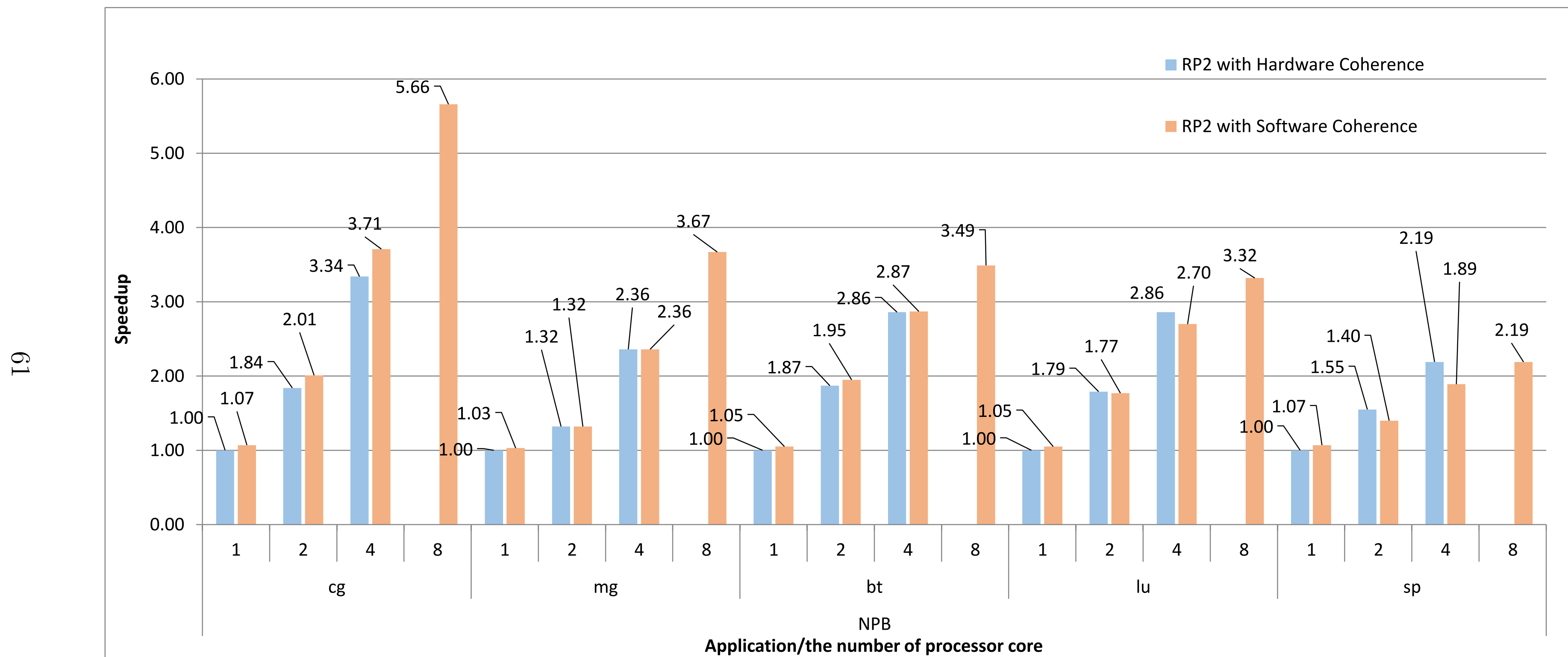


Figure 5-2: The performance of the proposed method on RP2 Processor system for NAS Benchmark. Image from [1]

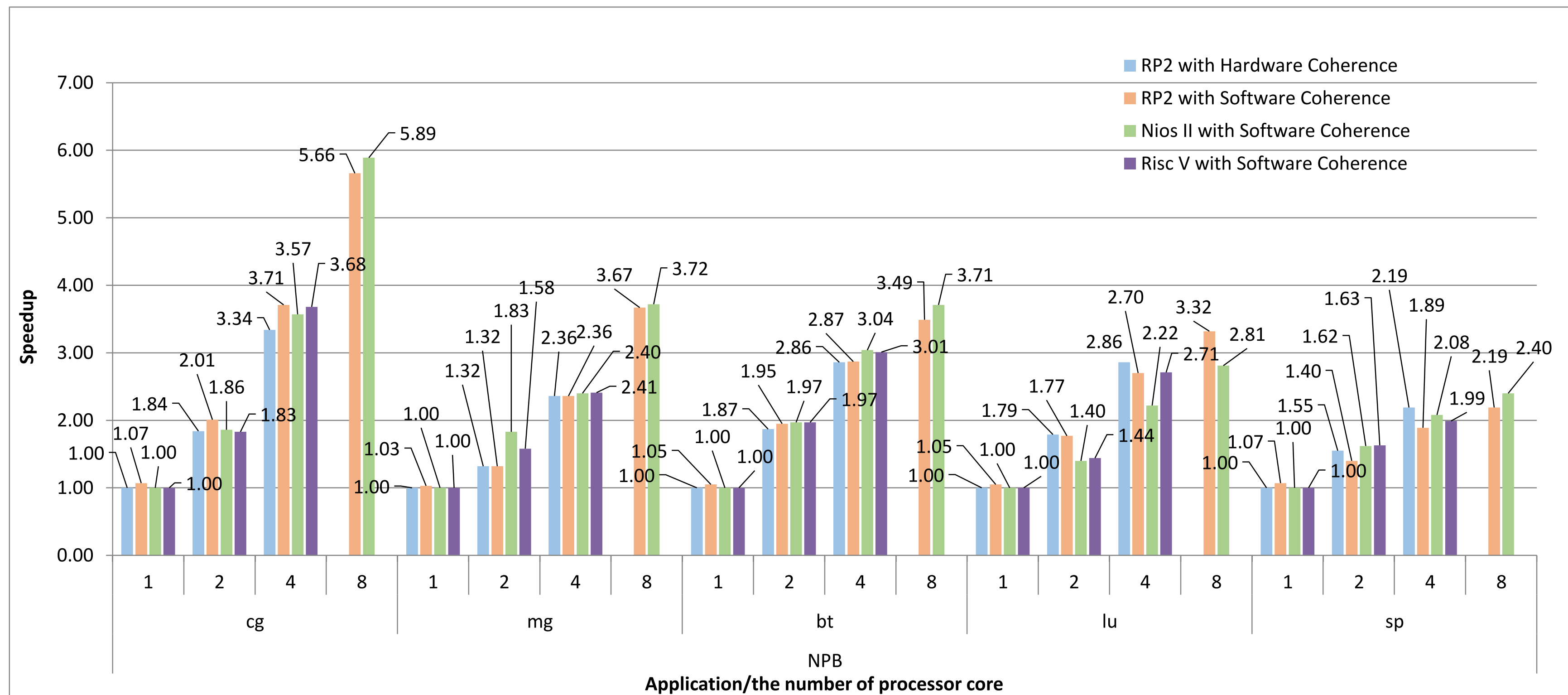


Figure 5-3: The performance comparison of RP2 Processor, Nios II and RISC-V multicore for NAS Benchmark.

5.2.1 Relative Speedup

Figure 5-1 and 5-2 show the speedup of the RP2 based system, for both hardware-based coherence and the proposed compiler controlled cache coherence. On these figures, the single-core performance with hardware-based coherence was selected as the baseline. All Nios II system performances are measured relative to their respective single-thread performance. Meanwhile, Figure 5-3 compares all three platforms except the Intel Xeon based machine.

Figure 5-1 and 5-2 show that the software-based coherence provides a comparable speedup compared to hardware-based coherence for up to four cores. Beyond four cores, the software-based coherence enables eight-core executions with a good speedup, which was initially impossible due to the lack of hardware coherence mechanism.

The performance of the proposed software-based approach gives us about 4% - 14% better performance compared to the hardware-based mechanism. For example, SPEC2006 "lbm" has 1.76 times speedup with hardware cache coherence and 1.9 times speedup with the proposed method on two cores. The proposed method is 7% faster than hardware-based coherence. The same benchmark, on four cores, scores 2.9 times speedup and 3.28 times speedup for the hardware-based coherence and the proposed method, respectively, which means 13% performance gain with the proposed method. Furthermore, the proposed method allows eight-cores execution with respectable 4.76 times speedup, which was previously unattainable due to the lack of a hardware-based mechanism for more than four cores execution. The performance gain is attributed to the reduction of bus activity in the RP2 platform. The RP2 platform has a snooping-based hardware cache coherence mechanism, which means each writes to the cache costs overhead for the invalidation packet sent to other cores on the bus. Meanwhile, the software-based approach does not require the transmission of such a packet as the compiler will insert self-invalidate instruction to the appropriate core. Therefore, the performance gain can be observed. The SPEC Benchmark "art", "quake", "lbm" , and "hmmer" are positively affected by software-based coherence as shown in Figure 5-1.

In Figure 5-2, albeit not as strong as for the SPEC benchmark, similar perfor-

mance gain for NPB is also observed on most benchmarks. Similar or slightly better performance is measured on NPB “cg”, “mg”, and “bt”. Meanwhile, on NPB “lu” and “sp” benchmark, the performance of the software-based approach suffers due to the cache-space wasting for stale data handling. Array padding and expansions reduce the amount of usable cache space in the system. On the other hand, NPB “cg” is a conjugate-gradient calculation with many DOACROSS loops. Selective cache operation allows a better performance by reducing the number of self-invalidation.

In Figure 5-3, both the Nios II and RISC-V based soft multicore could run the benchmark with respectable speed up for up to eight and four cores, respectively. Both systems do not have have any hardware-based cache coherence mechanism. The Nios II SoC is very simple and generated entirely by the Altera Platform Designer with minimum glue logic. Without the compiler support, writing a parallel program with a good speedup for this platform is close to impossible. The proposed method successfully provides an automatic speedup for this platform.

5.2.2 Performance Impact of the Proposed Method

Both the stale data handling method and false sharing handling method impose some overhead to the overall performance. The stale data handling mechanism inserts extra instructions, and the false sharing method reduces the effective cache space. On an SMP machine equipped with a hardware cache coherence mechanism, both methods are typically not activated. But, in order to understand the performance penalty of both methods, some SPEC benchmark programs were run on the RP2 platform with several different combinations of the methods.

Figure 5-4 shows the relative performance impact of each proposed method compared to the hardware-based mechanism:

- **SMP** is a standard shared memory architecture with the hardware-based coherence mechanism turned on. This is the baseline of the measurement.

- **Stale Data Handling + False Sharing without Hardware Coherence:** this is the performance of the proposed method with hardware coherence control turned off. As explained above, software-based coherence eliminates the amount

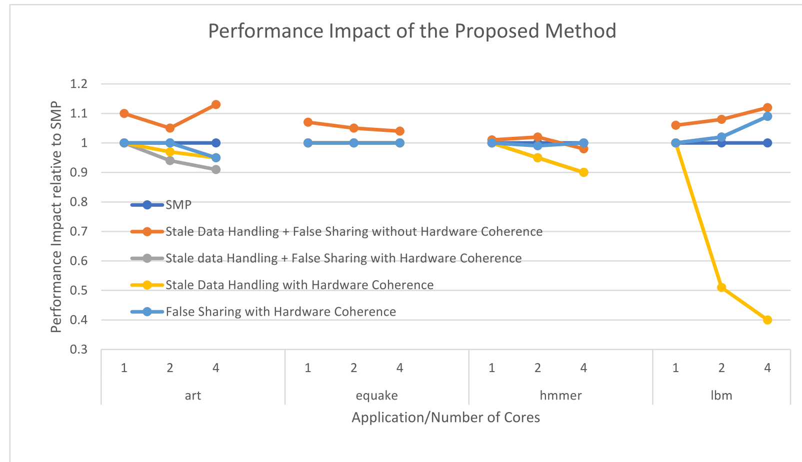


Figure 5-4: The performance impact of software cache coherence. Image from: [1]

of invalidate packets sent through the bus, thus providing a better performance compared to the hardware-based approach.

- **Stale Data Handling with Hardware Coherence:** stale data handling method with hardware-based coherence mechanism still turned on. The performance is negatively impacted. This is expected since the stale data handling method inserts unnecessary self invalidate instructions for every iteration of the busy loop, adding more delay to the program without any purpose.

- **False Sharing with Hardware Coherence:** this shows the impact of false sharing handling methods, which comprise of data alignment, cache line aligned data decomposition, and other layout transformation with hardware coherence control still turned on. This kind of combination provides various results. In "equake" and "hmmer", the cache line wasting effect is insignificant, thus resulting in a similar performance to the SMP machine. In "lbm", this approach improves performance. This is to be expected since false sharing is also bad even for hardware-based cache coherence control. Removing false sharing problems will improve the performance of a hardware-based coherence control. Meanwhile, for "art", false sharing wastes too much cache space resulting in poor performance.

- **Stale data Handling + False Sharing with Hardware Coherence:** this graph shows the overhead of both proposed methods for handling stale data and

false sharing turned on with hardware coherence still active. The combination of both methods turned on is usually very similar to **Stale Data Handling with Hardware Coherence** except for "art" in which the false sharing avoidance method lowers the performance even more.

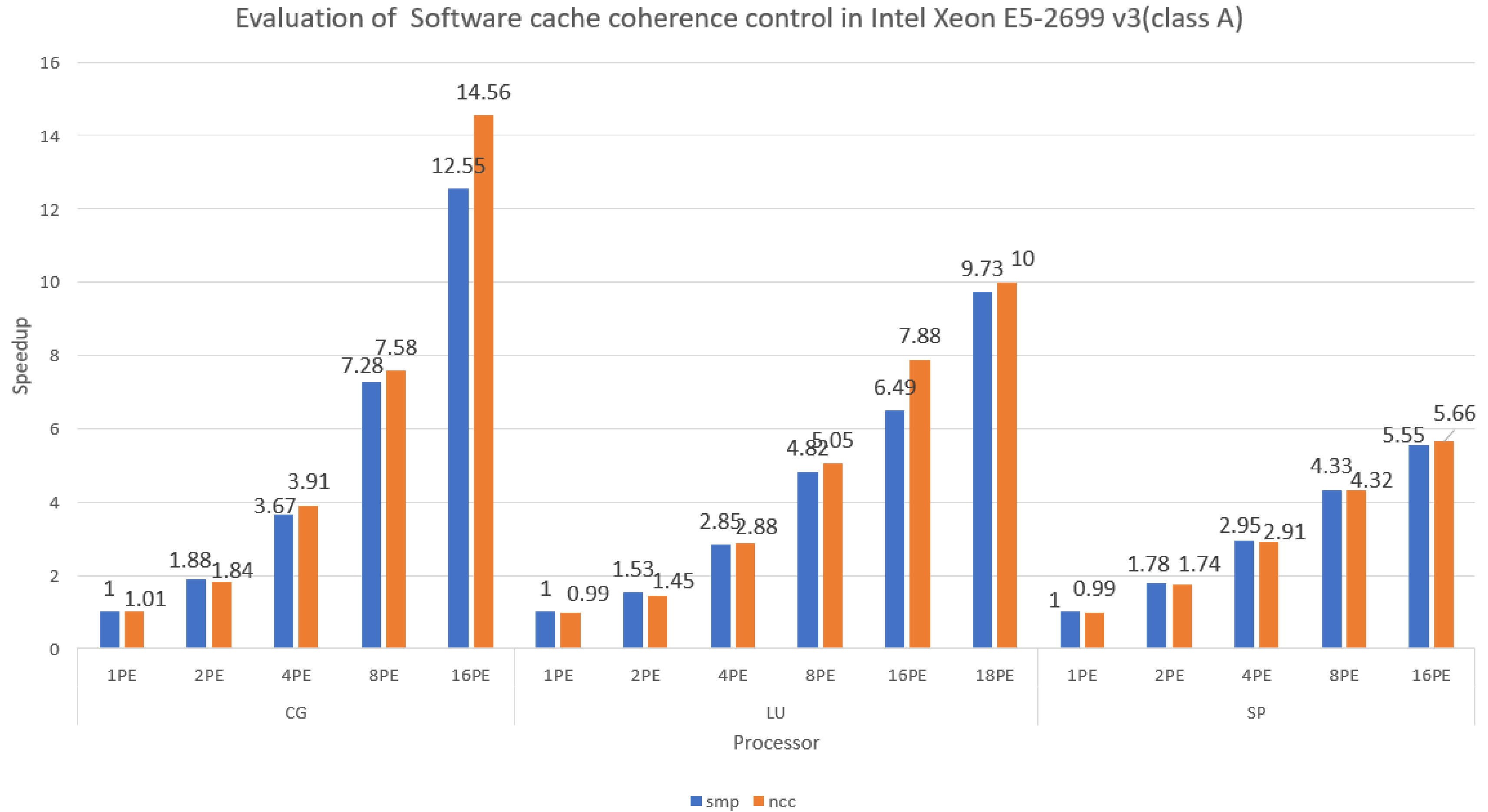


Figure 5-5: The performance benefit of false sharing avoidance in Intel SMP cache coherent machine on NAS Parallel Benchmark class A data size. Image from [1]

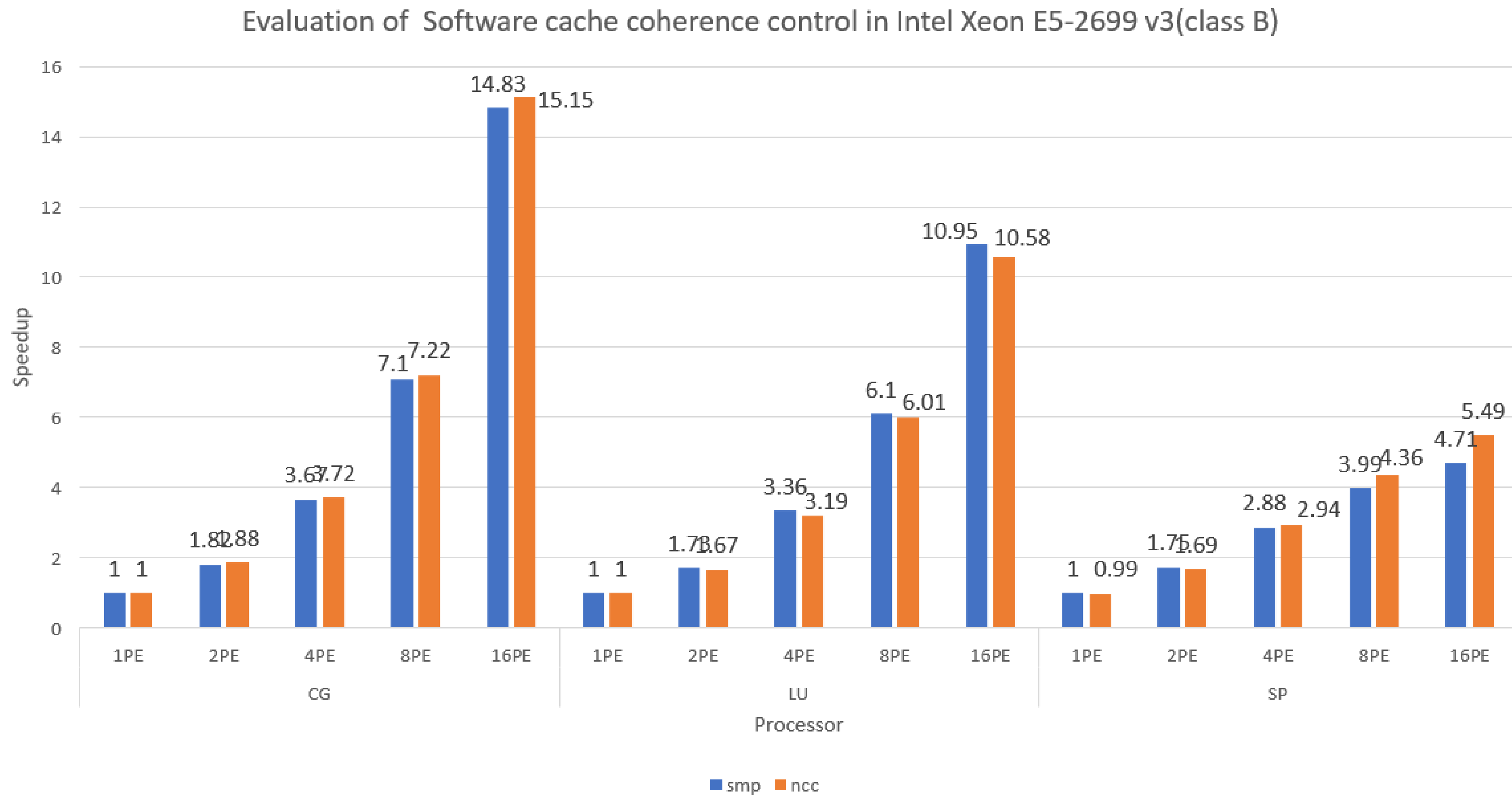


Figure 5-6: The performance benefit of false sharing avoidance in Intel SMP cache coherent machine on NAS Parallel Benchmark class B data size. Image from [1]

5.2.3 Performance Benefits of False Sharing Avoidance on SMP Machine

To further investigate the performance impact of the proposed false sharing mitigation on an SMP cache coherent system, the same OSCAR Compiler parallelized benchmark applications are executed on such a system. The self invalidation method is turned off. The SMP system used is Intel Xeon E5-2699 v3 CPU with 128 GB of DDR4 memory running a standard Linux OS. On this platform, as it is much faster and has significantly larger memory compared to the previous embedded platform, the NPB benchmark uses Class A and Class B data sizes to measure the running time reliably.

In the previous sub-section, false sharing avoidance methods sometimes improves performance. But, most often, it also lowers the performance due to cache space wasting. The RP2 platform is very sensitive to cache-space wasting due to its small size. For a better understanding of this effect, some NPB benchmark programs are compiled with false sharing avoidance mechanism turned on. The result in Figure 5-5 and 5-6 shows that on an SMP cache-coherent machine, while the self invalidation scheme is not useful, the false sharing prevention still helps to improve the performance. However, there is a slowdown on class B data size due to a reduction in effective cache size.

Chapter 6

Conclusions

6.1 Summary of Works

This thesis describes a method to manage cache coherency by OSCAR Compiler, an automatic parallelizing compiler, for a Non-Coherent Cache (NCC) system. The OSCAR Compiler decomposes an input program into the coarse grain task, then analyzes control flow and data dependence between the task. After the compiler figures out the earliest executable condition for each task, creates a static parallel execution schedule, then it analyses stale data and false sharing problem between the tasks. The compiler solves the stale-data problem by self-invalidation and synchronization. It also prevents false sharing problems with simple data restructuring, i.e., cache alignment, array expansion, array padding, and non-cacheable buffer. This part of the work was jointly developed with Mase, M. and Kishimoto, Y. on Kasahara laboratory, Waseda University[18].

For evaluating the performance of the method above, in addition to the existing Renesas RP2 platform, two new multicore systems are developed, namely the 8-core Nios II multicore, and the quadcore RISC-V multicore, both without hardware-based cache coherence mechanism. The Nios II multicore is very simple and generated entirely in the Altera Platform Designer. Meanwhile, the RISC-V is an emerging new popular opensource platform that originally does not support software cache manipulation. A new software-controllable cache is created and integrated into the system.

The proposed compilation method is then evaluated using ten benchmark programs from SPEC2000, SPEC2006, NAS Parallel Benchmark (NPB), and MediaBench II on Renesas RP2 8 core multicore processor. Due to the limitation of the evaluation platform, only NPB is used for both the 8-core Nios II multicore and the quadcore RISC-V multicore on Altera FPGA.

The performance of the NCC architecture with the proposed method is comparable or better than the hardware-based coherence scheme. For example, the RP2 platform with a hardware-based coherence mechanism gave 3.34 times speedup on four core for NPB "cg" while without any hardware-based cache coherence, the proposed method provided 3.71 times speedup on four cores. Moreover, the proposed method allowed

execution on all eight cores, which was originally not supported by the hardware cache coherence mechanism, with 5.66 times speedup. For the same benchmark program, the proposed method gave 3.68 times speedup on four cores RISC-V multicore and 3.57 and 5.89 times speedup on four and eight cores Nios II multicore. The proposed method also allows us to parallelize automatically and easily run the benchmark program on 8-cores Nios II multicore and four cores RISC-V multicore, which both are not designed for cache-coherent operation.

The results above show that the proposed method provides competitive performance advantages against the traditional hardware-based coherence control mechanism for the same number of processor cores. Furthermore, it gives a respectable speedup automatically for any number of processor cores without the hardware coherent control mechanism regardless of the processor architecture. Meanwhile, usually, programmers had to spend much effort to develop an application for the NCC platform.

The novelty of this research are:

- The first compiler controlled cache coherency that handles both true sharing and false sharing with automatic parallelization, which was successfully tested on practical hardware platforms.
- The first implementation of software cache control for RISC-V in FPGA.

6.2 Future Works

In the near future, wafer-level massively parallel multicore will become prominent. On such level of integration, a thousands-cores processor supported by on-chip memory with a supercomputer-level-memory bandwidth becomes possible. Relying on hardware cache coherence may not be feasible due to its complexity. This research provides an alternative to such a mechanism, freeing more silicon area for actual computational logic.

Likewise, this research also allows us an easy way to create a multicore SoC on FPGA with an automatic SoC builder currently available in the market. Most of the

SoC builders do not support generating a cache coherency mechanism automatically. Automatically generated SoC paired with an automatic parallelizing compiler with NCC support will provide a simple solution to speed up the execution of a complex algorithm.

Currently, support for multiple levels of cache and a more complex out-of-order 64-bit RISC-V core are under development. The algorithm needs further development to support multiple levels of private and cluster level cache. Compiler controlled processor-accelerator cache coherency is also considered. This allows better support for automatic accelerator code generation with OSCAR Compiler.

Acknowledgments

I am very thankful to Prof. Hironori Kasahara as my Ph.D. supervisor and Prof. Keiji Kimura as my co-supervisor for their invaluable support during my Ph.D. study. I would like to thank to Prof. Nozomu Togawa, for the constructive critics and suggestion. I am also very grateful for the help of Dr. Masayoshi Mase, whose works provided the foundation of this research effort. Thank you for Mr. Tomoya Kashimata and Mr. Ken Takahashi for helping me with the Nios II SoC and API Translator, also for Mr. Taisuke Onishi, who provided the evaluation on Intel platform. Thank you to all Kasahara and Kimura laboratory members who directly or indirectly helped me during my research.

I would like to express my gratitude to Hitachi Global Foundation for supporting my Ph.D. study. Without their support, it is impossible for me to finish this research. Special thanks to Mr. Kazuyuki Miyanaga, Mr. Masaaki Kawamoto, and Ms. Tamami Ono from Hitachi Global Foundation for their kind and understanding support.

This Ph.D. study was sponsored by Hitachi Global Foundation Scholarship batch 2015 grant number 178. Part of this research was also based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

Bibliography

[1] Boma Anantasatya Adhi, Tomoya Kashimata, Ken Takahashi, Keiji Kimura, and Hironori Kasahara. Compiler software coherent control for embedded high performance multicore. *IEICE Transaction on Electronics Special Section on Low-Power and High-Speed Chip*, Vol. E103(No.3):85–97, 2020.

[2] Boma Anantasatya Adhi, Masayoshi Mase, Yuhei Hosokawa, Yohei Kishimoto, Taisuke Onishi, Hiroki Mikami, Keiji Kimura, and Hironori Kasahara. Software cache coherent control by parallelizing compiler. In Lawrence Rauchwerger, editor, *Languages and Compilers for Parallel Computing*, pages 17–25, Cham, 2019. Springer International Publishing.

[3] Motor Industry Software Reliability Association. *MISRA-C: 2012: Guidelines for the Use of the C Language in Critical Systems.* HORIBA MIRA, 2019.

[4] Yan Bao and Mats Brorsson. An implementation of cache-coherence for the nios ii ™ soft-core processor. 2009.

[5] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[6] BSD Foundation. The 2-Clause BSD License, https://opensource.org/licenses/bsd-license.php.

[7] Yung-Chin Chen and Alexander V Veidenbaum. Comparison and analysis of software and directory coherence schemes. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 818–829, New York, NY, USA, 1991. ACM.

[8] H Cheong and A V Veidenbaum. A cache coherence scheme with fast selective invalidation. In *[1988] The 15th Annual International Symposium on Computer Architecture. Conference Proceedings*, pages 299–307, May 1988.

[9] H Cheong and A V Veidenbaum. Compiler-directed cache management in multiprocessors. *Computer*, 23(6):39–47, June 1990.

[10] 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. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 155–166, October 2011.

[11] Lynn Choi and Pen Chung Yew. A compiler-directed cache coherence scheme with improved intertask locality. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, Supercomputing '94, pages 773–782, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[12] D Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, May 2005.

[13] Giovani Gracioli and Antônio Augusto Fröhlich. On the design and evaluation of a real-time operating system for cache-coherent multicore architectures. *SIGOPS Oper. Syst. Rev.*, 49(2):2–16, January 2016.

[14] John L Hennessy and David A Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017.

[15] J Howard, S Dighe, Y Hoskote, S Vangal, D Finan, G Ruhl, D Jenkins, H Wilson, N Borkar, G Schrom, F Pailet, S Jain, T Jacob, S Yada, S Marella, P Salihundam, V Erraguntla, M Konow, M Riepen, G Droege, J Lindemann, M Gries, T Apel, K Henriss, T Lund-Larsen, S Steibl, S Borkar, V De, R V D Wijngaart, and T Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*, pages 108–109, February 2010.

[16] Intel. *Nios II Software Developer Handbook.* 2019.

[17] 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. In *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, pages 90–598, February 2008.

[18] H Kasahara, H Honda, A Mogi, A Ogura, K Fujiwara, and S Narita. A multi-grain parallelizing compilation scheme for oscar (optimally scheduled advanced multiprocessor). In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 283–297, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[19] Hironori Kasahara. *Parallel Processing Technology.* CORONA PUBLISHING CO., LTD., June 1991.

[20] Hironori Kasahara, Keiji Kimura, Boma Anantasatya Adhi, Y Hosokawa, Yohei Kishimoto, and M Mase. Multicore cache coherence control by a parallelizing compiler. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 01, pages 492–497, July 2017.

[21] Hironori Kasahara, Keiji Kimura, and M Mase. Method of generating code executable by processor, 2012.

[22] Stephen W Keckler, Kunle Olukotun, and H Peter Hofstee. *Multicore Processors and Systems*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[23] Keiji Kimura, Akihiro Hayashi, Hiroki Mikami, Mamoru Shimaoka, Jun Shirako, and Hironori Kasahara. 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.

[24] Yohei Kishimoto, Masahiro Mase, Keiji Kimura, Hironori Kasahara, and Others. Evaluation of software cache coherency control method by an automatic parallelizing compiler (japanese). *IPSJ SIG Technical Report (HPC)*, 2014(19):1–7, 2014.

[25] Leonidas I. Kontothanassis and Michael L. Scott. High performance software coherence for current and future architectures. *Journal of Parallel and Distributed Computing*, 1995.

[26] Milo M K Martin, Mark D Hill, and Daniel J Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, July 2012.

[27] Masayoshi Mase, Yuto Onozaki, Keiji Kimura, and Hironori Kasahara. Parallelizable c and its performance on low power high performance multicore processors. 2010.

[28] J Nowotsch and M Paulitsch. Leveraging multi-core computing architectures in avionics. In *2012 Ninth European Dependable Computing Conference*, pages 132–143, May 2012.

[29] Mark S Papamarcos and Janak H Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, pages 348–354, New York, NY, USA, 1984. Association for Computing Machinery.

[30] Fong Pong and Michel Dubois. Verification techniques for cache coherence protocols. *ACM Comput. Surv.*, 29(1):82–126, March 1997.

[31] Douglass Post. The Future of Computing Performance. *Computing in Science and Engineering*, 13:4–5, 2011.

[32] Steven K Reinhardt, Mark D Hill, Bradford M Beckmann, and Arkaprava Basu. Cmp directory coherence: One granularity does not fit all. 2013.

[33] S Schliecker, J Rox, M Negrean, K Richter, M Jersak, and R Ernst. System level performance analysis for real-time automotive multicore and network architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):979–992, July 2009.

[34] Yan Solihin. *Fundamentals of Parallel Multicore Architecture*. Chapman & Hall/CRC, 1st edition, 2015.

[35] SpinalDHL. VexRiscv, https://github.com/SpinalHDL/VexRiscv.

[36] S Tavarageri, W Kim, J Torrellas, and P Sadayappan. Compiler support for software cache coherence. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 341–350, December 2016.

[37] Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanovic. The risc-v instruction set manual v2.1. 2019.

[38] Y Yoshida, T Kamei, K Hayase, S Shibahara, O Nishii, T Hattori, A Hasegawa, M Takada, N Irie, K Uchiyama, T Odaka, K Takada, K Kimura, and H Kasahara. A 4320MIPS Four-Processor Core SMP/AMP with Individually Managed Clock Frequency for Low Power Consumption. In *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pages 100–590, February 2007.

[39] Xin Yuan, R Melhem, and R Gupta. A timestamp-based selective invalidation scheme for multiprocessor cache coherence. In *Parallel Processing, 1996. Vol.3. Software., Proceedings of the 1996 International Conference on*, volume 3, pages 114–121 vol.3, August 1996.

Achievement

Paper

○ Boma A. Adhi, Tomoya Kashimata, Ken Takahashi, Keiji Kimura, Hironori Kasahara "Compiler Software Coherent Control for Embedded High Performance Multicore", IEICE Transaction on Electronics Special Section on Low-Power and High-Speed Chip, Vol.E103-C, No.3, Mar. 2020. pp.85-97

○ Boma A. Adhi, Masayoshi Mase, Yuhei Hosokawa, Yohei Kishimoto, Taisuke Onishi, Hiroki Mikami, Keiji Kimura, Hironori Kasahara "Software Cache Coherent Control", The 30th International Workshop on Languages and Compilers for Parallel Computing (LCPC2017), College Station, Texas, Oct. 11-13, 2017, pp. 17-25

Hironori Kasahara, Keiji Kimura, Boma A. Adhi, Yuhei Hosokawa, Yohei Kishimoto, Masayoshi Mase "Multicore Cache Coherence Control by a Parallelizing Compiler", 2017 IEEE 41st Annual Computer Software and Application Conference (COMPSAC), Turin, 2017, pp. 492-497

Boma A. Adhi, Masayoshi Mase, Yuhei Hosokawa, Yohei Kishimoto, Taisuke Onishi, Hiroki Mikami, Keiji Kimura, Hironori Kasahara "Software Cache Coherent Control", Lecture Notes in Computer Science, vol LNCS 11403. Springer, 2019, pp. 17-25

Academic Exhibition

Hiroki Mikami, Boma A. Adhi, Tomoya Kashimata, Satoshi Karino, Kazuki Miyamoto, Takumi Kawata, Ken Takahashi, Tetsuya Makita, Tomoya Shirakawa, Yoshitake Oki, Toshiaki Kitamura, Keiji Kimura, Hironori Kasahara "OSACRA Vector Multicore System Platinum Vector Accelerator on FPGA", IEEE Supercomputing 2019, Denver, Colorado, Nov. 12-17, 2017