INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2023, VOL. 69, NO. 2, PP. 371-382
Manuscript received January 31, 2023; revised April, 2023. DOI: 10.24425/ijet.2023.144373

Shared-Semaphored Cache Implementation for Parallel Program Execution in Multi-Core Systems

Adam Milik, and Michał Walichiewicz

***Abstract*—The paper brings forward an idea of multi-threaded computation synchronization based on the shared semaphored cache in the multi-core CPUs. It is dedicated to the implementation of multi-core PLC control, embedded solution or parallel computation of models described using hardware description languages. The shared semaphored cache is implemented as guarded memory cells within a dedicated section of the cache memory that is shared by multiple cores. This enables the cores to speed up the data exchange and seamlessly synchronize the computation. The idea has been verified by creating a multi-core system model using Verilog HDL. The simulation of task synchronization methods allows for proving the benefits of shared semaphored memory cells over standard synchronization methods. The proposed idea enhances the computation in the algorithms that consist of relatively short tasks that can be processed in parallel and requires fast synchronization mechanisms to avoid data race conditions.**

***Keywords*—thread synchronization; scheduling; mapping; parallel execution; compiler**

I. Introduction

A. Concurrent Execution Models

There can be distinguished two basic data processing formats. The graphical processing unit (GPU) consists of multitude of cores that share the same cache; they are coordinated together by a main core. This design allows processing data set of mutually exclusive data in parallel using the same computation formula [4].

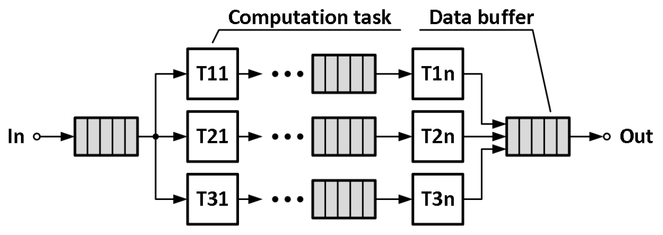


Fig. 1. Diagram Of Parallel Execution Model

This work was supported by funding from the Ministry of Education and Science for Statutory Activities of Digital Systems Division of the Silesian University of Technology of Gliwice (BK247/RAu12/2022 and BKM576/RAu12/2022).

Authors are with Silesian University of Technology, Gliwice, Poland, Faculty of Automatic Control, Electronics and Computer Science, Digital Systems Division, Poland (e-mail: adam.milik@polsl.pl, michal.walichiewicz@polsl.pl).

The following fig.1 represents the typical parallel execution model of GPUs. Each data from a data set is concurrently computed by the same task. The GPU compilers generate code for multiple-threads by vectorization of a generated intermediate representation code as one of the steps within the GPU's compiler code optimization. [4], [6]. Described model falls into Single Instruction Multiple Data (SIMD) multiprocessing scheme. Artificial intelligence, image processing, aerodynamics, etc. assimilate parallel data processing due to calculations of huge data sets [2]–[4].

The central processing units (CPUs) are single or multi-core systems that perform parallel or diverse sequential processing on the mutually inclusive or exclusive set of data. The CPUs most often execute multiple sequentially independent tasks which do not share any data. This universality makes the architecture more complex [2]–[4].

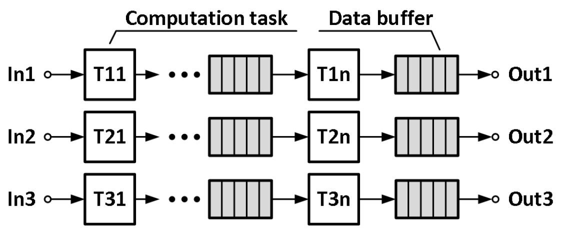


Fig. 2. Diagram Of Multiple Sequential Execution Model

The general purpose multi-core CPUs are the extension of the von Neuman machine (Singel Instruction Single Data - SISD) toward multiple machines sharing common memory space (Multiple Instructions Multiple Data - MISD). There can be considered a situation when the multiprogram operation is implemented where each core executes an independent program as shown in fig.2. The utilization of multiple cores for performing mutually dependent computations requires creating synchronization mechanisms eliminating the data race conditions. This allows the distribution of computations of a single program (a computation problem) across several processing units. Depending on the operating system (supervisor) implementation there can be distinguished different synchronization mechanisms available through the operating system functions call or are implemented independently by user programs dedicated synchronization mechanisms [4], [11], [12]. The compilers often are oriented to deliver the SISD implementation program passing the problem of partitioning and synchronizing the computations to the user. From the user (programmer) point of view this is a complicated task. This requires not only careful computation implementation but also requires partitioning to multithreaded execution. There arises the problem of race conditions between threads or deadlock conditions when multiple conditional paths are managed. The debugging is problematic and still remains the question about the partitioning correctness and effectiveness. In many applications parallel processing is possible but it is inefficient due to a lack of fast computing thread synchronization mechanisms disallowing for efficient utilization of multiple core architectures. At this point should be recalled specific types of computations performed by digital system simulators (event-driven simulators e.g. hardware description simulation, SystemC) and Programmable Logic Contorllers (PLCs) based on IEC61131-3 standard.

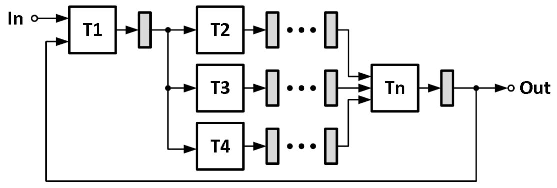


Fig. 3. Computation model of PLC control processing or digital simulation

The execution model in the fig.3 illustrates computations which are being done in computing control algorithms in PLCs or simulation of HDL models (event-driven simulation). Such models of execution enables parallel execution provided synchronization of computation task is efficient. The HDL model requires additionally an efficient scheduler that allows selecting tasks to be processed according to observed signal changes. The control program processing in PLCs requires only data dependency triggering that organizes ordered computation task processing. In this case scheduling problem can be simplified to static distribution of computation tasks across available processing cores. Both problems requires the correct synchronization of multiple computation threads sharing the same data. [5].

B. Shared-Memory Synchronization

The shared memory allows for fast data exchange between processors of the system. The Organization of memory hierarchy such as the distribution of local and shared caches between multiple cores, types of caches and the implementation of different cache coherence protocols are the factor contributing to the multi-threaded execution performance. [3]–[5].

The memory hierarchy model is shown in fig.4. The organization of memory increases the data exchange performance by implementing small size fast cache memories that reflects random data access of CPU to a single memory cell while the main memory (typically implemented as dynamic memory) is located behind the cache that exchanges entire data line allowing for fast and efficient data burst exchange. Additionally, a memory hierarchy allows having shared caches between multiple cores which allows for faster data synchronization by omitting a long time-consuming hardware mechanism of writing through the shared data back to the main memory.

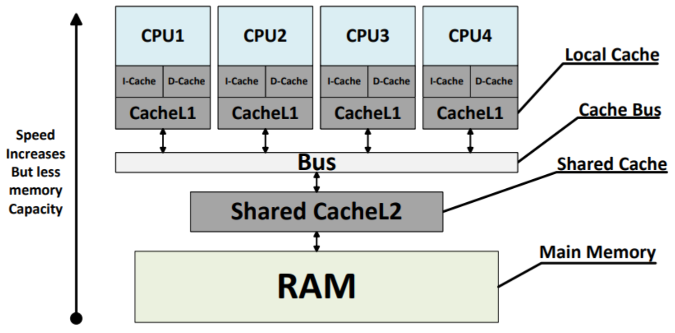


Fig. 4. Diagram Of Memory Hierarchy Model

Shared caches are connected to local caches by a bus which requires the implementation of cache coherency mechanisms through Cache Coherency Controllers (Snoopers) to maintain the cache coherency in the local caches [3], [5].

Listing 1. Atomic Operation Lock

```
1: Lock L
2: Thread 1:              Thread 2:
3:     L.acquire()             L.acquire()
4:         $cnt$++;                 $cnt$++
5:     L.release()             L.release()
```

It is expected to have data races in the shared data synchronization that is originated with differences in the completion time of particular computation tasks. The data races in the shared data synchronization can corrupt the results of computations. It is necessary to implement a mechanism which would allow a thread to acquire and lock a block of data, and let other threads wait for the given data to be released. As it is shown in the **Atomic Operation Lock**, the first thread is going to acquire and lock data, which forces the second thread to wait until the other thread releases the lock after the increment of the *cnt* variable.

The Primitive atomic instructions are used to get exclusive access to data. The compilers that provide functionality implements control blocks that guarantee variable protection for requesting thread [5].

II. Synchronization

A. Primitive Atomic Instructions

Introducing the automatic parallel execution of algorithms that are represented by sequential languages (here PLC languages are considered) or sequential processing of concurrent or event-driven modelling requires precise task scheduling and synchronization. The task can be simplified to the function that is described over three sets of variables **X, Y, Q**.

$$\{Y, Q\} = f(X, Y, Q) \tag{1}$$

Where: **X** is a set of input variables, **Y** is a set of output variables and **Q** is a set of internal state variables. In order to retain the function in processing, there should be a nonempty

X set. When the **Q** set is empty the described processing is memoryless and the result of processing depends only on the current value of variables in the **X** set, independent of the previous history of changes. The processing order of the task is created based on the variable dependencies. The function cannot be scheduled before all arguments are known. In order to meet this requirement, a synchronization of computations between tasks is necessary, while in general the task execution time varies and depends on logic conditions. The processing start of the task(s) must be synchronized. The considered situation resembles the transformation of the tasks to be executed into a Petri net, where places denote tasks and transitions are unconditional. Directed arcs record the order of control passing. The token passing is only possible when all preceding tasks possess the token. The possession of the token represents the completion of processing. The synchronization mechanisms allow correct data flow across the task, independently of completion time variation of particular tasks. The number of available cores and task ordering enables some optimizations in the observation of tokens' arrival. Let assume that the program is partitioned statically between processing cores. Here are examined the commonly used atomic procedures for data exchange

Listing 2. Atomic Compare-And-Swap

```
procedure CAS($Data\_Pointer, Old, New$)
    if $Data\_Pointer \neq Old$ then
        return $False$
    else
        $Data\_Pointer := New$
        return $True$
    end if
end procedure
```

Listing 3. Atomic Load-Linked/Store-Conditional

```
procedure LL($Data\_Pointer$)
    Link $Data\_Pointer$
    return $Data\_Pointer$
end procedure

procedure SC($Data\_Pointer, New$)
    if No Change since LL in $Data\_Pointer$ then
        $Data\_Pointer := New$
        return $True$
    else
        return $False$
    end if
end procedure
```

The given CAS and LL/SC are most often use atomic primitive instructions that are capable of implementing shared data algorithms for an arbitrary number of threads. However, **Atomic Compare-And-Swap** can lead to the ABA problem in the multi-core systems which can lead to unpredicted behaviours (in the critical case to deadlock condition). The implementation of local caches in each core makes this phenomenon possible and fatal [5].

B. Mapping Multi-Threaded Algorithms

In order to map the sequentially described processing over multiple processing resources the computation dependencies and complexity must be determined. The computation process is expressed by means of data flow graphs (DFGs) and control flow graphs (CFGs) as a result of systematic analysis of language statements [10]. Using dual graph representation introduces difficulties in revealing parallel processing. This is caused by the sequential nature of the control flow graph. It could be observed that a correctly formulated control program can be represented by acyclic data and a control flow graph after merging. It illustrates all possible processing in a single processing cycle of a control program. Owing to the graph representation of computations, mutual dependencies can be explored, and possible parallelism can be used for speeding up the computations. An exemplary control and data flow graph is shown in fig.5.

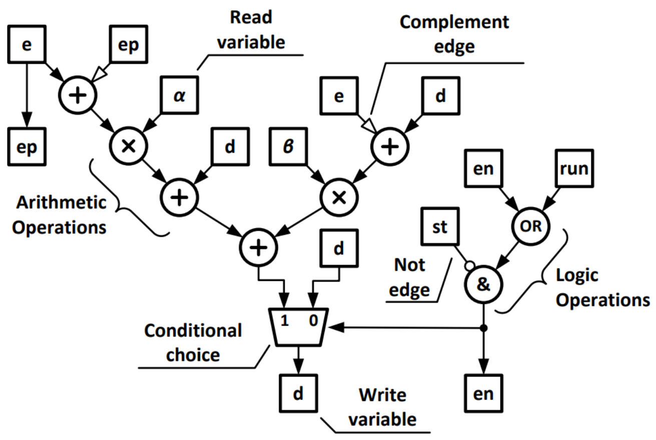


Fig. 5. Diagram Of Data Flow

The mentioned tools are used in compilers to optimize the code written in high-level syntax to assembly. [6]–[8]. Besides generating a more optimized assembly code for a CPU, it also enables to map a multi-threaded execution code for algorithms such as **Parallel Execution** shown below.

Listing 4. Parallel Execution

```
function PARALLEL($Data\_Buffer$)
    for all $Data\_Buffer$ do
        Spawn_Work()
    end for
end function
```

The GPUs tend to be most efficient at computing multiple data in parallel; however, it is only for algorithms which can be vectorized such as **Parallel Execution**; thus, each data in the set must have the same processing scheme which is the SIMD mentioned previously. The following fig.6 serves as a straightforward diagram of the execution model which GPU can compute faster than CPU.

The data buffer in the diagram consists of three data which have to be sequentially computed by a task T1. The total

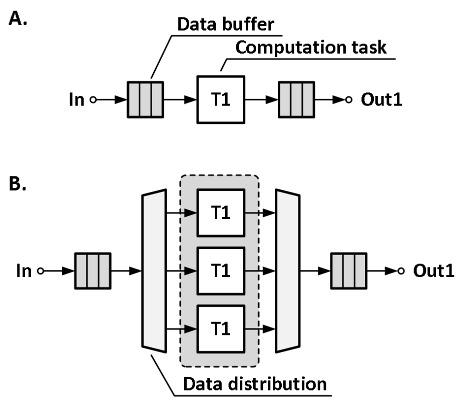


Fig. 6. Diagram Of Execution Model using sequential (A) and vectorized approach (B)

sequential computation of the data buffer takes 3 processing cycles of T1. Such an Execution model can be vectorized which produces a parallel execution model as shown in case B of fig.6. The vectorized execution model permits the GPU to execute the whole data buffer in parallel by three processing units sharing a common control scheme resulting in a triple performance increase.

The GPU's cores are controlled by a main core, and its internal memory allows multitude of cores to write and read huge data blocks simultaneously. However, algorithms in which each data of the buffer, is computed by distinctive tasks, and the anterior functions depend on the prior inputs and outputs of the posterior computations as in the fig.7. In such execution models, vectorization is very limited. This imposes a serious problem for the GPU to compute [4], [6]–[8].

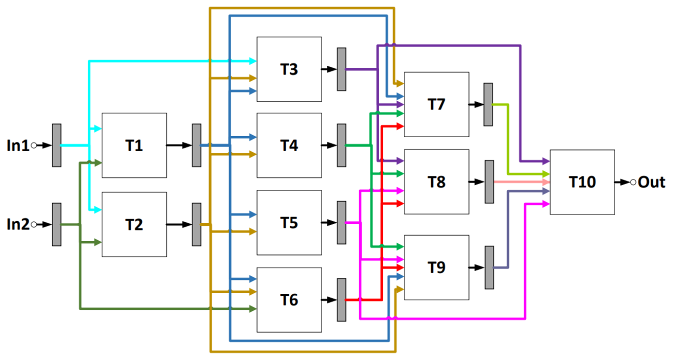


Fig. 7. Execution Model Which cannot be vectorized

C. Multi-Threading and Compiler Optimizers

The computation models such as shown in fig.7 are not impossible for a compiler to analyze. Compilers often are divided into front-end, middle-end and back-end layers. The front-end is responsible for parsing and generating the Intermediate Representation (IR) instruction of the code. The Middle-end as it can be seen in the fig.8, is responsible for running optimizes (also called Passes) to generate optimized code for a specific problem such as vectorization of the execution model (which was outlined in the previous section). [6]–[8].

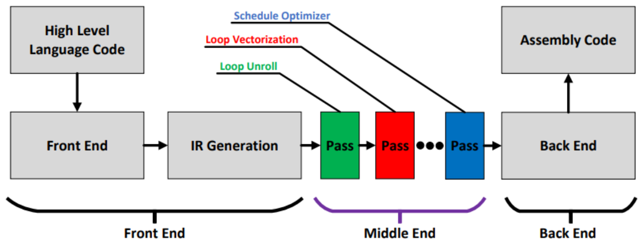


Fig. 8. Diagram of Compiler Architecture such as LLVM

The Intermediate Representation Code is the compiler's private data structure to represent a parsed code from a high-level language. Each compiler has its own IR structure from which, it is able to analyze, optimize and generate assembly code.

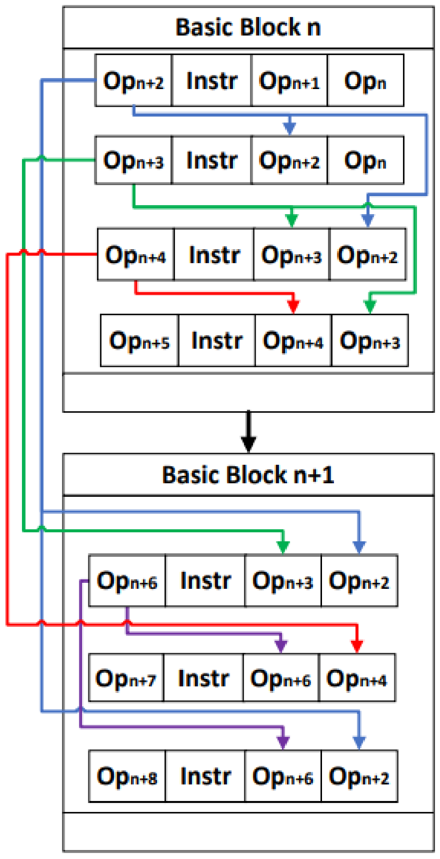


Fig. 9. Exemplary intermediate representation using Basic Blocks and IR Instructions Dependency

The intermediate representation is organized into "Basic Blocks" which contain intermediate representation instructions. Every IR Operand can be used only once as a Result Operand; thus, it is very simple to trace data flow and dependency as it can be noticed in the fig.9.

In the following example shown in fig.10. It can be deduced that there is shared data dependency in the Sensor A and B

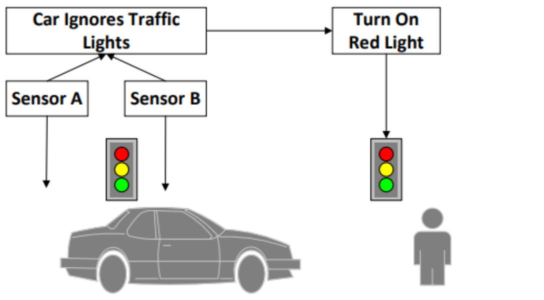


Fig. 10. Data Dependency Example using automatic pedestrian crossing

functions. However, Sensors' functions introduce branches in the data flow shown with red arrows in the fig.11. The branches do not change the data dependency, but it only forces multiple cores to execute longer chunks of code before synchronization occurs as it is shown with the blue and green arrows.

As it was shown, a custom optimizer can map multi-threaded programs for the execution models such as fig.7 by tracing the data dependency and in the case of more complex programs through a combination of data flow and dependency.

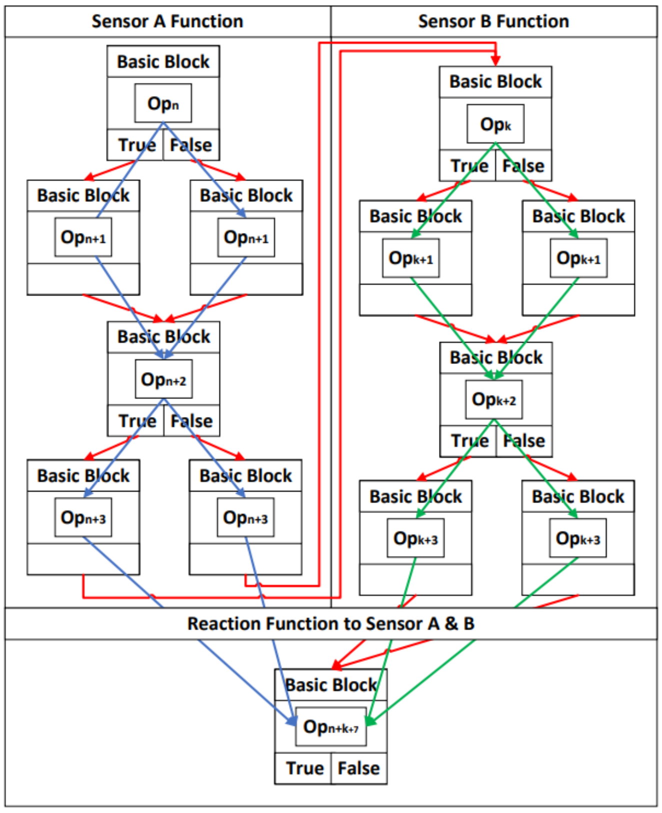


Fig. 11. Diagram of The Data Dependency Within Functions

The fig.12 shows that synchronization of multiple cores requires buffers and a lock mechanism to synchronize cores at each buffer. This mechanism can be implemented by an atomic instruction or binary semaphores.

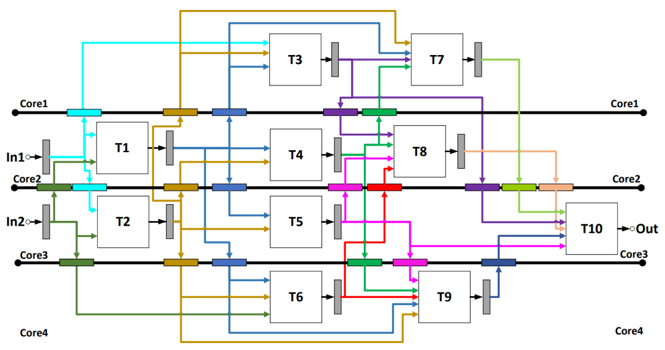


Fig. 12. Data Dependency Flow for the Parallel Execution Model

III. Cache Coherency

A. Coherency Protocols

The cache hierarchy significantly decreases the latency caused by the main memory. There are different cache hierarchies designs which consist of multiple levels of local or shared caches. The design of cache hierarchies in multi-core systems highly influences the latency related to fetching data from the main memory and data sharing synchronization across multiple cores.

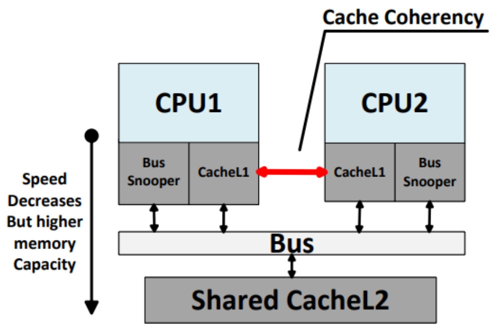


Fig. 13. Diagram Of Cache Coherency in Shared-Cache

Cache coherency is the problem which emerges when instances of the same data block are found in numerous local caches. The red line in fig.13 points to the coherency of shared data in the local caches. The shared data is synchronized by bus snoopers that implement cache coherence protocols which is the hardware mechanism for data synchronization in multiple local caches.

There are various cache coherence protocols. The choice of a particular cache coherence protocol can be influenced by the overall CPU design such as the need for minimizing power usage, design complexity, bus transactions, writebacks, etc. Additionally, different cache coherence protocols can be used for different hierarchy levels of the caches in the microprocessor. This hardware mechanism synchronizes data by monitoring the corresponding core's actions to its local cache lines, a bus connecting the local caches and other local caches' lines [3], [5].

The fig.14 illustrates the MSI protocol from which more advanced protocols are derived. The state diagram of the MSI

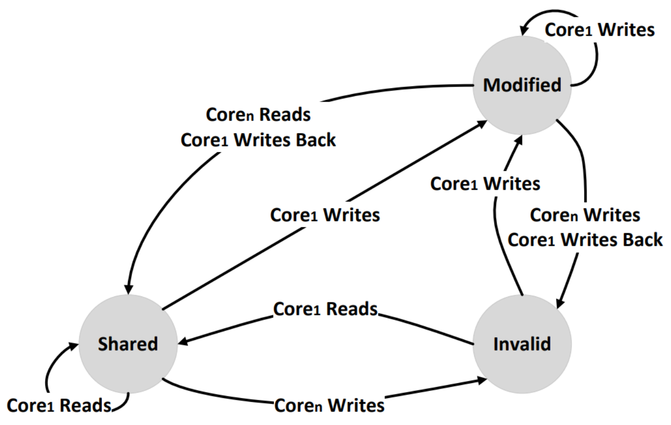


Fig. 14. State Diagram Of MSI protocol for the Core1 and the Same Data

protocol shown is performed for every cache line in every local cache. The state of the line to which data is assigned changes depending on the core's and other cores' actions that share and modify the data in the line. Cores can read data directly from another core's cache which has the most recent copy of the data. When a core requests to read the latest instance of data which is found in another core, the second core flushes the data to the bus which allows the first core to read the data and to be written back to the shared memory [3], [5].

B. False Sharing

The cache memory size is limited, therefore cache mapping is used to enable caches to point to every fragment of the main memory, and it allows to fetch a segment of a program (a line size) which is being executed at a time.

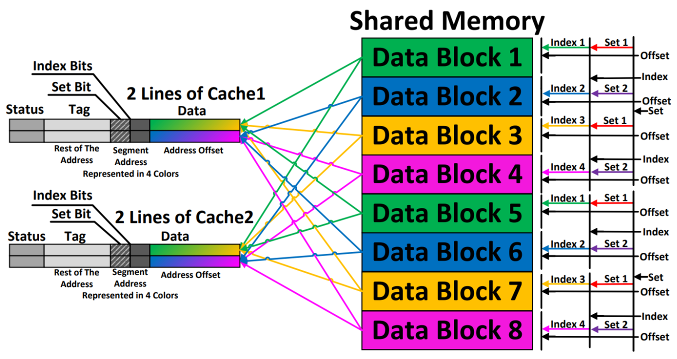


Fig. 15. Diagram Of Cache Line used for Memory Mapping

The fig.15 demonstrates the dynamics of the caching system. To limit non-organized bus usage between caches and the main memory, each cache line fetches a block of data. The tag is used to save the most significant bits of an address which are left after address offset and index bits. The index segment of the cache's lines allows associating each line with a segment of memory. It can be noticed in the fig.15 that the memory is divided into 4 colours indicating that there are 4 cache lines; however, one of the Index bits is used as a Set bit that allows each cache line to fetch two different segments represented in two colours in the diagram. Meanwhile, this structure allows for better management of limited-size caches because each line is associated with a specific cache line. Setting two virtual lines into one physical reduces expensive cache lines. It also brings forth the problem associated with false sharing between multiple local caches.

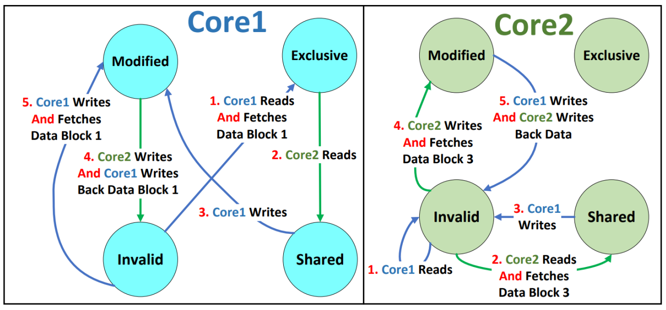

Core1
Modified
Exclusive
Invalid
Shared
5. Core1 Writes And Fetches Data Block 1
1. Core1 Reads And Fetches Data Block 1
4. Core2 Writes And Core1 Writes Back Data Block 1
2. Core2 Reads
3. Core1 Writes
Core2
Modified
Exclusive
Invalid
Shared
5. Core1 Writes And Core2 Writes Back Data
4. Core2 Writes And Fetches Data Block 3
3. Core1 Writes
1. Core1 Reads
2. Core2 Reads And Fetches Data Block 3

Fig. 16. State Diagram Of MESI protocol for Two Cores Showing False Sharing

The false sharing does not corrupt computation, but it is a huge performance loss. In the case when cache1 from the fig.15 fetches the first block and the third block is fetched by cache2. Both of the caches are going to use the first line which is associated with the green and orange memory segments.

However, the bus snoopers set the status of both lines as shared because the Tag segment of both lines in cache1 and cache2 are the same. As it can be realized from the fig.16, a lot of time is wasted for the unnecessary fetching and writing back data blocks.

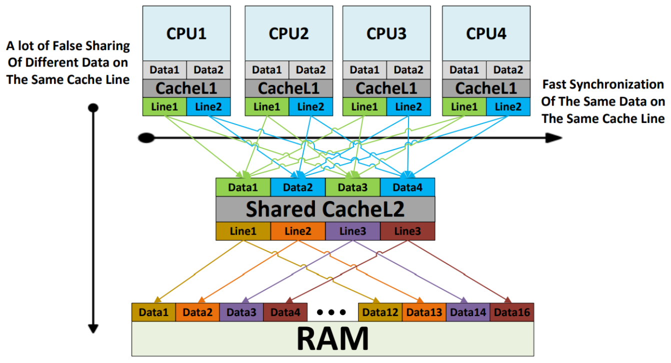


Fig. 17. State Diagram Of Four Cores Sharing One Cache

There are different multi-level cache design hierarchies for multi-core systems. Placing a single shared cache between multiple cores as in the fig.17, increases the synchronization of cache lines, but introduces huge false sharing between them. Meanwhile, separating cores into clusters shown in the fig.18, reduces the false sharing, but it also cuts down the synchronization performance.

This is a serious problem imposed by caching the main memory by multiple cores. The introduction of an additional

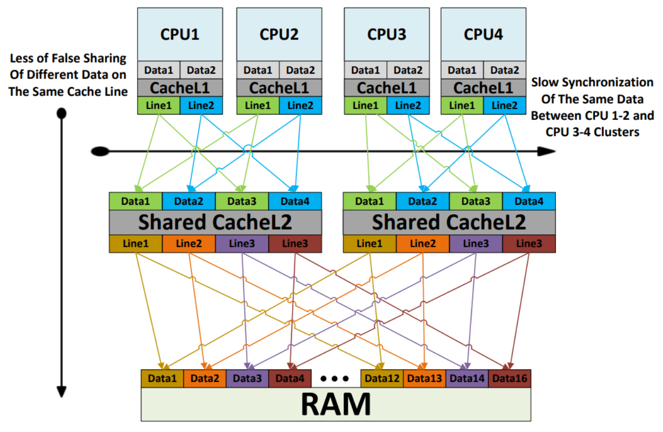

CPU1
CPU2
CPU3
CPU4
Data1
Data2
CacheL1
Line1
Line2
Less of False Sharing Of Different Data on The Same Cache Line
Slow Synchronization Of The Same Data Between CPU 1-2 and CPU 3-4 Clusters
Data3
Data4
Shared CacheL2
Line3
Data12
Data13
Data14
Data16
RAM

Fig. 18. Block diagram of two Dual-Core clusters

level of local caches as in fig.19 introduces a reflection barrier for false sharing. The barrier moves the phenomena shown in fig.16 to a lower level of the memory hierarchy which decreases the performance loss caused by false sharing, but it is never eliminated.

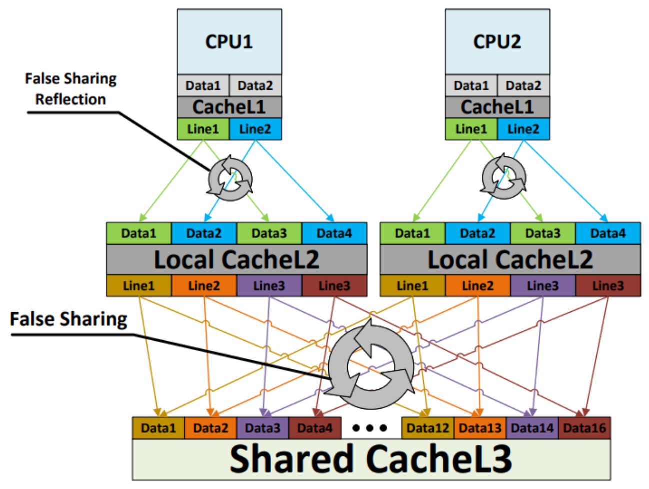


Fig. 19. Block diagram of Two Cores utilizing additional Local Cache

IV. SHARED-SEMAPHORED CACHE

The parallel execution models such as the fig.7 impose a problem for the GPUs and CPU's caches. In multi-threaded computations which require a rapid and frequent exchange of data across multiple cores which cannot be vectorized; the memory latencies suddenly cause a serious performance decrease as it was shown in the previous section.

The shared-semaphored cache is able to synchronize the processes at the moment when they exchange data. Semaphored-based hardware synchronization is a faster and simpler approach than the cache coherency protocols. The shared semaphored cache is implemented at the same level of memory hierarchy as the local caches as it is shown in the fig.20.

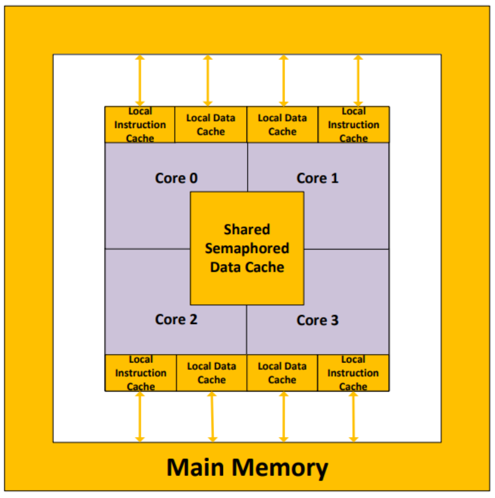


Fig. 20. Memory hierarchy model with shared-semaphored cache

A. *Semaphore Implementation*

The hardware implementation relies on the binary semaphores within the shared memory cells as it is shown in fig. 21, binary semaphores are able to synchronize multiple processes by making a process wait for another process to pass the data. The hardware semaphored data synchronization is a faster and simpler approach than any type of cache coherency protocols; additionally, it does not require new atomic instructions which might require modification of a CPU's pipeline, but a simply dedicated area of addresses to the semaphored cells. As it was already mentioned in the previous section, a dedicated compiler is required for multi-threaded mapping of a control program.

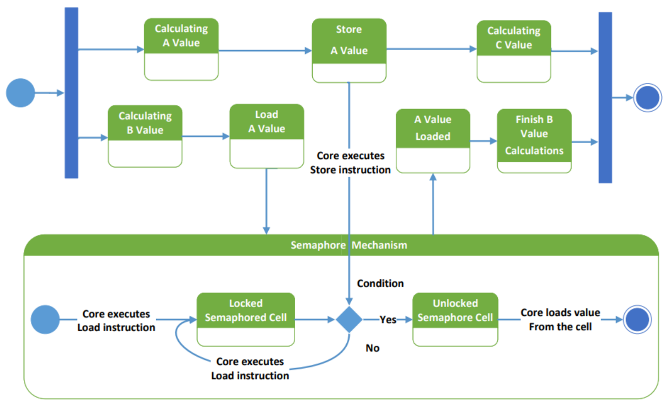


Fig. 21. UML model of process synchronization using semaphored cache

The diagram in fig.22 represents the hardware implementation of the shared-semaphored cache and how it is connected to the cores. Each core has a status register which is used to lock or unlocked a core according to the status of the given address of a semaphore cell.

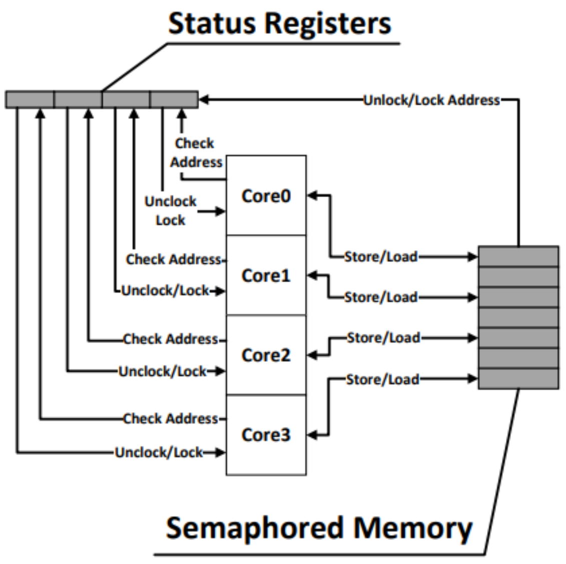


Fig. 22. Diagram of a Multi-Core System with the Shared-Semaphored Cache

B. Status Register

The status register shown in the figs.22 and 28 is an array of registers. Each register signifies a status of semaphore. Multiple decoders are used to associate each register with an address and decode multiple addresses at a time. An encoder allows to retrive an address from the status register to check the status of a semaphore.

C. Shared-Semaphored Cache

The diagram in fig.25 represents the hardware implementation of the shared-semaphored cache. It can be observed that each core is connected to one of the memory blocks via a separate input port. Additionally, each core has its own output port. This was done to speed up the write mechanism for the multiple cores. As it can be seen in fig.23, a single block of memory would radically slow down the synchronization of the cores.

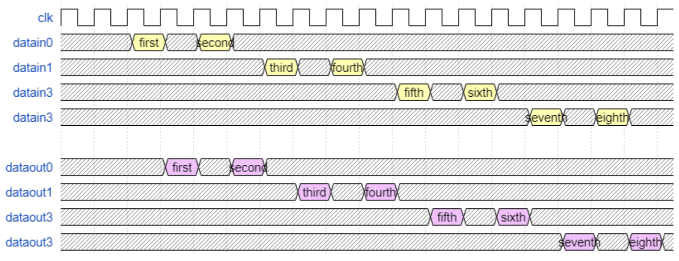


Fig. 23. WaveForm of Single Port Shared-Semaphored Cache

The fig.24 shows that adding additional blocks of memory into the shared-semaphored cache significantly speeds up the process of writing and reading data by multiple cores.

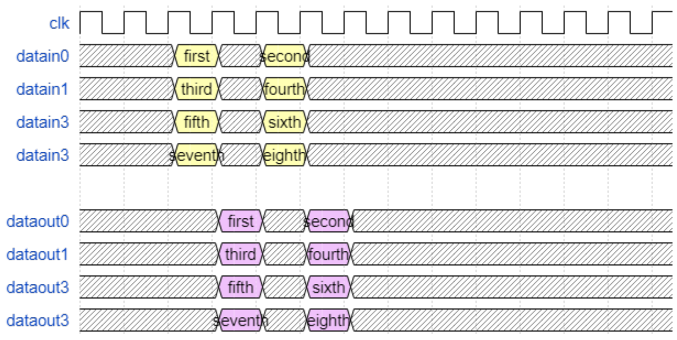


Fig. 24. WaveForm of Four Ports Shared-Semaphored Cache

The fig.25 represents that the multiplexers are used to select the correct memory block.

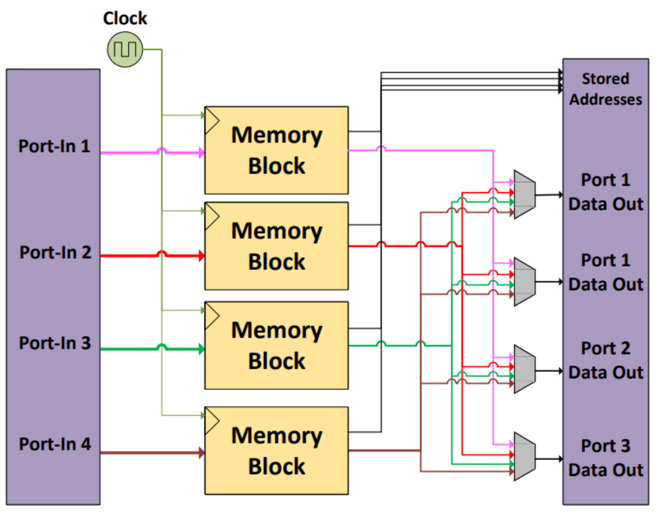


Fig. 25. Diagram of Shared-Semaphored Cache

The occupied bits in the memory cells are used to select the correct memory as it is seen in fig.26. Furthermore, the 8 bit registers are used to temporary store the address, so that the status registers manage to retrieve the address from the semaphored cache.

V. Memory Mapping

The CPU's pipeline is divided into multiple stages which are separated by pipeline registers. During each clock cycle, instructions are executed by previous pipeline stage and saved to the next pipeline stage in registers. The amount of stages depend on the architecture's implementation; however, every CPU has a stage in which data can be written to memory, GPIOs, peripherals or written back to general purpose registers. CPUs use memory mapping to physically map addresses of memory, GPIOs and peripherals as it is shown in the fig.27. The shared-semaphored cache's addresses are mapped this way; thus, the shared-semaphored cache can be universally implemented in different architectures, and it does not require modification of a core's pipeline.

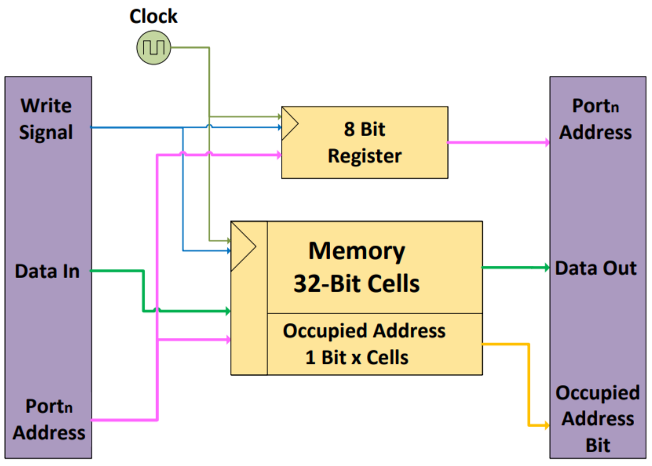


Fig. 26. Diagram of Memory Block for Each Port-In in the Shared-Semaphored Cache

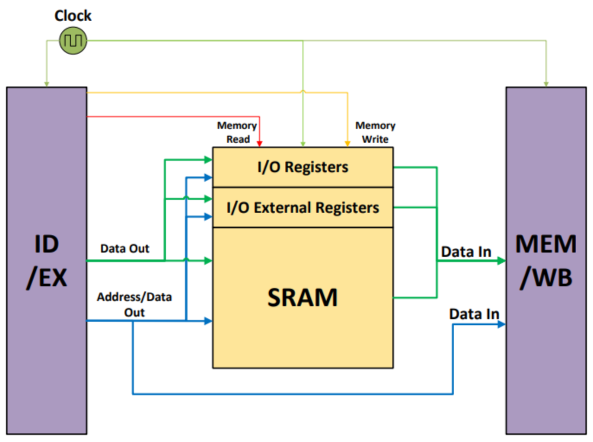


Fig. 27. Diagram Of Memory/WriteBack Pipeline Stage

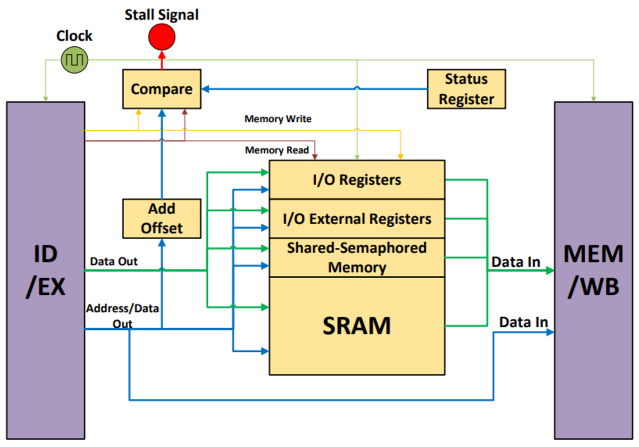


Fig. 28. Diagram of Memory Map

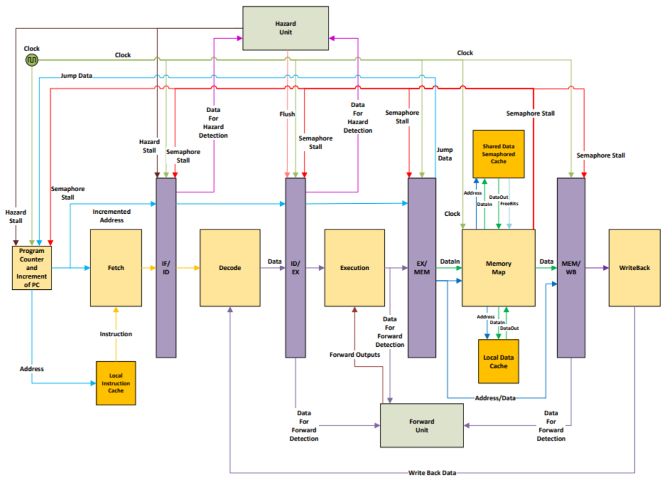


Fig. 29. Diagram Of CPU Pipeline With Mapped Shared-Semaphored Cache

A. Mapping Shared-Semaphored Cache

The diagram in fig.28 represents the memory mapping with the shared-semaphored cache. The status register is responsible for stalling a core to synchronize it with another core from which it is fetching the data.

The Stall CPU signal is connected to the pipeline registers which enables to stalls the whole pipeline as it is shown in the fig.29. The stalling of the pipeline works the same way when the hazard unit stalls half of the pipeline during fetching of data from the memory.

B. Semaphore Instructions

By Mapping semaphores, cores can write and fetch data from semaphores without any modification of the CPU pipeline. Furthermore, the constant values of shown in the **Semaphore Read Operation** and **Semaphore Write Operation** are stored in the instruction part of the memory which further decreases the synchronization performance loss because it does not rely on any data which needs to be fetched from the main memory.

Listing 5. Semaphore Read Operation

```
1: li $t1, SemaphoreOffSetAddr
2: lw $t2, SemaphoreNum($t1)
```

Listing 6. Semaphore Write Operation

```
1: li $t1, SemaphoreOffSetAddr
2: sw $t2, SemaphoreNum($t1)
```

VI. Multi-Thread Mapping

Mapping to a multi-threaded program can be achieved by representing the whole (sequentialy given) input program into a data dependency graph as it ilustrated in fig.30. Intermediate representation instructions are represented as Nodes and their dependency as edges with blue arrows. A custom written optimizer for the LLVM compiler was created to map multiple threads. Additionally, other optimizers (inherited from the main implementation of LLVM) are also used to eliminate dead code, unroll loops and combine redundant instructions before the optimizer for mapping multiple threads is called.

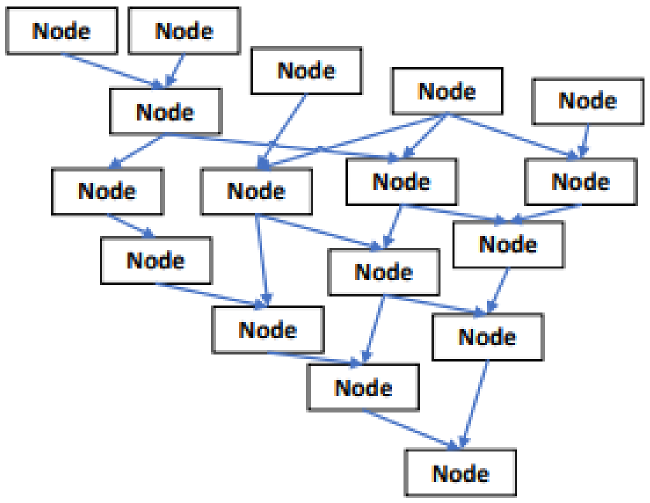


Fig. 30. An exemplary Data Dependency Graph of a program

The optimizer treats each Node input as the beginning of a thread. It places Input Nodes into a List representing a Thread as it is seen in the fig.31.

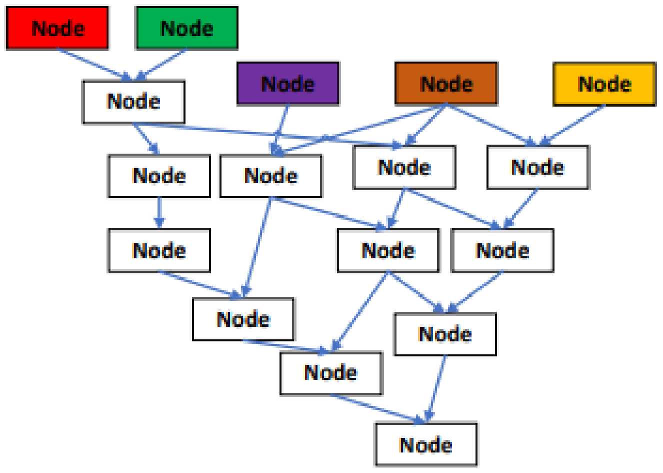


Fig. 31. Initial threads assignment to independent nodes

At moment when the optimizer notices that the result operand of the current node is used as second source operand in the next node, it places a temporary node between the nodes as it can be observed in the fig.32.

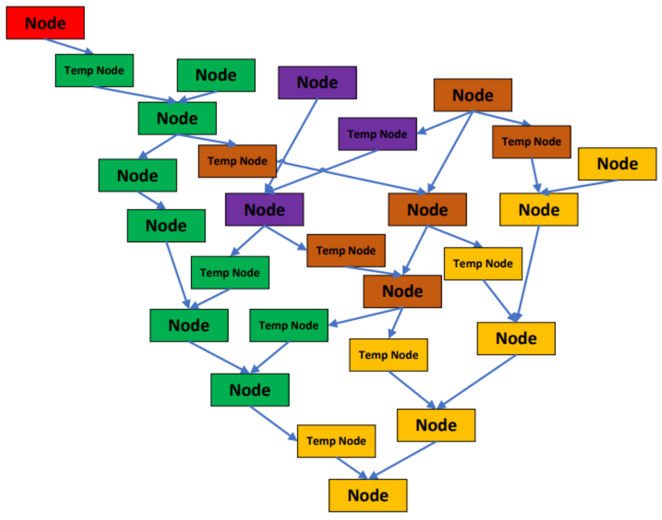


Fig. 32. Program with completely mapped threads

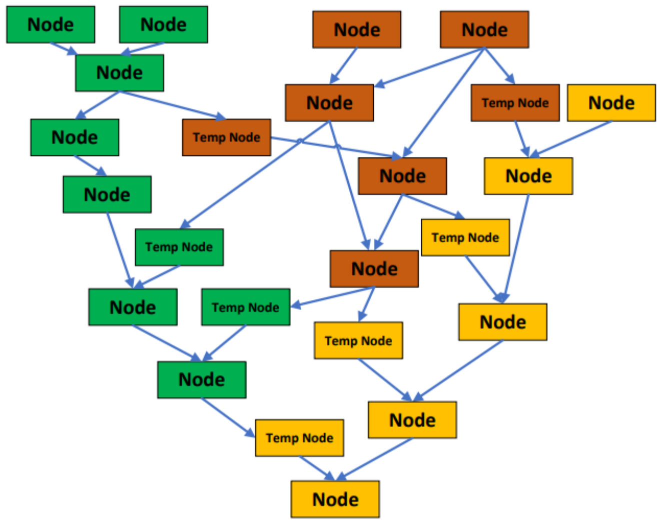


Fig. 33. Threads merging step: Red to Green and Purple to Brown Thread

When all the instructions are placed into multiple threads, the optimizer merges the shortest threads that share edges and removes temporary nodes as it is shown in fig.33. The optimizer merges threads until the specified number of threads remain and permanently places Synchronization Nodes as it is demonstrated in the fig.34.

The Assembly code is generated from each Thread List. The Synchronization Nodes which are placed between different threads are generated into store and load operations of semaphores. The colour of Synchronization Nodes generates into fetching of data from semaphores; meanwhile, the edges to the previous Node pass data to semaphores.

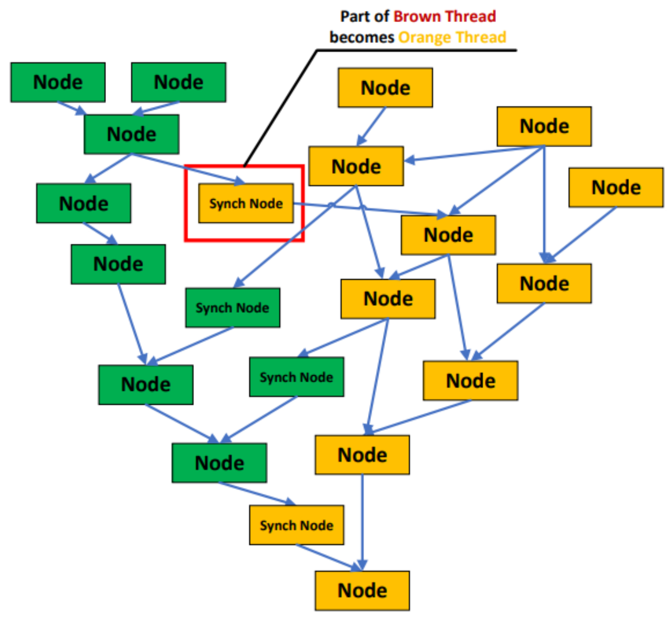
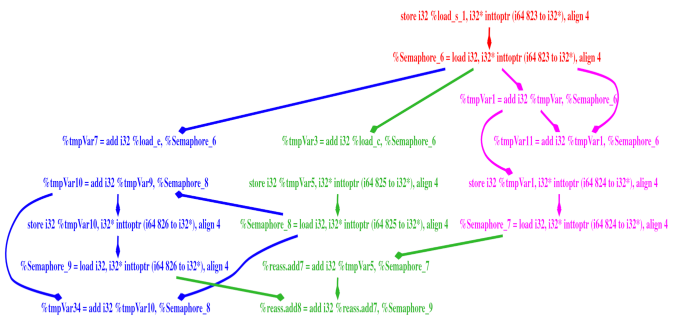


Fig. 34. Threads merging step: Brown Thread to Orange Thread

VII. Results

The implementation of the custom optimizer for the LLVM Compiler has allowed mapping multiple threads to execute in parallel as it is given in fig.12. Additionally, it has enabled an automatic generation of multi-threaded assembly code utilizing semaphored synchronization mechanisms.



```
store i32 %load_s_1, i32* inttoptr (i64 823 to i32*), align 4
%Semaphore_6 = load i32, i32* inttoptr (i64 823 to i32*), align 4
%tmpVar1 = add i32 %tmpVar, %Semaphore_6
%tmpVar7 = add i32 %load_e, %Semaphore_6
%tmpVar3 = add i32 %load_c, %Semaphore_6
%tmpVar11 = add i32 %tmpVar1, %Semaphore_6
%tmpVar10 = add i32 %tmpVar9, %Semaphore_8
store i32 %tmpVar5, i32* inttoptr (i64 825 to i32*), align 4
store i32 %tmpVar1, i32* inttoptr (i64 824 to i32*), align 4
store i32 %tmpVar10, i32* inttoptr (i64 826 to i32*), align 4
%Semaphore_8 = load i32, i32* inttoptr (i64 825 to i32*), align 4
%Semaphore_7 = load i32, i32* inttoptr (i64 824 to i32*), align 4
%Semaphore_9 = load i32, i32* inttoptr (i64 826 to i32*), align 4
%reass.add7 = add i32 %tmpVar5, %Semaphore_7
%tmpVar34 = add i32 %tmpVar10, %Semaphore_8
%reass.add8 = add i32 %reass.add7, %Semaphore_9
```

Fig. 35. Data Dependency Graph of Multi-Threaded Synchronization

The fig.35 shows four cores represented in different colours, synchronizing the data-dependent instructions using semaphore operation instructions.

The fig.36 and fig.37 simulate the synchronization of two cores. It can be spotted that it takes 6 Clock Cycles For Core3's register V0's value to Synchronize with Core0's register At's value. The synchronization takes 4 Clock Cycles after the Core3 requests the value. This is possible due to the fact that cores bypass the traditional hierarchy of local and shared caches for which the multi-core synchronization is a performance and design challenge as it was demonstrated in the article. Mapping the shared-semaphored cache in a multi-core system permits to use of write and read operations as synchronization and data-sharing instructions with no modification of the pipeline.

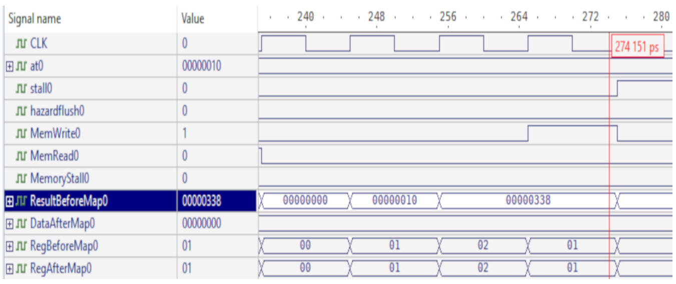


Fig. 36. WaveForm of the simulated shared-semaphore cache is shown when Core0 Writes a Data to a Semaphore

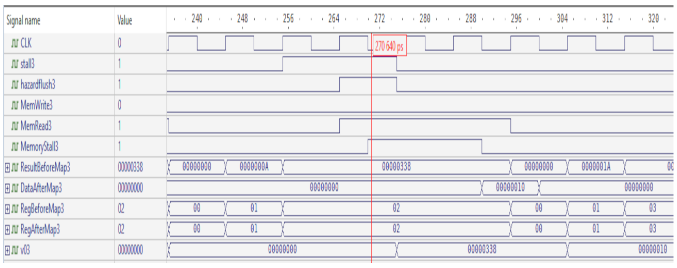


Fig. 37. WaveForm of the simulated shared-semaphore cache is shown when Core3 Stalls for the Core0 and Gets the Data from the Semaphore

VIII. Conclusion

The paper presents an approach to multi-threaded sequential program execution focusing on algorithms in which each data set is computed in parallel by a completely different computation task. This is done by implementing a custom optimizer for LLVM Compiler which maps intermediate representation instructions to multiple threads and automatically places synchronization operation instructions between data-dependent sequences assigned to two different threads. The generated multi-threaded assembly code is dedicated to a multi-core system. The multi-threaded parallelly executed program synchronizes and shares data via automatically mapped binary semaphores and omits the usage of the caches.

References

[1] Morris Mano, Charles R. Kime and Tom Martin *Logic and Computer Design Fundamentals* Pearson; 5th edition, 2015.
[2] David A. Patterson and John L. Hennessy *Computer Organization and Design.* Morgan Kaufmann; 4th edition, 2011.
[3] William Stallings *Computer Organization and Architecture: Designing for Performance.* Pearson; 10th edition, 2015.
[4] David A. Patterson and John L. Hennessy: *Computer Architecture: A Quantitative Approach.* Elsevier, Oxford, UK; 6th edition, 2017.
[5] Michael L. Scott: *Share-Memory Synchronization.* Morgan & Claypool Publishers, 2013
[6] Nacke, Kai *Learn LLVM 12.* Packt Publishing, 2021.
[7] Keith D. Cooper, Linda Torczon *Engineering Compiler.* Morgan Kaufmann; 2nd edition, 2011.
[8] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman *Compilers: Principles, techniques & tools* Addison Wesley; 2nd edition, 2006.
[9] Vivek Sagdeo *The Complete Verilog* Springer; 1998th edition, 2007.

[10] P. Coussy, D. D. Gajski, M. Meredith and A. Takach *An Introduction to High-Level Synthesis* IEEE Design & Test of Computers, vol. 26, no. 4, pp. 8-17, July-Aug. 2009, https://doi.org/10.1109/MDT.2009.69

[11] Robert Love *Linux Kernel Development* Addison-Wesley Professional; 3rd edition, 2010.

[12] Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, Philippe Gerum *Building Embedded Linux Systems* O'Reilly Media; 2nd edition, 2008.