An Efficient Barrier Implementation for OpenMP-like Parallelism on the Intel SCC

Hayder Al-Khalissi, Syed Abbas Ali Shah, and Mladen Berekovic
Chair of Chip-Design for Embedded TU Braunschweig, Germany
{alkhalissi, shah, berekovic}@c3e.cs.tu-bs.de

Abstract—This paper proposes an effective barrier synchronization implementations for shared memory-based parallel programming models (e.g. OpenMP) on the Intel SCC non-cache-coherent platform. Barrier synchronization primitives are key components of these programming models to coordinate the parallel threads. Therefore, we need an efficient implementation of the underlying synchronization algorithms to allow high-level barrier constructs for better performance. In particular, we present an efficient evaluation method to determine the overhead associated with integration of barrier algorithms that is required for OpenMP runtime libraries. We validate several implementation variants that efficiently use the network topology and SCC-specific hardware. Our experimental results for different Microbenchmarks show significant performance improvement up to 98% for 48 cores.

Index Terms—Barrier synchronization, System-on-Chip, Many-cores, OpenMP model, Performance Evaluation.

I. INTRODUCTION

Systems-on-chip (SoCs) nominate today the best way to cover an increasing number of cores available in a single chip. With systems featuring 100 cores and more being on the market and core numbers steadily increasing – making the shift from the former multi-core to today’s many-core era. They typically provide higher performance, lower power consumption, but more complexity than multi-core technology. As a consequence, it is no longer possible to ignore the challenges caused by the convergence of software and hardware development [1]. In particular, effective programming models face new challenges due to the effect of scaling, which is a huge concern for developers. The software stack is nowadays responsible for making effective use of the systems' tremendous peak performance. This, of course, requires to employ all processors for most of the time. Therefore, it is necessary to provide efficiently coordinated execution of a multi-threaded application on the system cores.

Parallelism within a many-core system is most commonly exploited by using either a Message Passing Interface (MPI) approach or a Shared memory-based programming model using OpenMP. The purpose of these programming models is to extend the source language (normally C) to include multi-core features in a scalable way.

Recently, OpenMP [2] has emerged as a de-facto standard for shared memory programming, since it provides very simple means to expose parallelism in a standard C (or C++, or Fortran) application, based on code annotations (compiler directives). This appealing ease of use has recently led to the flourishing of a number of OpenMP implementations for embedded Multiprocessor systems-on-chips (MPSoCs) [3], [4], [5], [6], [7]. OpenMP (and most related shared memory-based programming models) relies on a fork/join execution model, which leverages a barrier construct to synchronize the threads that share the work. Barriers –implicit or explicit – are central constructs to the OpenMP execution model and to any shared memory parallel program. Barrier synchronization overhead has been recognized as an important source of the performance degradation in the execution of parallel programs [8], [9], [10], [11].

The longer-term goal of our project is providing an efficient implementation of the OpenMP programming model to MPSoCs. In this paper, we cover a fair share of implementations of OpenMP-like barrier algorithms for the Single-Chip Cloud Computer (SCC) in order to determine which barrier algorithm is most appropriate for a given scenario. Therefore, we need to gain insight into the behavior of different barrier approaches in the OpenMP context.

Intel’s SCC platform [12] is dedicated to exploring the future of many-core computing. It is a research architecture resembling a small cluster or “cloud” of computers. The SCC architecture has 48 independent Pentium P54C cores, each with 16kB data and program caches and 256kB L2 cache. The cores are organized as 24 dual-core tiles connected via a low-latency mesh network. The SCC tiles structure are coordinated in a 6 x 4 grid and further decomposed into distinct voltage and frequency domains. Each tile connects to a router and contains two cores, a Mesh Interface Unit (MIU), and a pair of test-and-set registers for realizing atomic access. The SCC does not offer cache coherency between the cores, but rather employs special 16kB-sized Message Passing Buffer (MPB) for fast, hardware-assisted explicit message passing between cores.

The contribution of this paper is as follows. We have implemented a several approaches to barrier synchronization and integrated them into the SCC OpenMP runtime library. Second, we have investigated a number of techniques for reducing the barrier overhead by leveraging SCC-specific hardware support for synchronization and its explicitly-managed portion of the memory hierarchy (i.e., MPB), and pattern communication analysis similar to that performed for message-passing machines. Finally, we have studied the barrier synchronization using micro-benchmarks to track a number of important methodological challenges on different frequency scaling. Our
experimental results section provides a detailed evaluation of the performance achieved by different approaches and shows benefits and drawbacks of individual approaches as well as significant performance improvements for the optimal solutions. The remainder of this paper is structured as follows: An overview of different barrier algorithms is given in Section III. Methodology and micro-benchmark implementations are discussed in Section IV, while the experimental results are evaluated in Section V. Finally, our conclusion and future works are given in Section VI.

II. OPENMP MODEL AND TARGET BARRIER

The cores of SCC are single-threaded. Therefore, in the rest of paper we consider cores and threads to be equivalent, as we do not “oversubscribe” cores but only assign one thread per core. OpenMP [13] employs the fork/join programming model that is easy and flexible to handle sequential and parallel parts of an application. The program executes sequentially within a single thread, referred to as the Master thread, until it encounters a #pragma omp parallel directive. Here, execution forks into a multitude of threads by assigning (forking) computation to a number of worker threads (slaves). As a result, a parallel region is created. At the end of the parallel construct the master waits for all slave threads to complete (join) before continuing execution. Then only the Master thread resumes execution. A barrier is used to control and ensure all slave threads have completed before the master thread can continue. Control synchronization reduces parallelism in a program by forcing threads to wait until a certain condition holds. Our implementation of OpenMP model is based on the work of Marongiu et al.[7] that relies on a custom micro-kernel code [7], [14] executed by every core at start-up. Master and slave threads execute different code based on their core IDs. After system initialization, the Master core jumps to execution of the parallel program’s master thread, while the slaves wait for activation (fork). When the Master encounters a parallel region, it invokes the runtime system, points the slaves to the parallel function, and triggers execution. At the end of the parallel region, a global barrier synchronization step is performed. Therefore, the fork/join model imposes two synchronization events per parallel loop as mentioned before. Consequently, the costs of barrier for fork/join model can be high, especially when the application has nested loops such as parallel inner and sequential outer loops.

III. BARRIER SYNCHRONIZATION

Several implementations of OpenMP for MPSoCs have adopted a centralized shared barrier [3], [4], [5]. It relies on shared entry and exit counters, which are atomically updated through lock-protected write operations. The counters are used to hold the number of threads that have reached a barrier. The threads wait for the last thread reaching the barrier, subsequently signalling all other threads of its arrival by setting the flag. This algorithms yields bad performance as the access to the counter is serialized [14]. Additionally, in non-cache-coherent systems such as SCC, the updates to barrier structures (e.g., control flags, counters) in shared memory must explicitly be kept consistent. The SCC platform exploits the message-passing programming model. One well-known library supporting this model is RCCE [15], featuring a simple barrier algorithm based on a local put/remote get approach. It uses flags for synchronization by allocating them in that core’s MPB, which has initiated an update. The Master core uses remote polling on the release flag repeatedly for all following cores. In this section, several different barrier algorithms are examined as part of an effort to investigate ways in which OpenMP and its implementations may scale to large thread counts. We would first like to give an overview of software implementation for barrier algorithms before presenting a brief description of the various implementations.

Barrier synchronization typically involves three phases. First, A thread posts its entry into the barrier, then waits for the last thread to arrive at the barrier, and finally receives a notification signal (from the master thread) releasing all threads from the barrier. We therefore implemented those phases as separate functions in order to analyze the overhead individually for every phase. We exploit a Master/Slave scheme to implement several algorithms on the SCC. This approach is accomplished in two phases, the Entry or Gather phase and the Signal or Release phase. Each Slave signals its entry into the barrier by using a Slave_Enter() function in the entry phase; it then waits for the release signal. Gather and release phases are controlled by the Master thread through the corresponding Wait() and Release() functions. These functions have been implemented individually for being able to execute additional housekeeping functions before releasing the slaves.

A. Shared Master-Slave Algorithms (S-MSB)

The S-MSB is linear barrier implementation and presented in our previous work [16]. This scheme uses a local get/remote put approach as depicted in Figure 1 (where threads are represented by a circle, time flows downward, and memory allocation is symbolized by a square shape). Two flag arrays are allocated in different memory portions: the Master flags are allocated in the Master’s MPB and Slave flags are distributed over the Slave’s MPBs. In this approach, the Master core is responsible for accepting all the gather signals at the barrier and issuing release signals. The S-MSB algorithm removes contention for shared counters and congestion by allocating
variable. To avoid a possible issue for the case that one thread enters a barrier with a polarity that is opposite to the polarity of another thread, we start the barrier using a default initial polarity value. Thus, all threads enter barrier with same polarity and this issue does not occur. Furthermore, the Master thread has no information about the slaves received the release signal. The Master only knows that all threads have received previous release signal once a thread enters the barrier again. As shown in Figure 3, CPB allocates each of the slave’s poll flag onto their local memory and uses the chain mechanism to broadcast the release signal and gather the entry signals. This approach uses a chain scheme to reduce the overhead of publishing the release signal in the Signal phase and avoid contention access in a single shared variable approach. In the chain mechanism, each thread receives the entry signal from its next higher-numbered neighbor and then sends its entry signal to the next lower-numbered neighbor. With each core having one predecessor and one successor, this effectively creates a chain topology.

C. LUT Barrier Algorithm (LUTB)

In the SCC, every core has a lookup table (LUT) with 256 entries used for physical-to-physical address mapping [12]. It is part of the configuration registers system that is mapped by a LUT entry also. The LUT is a shareable between all the cores and mostly used by operating system, but may be used also on application level. It is possible to change the contents of the LUT dynamically without causing problems to the core’s memory management. This raises the idea of using the LUT entries for barrier synchronization. This is granted: not only can every core access its own or any other core’s LUT, but also LUT entries are mapped by using mmap() in un-cached (UC) mode. In addition, by using LUTs we avoid the issues of ensuring a consistent view of MPB. The LUTs are located very close to the core and, thus, we avoid the extra overhead of access local memories and off-die registers (e.g, atomic Increment Counters and Global Interrupt Registers). Therefore, LUTB is a new implementation that uses an one LUT entry in every core to track notify and release signals. As a consequence, this algorithm requires a single flag (LUT entry) array being shared between the cores. In order to notify the Master thread that it has arrived, the Slave thread changes the state of its flag in an array to positive. When the Master thread has verified that this transition has occurred and counts the number of threads have arrived, it releases the Slave by
relying its flag to negative transition. The Slave threads wait until the negative transition occurs. Each core can cause only one of the two transition changes. In this approach, every thread is responsible on reinitialize its flag that is allocated in its own, local LUT.

D. Polarity-LUT Barrier Algorithm (P-LUTB)

It follows the LUTB approach, but uses a single shared entry in LUT to release all slaves instead of relaying its flag to negative transition. By using a single shared variable to convey the release information, we could avoid extra overhead for updating all flags of participating slaves in the barrier. We used a single LUT entry with a private variable (similar to exit polarity mechanism) to handle the reinitialization problem.

E. LUT Polarity Barrier Algorithm (LUTPB)

This approach uses LUT entries and a single shared variable (flag) that is allocated in Master MPB. The Entry phase uses a similar entry mechanism as LUTB and employs the Signal phase schema in PLUTB for terminating the slaves. Moreover, this algorithm provides two access modes, UC mode to access LUT entry and MPBT mode to access a single shared variable.

F. Chain LUT Polarity Barrier Algorithm (C-LUTB)

C-LUTB is a linear algorithm using chain mechanism for reducing the overhead of gathering the notify signals in the Entry phase of P-LUTB. To implement this algorithm we need to use one entry in each core. In the signal phase, the master thread waits to receive the notification signal from its higher neighbor and exchanges the LUT entry of the participant’s last thread. Because of the LUT entry of the last thread is not used (no further neighbor in the chain), we use this entry for releasing the slaves. All slaves wait for exchanging the state of the last thread’s LUT entry that indicates a release signal.

G. Binary-Tree LUT Polarity Barrier Algorithm (BT-LUTPB)

Yew et al [18] proposed the tree algorithm in order to increase the performance of the Central Barrier algorithm by reducing its associated contention. The tree algorithm exploits logarithmic mapping. It is somewhat similar to the S-MSB scheme. Each phase of the tree barrier approach inherently requires more computation than a linear algorithm, because each thread must calculate its leaves or children. In this approach, each parent node has either two leaves or one leaf. It allows for a single thread to synchronize with several neighbors and also allows to propagate several signals to others. Traditionally, the implementation of binary tree is very complex because of data structure, initialization, and determination of parent and children nodes.

In our implementation of Binary Tree-based barrier synchronization on SCC, we avoid all these complexities by initializing and allocating an array of flags in every node. The BT-LUTPB approach uses the binary tree scheme in the Entry phase and allocates the flags in LUT. Firstly, we allocate every slave poll flag in the slaves’ local memory and also every core is responsible for initializing its own flag therein. Each thread then determines its children according to its assigned node id without the need for knowing its parent. To notifying the parent thread by its children upon their arrival at the barrier, each child thread exchanges the state of its flag and waits for the release signal from its parent. The parent waits for all notifications based on individual check-in flag(s) of its child(ren) before updating its own flag; this propagates through the tree until the topmost parent (Master / root) receives the notification. During Signal phase, the Master thread releases all slaves by updating a shared release value similar to the release mechanism in P-LUTB algorithm. The algorithm therefore needs at most \( \log_2(\text{number of threads}) \) rounds to complete the barrier entry process.

IV. METHODOLOGY AND MICRO-BENCHMARKS

Culler et al. [19] proposed LogP-model that accurately predict performance of a large and complex program on active-message based systems. This model summarizes the performance of a platform in four parameters: the network latency \( L \), the overhead \( O \), gap \( g \) represents the minimum time interval between subsequent messages, and the processor number \( P \). We use this model to represent the overhead of barrier algorithms as depicted in Figure 4. The parameters of this model are listed in the Table I. In most previous studies a simple micro-benchmark used to measure the time only in the Master thread. They measured the latency of gathering and releasing the participating processors as illustrated in the Figure 4 (red bar), but they overlooked the overhead by slave threads in the phases of the barrier in their works. However, we can calculate overhead of LogP-model parameters (Table I) by measuring the time in two groups which are Master Overhead (MO) and Average Slave Overhead (ASO). The MO represents the cost for different approaches of performing barrier synchronization in the Master thread, including the two barrier phases.

### TABLE I: The parameters of the barrier performance model

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O_g )</td>
<td>Core overhead to read the flags in Master thread.</td>
</tr>
<tr>
<td>( L_g )</td>
<td>Communication time to gather flags in Master thread.</td>
</tr>
<tr>
<td>( O_r )</td>
<td>Core overhead to check the status of flags.</td>
</tr>
<tr>
<td>( O_s )</td>
<td>Core overhead to update the flags in Master thread.</td>
</tr>
<tr>
<td>( L_r )</td>
<td>Communication time to update flags in Slave thread.</td>
</tr>
<tr>
<td>( L_s )</td>
<td>Communication time to read flags in Slave thread.</td>
</tr>
<tr>
<td>( O_c )</td>
<td>Core overhead to get the new flag signal in Slave thread.</td>
</tr>
</tbody>
</table>
The influence of the overhead of inserting the new value of flag (yellow bar in the Figure 4) that is included in ASO sight. The ASO collects the overhead per participant (excluding the Master) by summing up all slaves’ overhead divided by their number. This procedure allows a direct comparison of barrier costs on Master and Slave side and therefore determining the quality of the evaluated algorithms with regard to execution time and parallel speed-up.

EPCC [20] is a simple micro-benchmark used for measuring barrier performance; however, as it does not regard the implicit barriers in OpenMP directives, we consider it insufficient for our cause. Kumar et al [21] reported on overhead of the barrier under three conditions: random load, impact of load imbalance, and effect on network of synchronization operation. In addition, G.H. et al [22] proposed to add a certain delay in critical section. In this work, we implemented two kinds of micro-benchmarks based on the above scenarios: Pure Overhead and Impact of static load. We have reprogrammed the existing micro-benchmarks to analyze the Impact of memory access mode and also studied the Impact of frequency scaling on the performance of barrier algorithms. In the Pure Overhead micro-benchmark, barrier code is executed only on the platform without any communication between cores takes place. This allows estimating how the algorithm scales with increasing synchronization traffic only [16]. For computing the average pure overhead of barrier, we use the following equations:

$$\text{Avg}_M = \frac{\text{TotalBarrierTime}}{(\text{No.ofIterations} - \text{No.ofIgnores})}$$  \hspace{1cm} (1)

$$\text{Avg}_S = \sum_{i \neq \text{Master}} (\text{AvgBarrierTime})$$ \hspace{1cm} (2)

$$O_P(\mu s) = \begin{cases} \text{Avg}_M, & \text{if } T_{id}s \text{ Master} \\ \text{Avg}_S, & \text{else} \end{cases}$$ \hspace{1cm} (3)

Where:
1) $\text{Avg}_M$: is the average time of Master thread including Wait() and Release() functions.
2) $\text{Avg}_S$: is the average time of Slave threads including only the Slave_Enter() function.
3) $O_P(\mu s)$: is the average execution time of Master or Slaves in $\mu$ seconds.
4) $\text{No.ofT_{ids}}$: is the number of running threads (participants)

Figure 5 depicts the implementation of the micro-benchmark depending on added delay: we add delay in the master thread between the Entry and Signal phases to avoid interference with the barrier’s next iteration. Here, we implement the static loads by adding static delay before the critical section of the slave threads. This stagger their arrival at the barrier point as well as inside the critical section in the Master thread’s release phase. The static case is set up in a way that a core arrives with maximum efficiency. Thus, these micro-benchmark measure the overhead of the slaves’ arrival at the barrier. The average overhead time of the load (static) impact are computed based on equations (1),(2):

$$\text{Avg}_M = \frac{\text{TotalDelayTime(Static)}}{(\text{No.ofThreads} - \text{No.ofIgnores})}$$ \hspace{1cm} (4)

$$\text{Avg}_S = \sum_{i \neq \text{Master}} \frac{\text{TotalDelayTime(Static)}}{(\text{No.ofIterations} - \text{No.ofIgnores})}$$ \hspace{1cm} (5)

$$O_L(\mu s) = \begin{cases} \text{Avg}_M - \text{Avg}_D, & \text{if } T_{id}s \text{ Master} \\ \text{Avg}_S - \text{Avg}_D, & \text{else} \end{cases}$$ \hspace{1cm} (6)

Where:
1) $\text{Avg}_M$: is the average time of Delay (static) in Master thread before entering the barrier.
2) $\text{Avg}_S$: is the average time of Delay (static) for Slave threads before entering the barrier.
3) $O_L(\mu s)$: is the average load impact for Master or Slaves in $\mu$ seconds.

The SCC features two types of accesses to its MPB, cached (MPBT) and un-cached (UC). In MPBT mode, data caches in the (L1) cache only and issues write ordering after filling the write-combine buffer (WCB). While in UC mode, data reads are not cached and writes are directly issued to the network. The default implementation of barrier algorithms which have access to MPB memory, it is based on MPBT mode. As explained in our previous work [16], the UC mode has impact in the overhead reduction more than 41%. Because of UC mode avoids extra overhead for invalidating MPBT lines before read and write operations, as well as the cycles required to flush the WCB. We measure the influence of memory mode access on the performance of Pure Overhead and Static Load micro-benchmarks for algorithms which implemented based on MPB.

Finally, the different frequency of processor and network (router) has impact on the contention behaviour. We measure the impact of changes frequency on the previous micro-benchmarks. In addition, the analysis includes the effect of varying the NoC topologies as the number of cores increases.

V. PERFORMANCE EVALUATION

The experimental results are generated using the default SCC settings, which are standard LUT entries, 533 MHz tile frequency, 800 MHz mesh and DRAM frequency for all micro-benchmarks except Impact of Frequency scaling, where the influence of the different frequencies is regarded separately. The experiments are conducted using sccKit 1.4.2.2
running a custom version of sccLinux based on the 2.6.32.24-generic kernel. For timing analysis, RDTSCK (Read Time Stamp Counter) instructions [23] are inserted before and after the barrier algorithm. We execute each barrier algorithm 100,000 times, determining the mean execution time in µs. The microbenchmark results in this section are mainly divided into two groups, MO and ASO.

A. Pure Overhead

Figure 6(a) shows the average overhead of the Master thread that includes time to wait for the slaves and to release them on the SCC. While Figure 6(b) shows the average overhead to notify the Master thread by the slaves and waiting time for the exit (release) signals. As depicted in Figure 6, C-LUTB and CPB algorithms on the MO and ASO achieve more than 97% overhead reduction for 48 threads. BT-LUTPB scheme achieves about 96% performance optimization compared to the baseline S-MSB across threads < 48. For the ASO, the average overhead of the CPB approach is about 26 times slower than S-MSB, because there is no contention to notify the Master and to access a shared signal variable. It is important to note that extra overhead is added to the BT-LUTPB as a direct result of the x/y routing effect on the resource access behavior. In our experiments, C-LUTB is always the fastest barrier, making it the ideal candidate to perform barrier synchronization regardless of increasing number of cores. For more than two cores, C-LUT shows significantly higher ASO than others.

B. Impact of Static Load

We measure the static load on barrier algorithms which affects performance. Figure 7 depicts two curves corresponding to various barriers with different number of cores. Figure 7(a) shows CPB and C-LUTB algorithms benefit from adding fixed delay to the barrier while others show extra overhead. Evidently, adding fixed delay to barrier improves the performance of CPB and C-LUTB algorithms as compared to Figure 6(a). Also Figure 7(b) shows C-LUTB performs better because of less time required by the slaves to update their poll flags allocated in their LUT. Consequently, this algorithm achieves good performance when increasing number of cores.

C. Impact of Memory Access Mode

To study the impact of memory access mode, we re-implemented the algorithms (e.g. S-MSB, CPB, and LUTPB) that have access to local memory (MPB). Figure 6 and Figure 7 show results from this experiment. In this experiment, the UC mode reduces the overhead on MPB-based algorithms, also it significantly improves the CPB algorithm as shown in Figure 6(a) and Figure 7(a). The CPB-UC apporch shows worse results by more than 96% compared to its overhead in the previous experiments. Figure 6(b) and Figure 7(b) show that CPB-UC reduces the overhead by 19.9% approximately for 48 threads. Namely, implementing UC mode contributes on reducing the overhead, because of the setting does not require to invalidate L1 cache lines before accessing the UC-mapped memory and flushing the WCB after write operations. In addition, there is a slight overhead difference between CPB-UC and C-LUTB; we also note that there is no impact on the overhead with the increasing number of cores.
D. Impact of Frequency Scaling

As part of our work, we investigate the impact of frequency scaling on the barrier algorithms for each micro-benchmark. In this section, we studied the influence of contention on the barrier algorithms with different processor and network frequencies. Therefore, we experiment the pure overhead and Static load micro-benchmarks (included algorithms with UC mode optimization) with the setting Tile800_Mesh800_DDR800 and Tile800_Mesh1600_DDR800 respectively as illustrated in Figures 8, 9, 10, and 11.
For the BT-LUTPB algorithm, we observe significant reduction of overhead more than 73% in MO and 30% in ASO, as shown in Figure 8 and Figure 9. Namely, increasing processor frequency by 50% approximately reduces the overhead of scheduling in BT-LUTPB. Where as, the reduction on the overhead for other algorithms range between 25% and 32% in MO and ASO. The C-LUTB has extra overhead on ASO by approximately 2% for 48 threads, because of more contention added on Signal phase to read a single shared LUT entry.

Increasing the speed of switching packet by 2x improves the performance of barrier algorithms ranging between 4% (CPB) and 28% (BT-LUTPB) in MO as shown in Figure 10(a) and Figure 11(a). It contributes to reduce the overhead for C-LUT by more than 39% in ASO as depicted in Figure 10(b) and Figure 11(b).

VI. CONCLUSIONS AND FUTURE WORK

It is an interesting case to face challenges due to the increasing number of cores. Therefore, we have implemented several barrier algorithms to optimize the OpenMP runtime library by using the specific hardware features of the SCC architecture. As part of a quantitative performance evaluation, our experimental results highlight that we can obtain a significant reduction in overhead for barrier algorithms by using Chain LUT-Polarity busy-wait approach. The C-LUTB is the best barrier synchronization which allows 98.5% (MO in Pure Overhead) faster synchronization than S-MSB and approximately 25.6% (MO in Pure Overhead) faster than CPB-UC algorithm. Also, C-LUTB has low usage for the SCC features and it shows low power consumption. However, we first developed the evaluation criteria with micro-benchmarks for choosing barrier synchronization OpenMP schemes of many-core system. We also have developed barrier algorithms based on three concepts; a communication patterns, explicit allocation of barrier structures in the MPB and the memory access. For a simplest implementation of a spin-lock routines, the barrier synchronization rely either upon memory access into shared memory on-chip (e.g. MPB) or it uses configuration registers (e.g. LUT). Finally, the finding of this paper will constitute the basis for our future work, namely the implementation of a fully compliant efficient OpenMP programming model for the SCC.

REFERENCES