

\section*{C-Lock: Energy Efficient Synchronization for Embedded Multicore Systems}

Seung Hun Kim, Student Member, IEEE, Sang Hyong Lee, Minje Jun, Member, IEEE, Byunghoon Lee, Won Woo Ro, Member, IEEE, Eui-Young Chung, Member, IEEE, and Jean-Luc Gaudiot, Fellow, IEEE

Abstract—Data synchronization among multiple cores has been one of the critical issues which must be resolved in order to optimize the parallelism of multicore architectures. Data synchronization schemes can be classified as lock-based methods (“pessimistic”) and lock-free methods (“optimistic”). However, none of these methods consider the nature of embedded systems which have demanding and sometimes conflicting requirements not only for high performance but also for low power consumption. As an answer to these problems, we propose \textit{C-Lock}, an energy- and performance-efficient data synchronization method for multicore embedded systems. \textit{C-Lock} achieves balanced energy- and performance-efficiency by combining the advantages of lock-based methods and transactional memory (TM) approaches; in \textit{C-Lock}, the core is blocked only when true conflicts exist (advantage of TM), while avoiding roll-back operations which can cause huge overhead with regard to both performance and energy (this is an advantage of locks). Also, in order to save more energy, \textit{C-Lock} disables the clocks of the cores which are blocked for the access to the shared data until the shared data become available. We compared our \textit{C-Lock} approach against traditional locks and transactional memory systems, and found that \textit{C-Lock} can reduce the energy-delay product by up to 1.94 times and 13.78 times compared to the baseline and TM, respectively.

Index Terms—data synchronization, multicore, clock, energy, performance.

\section{Introduction}

Multicore processors have become prevalent in modern computer systems, not only for high performance desktops or servers but also for mobile devices. In order to meet the increasing demands for higher performance, increasing CPU clock frequency was one of the most obvious methods in traditional processors. However, for single cores, this is turning out to be impractical due to prohibitive power and heat dissipation requirements \cite{1}. This limitation made the multicore approach a more viable and scalable solution to the performance demands of embedded systems. In fact, contemporary embedded systems, especially high-end products such as smartphones, are rapidly adopting multicore chips at their core \cite{2}.

However, adding more cores does not necessarily lead to a predictable gain in system performance due to the limited parallelism of real world programs (which was predicted by Amdahl’s law \cite{3}). There have been tremendous efforts to reach the theoretical limit of parallelism from many different perspectives, and data synchronization among multiple cores has been one of the most vexing issues. The data synchronization issue arises when two or more processors attempt to access any shared data simultaneously, and mishandling of these conflicts results in incorrect operations, which is likely to cause fatal errors.

Existing data synchronization methods are either lock-based or lock-free. The former includes locks, semaphores, and barriers; these methods block the accesses to the shared data from the processors which fail to acquire the permission. On the other hand, the latter allow all processors to access the shared data in an optimistic manner, and then perform rollback and/or re-execution when a conflict occurs. A well-known technique in this category is that of transactional memory (TM) \cite{4}.

The data synchronization techniques which were originally developed for general purpose systems cannot be transferred directly into the embedded world since they do not take the nature of embedded systems in sufficient consideration: these include stringent requirements for low energy consumption as well as high performance. More specifically, lock-based methods are widely adopted in embedded systems because of their simple control mechanism, but they sacrifice much parallelism, resulting in poor performance. On the other hand, lock-free methods such as TM perform speculative execution which might turn out to be wasteful of energy when the execution must be rolled back. In such cases, the rollback operation consumes additional energy.

In this paper, we propose \textit{C-Lock}, an energy- and performance-efficient data synchronization method for embedded systems. \textit{C-Lock} delivers TM-like parallelism in race conditions by detecting true data conflicts. The detection is done by considering the type, address range, and dependency of simultaneous accesses. In those cases when true data conflicts are detected, the cores which are not given permission to access the data are immediately clock-gated in order to minimize the dynamic

\begin{itemize}
  \item S.H. Kim, B. Lee, W.W. Ro and E.-Y. Chung are with the Department of Electrical and Electronic Engineering, Yonsei University, Seoul, Korea. E-mail: kseunghun@gmail.com, bh2@dtl.yonsei.ac.kr, wwo@yonsei.ac.kr, eychung@yonsei.ac.kr
  \item S.H. Lee and M. Jun are with Samsung Electronics, Korea. E-mail: sh1977.lee@samsung.com, minje.jun@gmail.com
  \item J.-L. Gaudiot is with the Department of Electrical Engineering and Computer Science, University of California, Irvine. E-mail: gaudiot@uci.edu
\end{itemize}
power consumption. Since no speculative execution and rollback are performed, **C-Lock** results in a higher energy efficiency than TM. Also, due to the immediate clock-gating of cores, **C-Lock** can consume less energy than lock-based methods. All these advantages can be achieved with simple hardware support and marginal modifications of the software, as will be shown throughout the paper.

In the rest of the paper, we will summarize and discuss the background and related work in Section 2. Next, we will present the motivation of **C-Lock** in Section 3 and the technical details and implementations of **C-Lock** in Section 4. Finally, we will evaluate the performance of **C-Lock** against the state of the art baseline and TM in Section 5, followed by a conclusion of this work in Section 6.

## 2 Related Works

The two most common paradigms in explicitly parallel application programming are shared memory and message passing [5]. Among them, the former is usually used in the context of single-chip multicore system, and locks and TM are the two most prominent methods for such a system. Most of the existing literatures have evaluated these methods with respect to performance and software programmability. However, most of them have ignored energy efficiency which is one of the most important metrics in the context of embedded systems. In the rest of this section, we will summarize the previous data synchronization methods from a system energy perspective by classifying them into three categories: Lock, TM, and hybrid scheme.

### 2.1 Lock-based Approach

Rajwar et al. proposed two methods: Speculative Lock Elision (SLE) [6] is a hardware-based approach which elides the unnecessary lock-induced serialization from dynamic execution stream. More specifically, it allows non-conflicting critical sections to be executed and committed concurrently. When data conflicts occur, the corresponding threads are restarted to acquire the lock in a serialized manner: Transactional Lock Removal (TLR) [7] also uses hardware to convert lock-based critical sections transparently and dynamically into lock-free optimistic transactions. It resolves the data conflicts based on the time-stamp in order to provide transactional semantics and freedom from starvation.

Monchiero et al. [8] proposed a hardware lock that optimizes power and performance by replacing the processors polling with hardware notification; they used a hardware block called *Synchronization-operation Buffer (SB)* which monitors the shared variable and, if it is changed, notifies the processor of the change so that the processors energy- and bandwidth consuming polling operation can be avoided. This improves the performance and power-efficiency of data synchronization by reducing memory overhead. They have extended the work to reduce the hardware complexity of **SB** and to improve the scalability of the proposed architecture [9]. Yu and Petrov also proposed a hardware synchronization module called Distributed Synchronization Controller (DSC) to reduce the bus contention traffics and achieve high energy efficiency [10]. Also, it has been mentioned that various power-down modes including clock-gating technique can be applied in the proposed **DSC**.

Some other works proposed energy-aware lock methods. The thrifty barrier [11] is proposed as a hardware-software approach to reduce the energy waste in barrier spin-loops by estimating the wait time and forcing the processor into an appropriate low-power sleep state. The threads arriving earlier than the thrifty barrier push the target processors in one of sleep states by predicting the barrier stall time. Also, Liu et al. [12] achieved some measure of power saving by applying the Dynamic Voltage Frequency Scaling (DVFS). They predicted the stall time of the barrier which is estimated by simple predictors based on prior history.

Golubeva et al. [13] evaluated busy-waiting spinlocks, delay-based spinlocks, and sleep-based spinlocks on the MPARM simulation framework. They explored several features provided by the hardware, and various operating conditions imposed by the software.

Ferri et al. [14] initially proposed a hardware semaphore in which cores spin on a local scratchpad memory (connected directly to each core) to reduce the access frequency of the shared bus. Later, they extended this work to a hybrid (wait/sleep) semaphore which pushes a processor into a sleep state only after a fixed number of busy-wait cycles [15]. They also compared their method to other energy-oriented methods such as HW-locks (using Test-and-Set instruction), DVS-policy (with HW-locks), HW-lockssleep, and DVS-sleep. The proposed timeout-based semaphore provides higher energy savings (30% on average).

Also, there are efficient software oriented mechanisms that are based on synchronization primitives such as **TEST&SET** and **TEST&TEST&SET**. Kogi et al. analyzed the overhead of the synchronization primitives and proposed four mechanisms that reduce the overhead [16]. They concluded that the Queue-On-Link-Bit (QOLB) primitive provides substantial speedup compared to the other methods. Also, Rajwar et al. pointed out the protocol complexity and software overhead of QOLB and they proposed Implicit QOLB (IQOLB) to solve the problems [17].

Even though several lock-based schemes have been proposed, they share the drawback of being overly conservative in their exploitation of parallelism. More specifically, they deal with the data synchronization issue at the process-level by granting a unique identifier to each process. For this reason, a process cannot simultaneously run on two shared data elements if it has already requested a lock for one of them. To counter this problem, an identifier should be given to each shared data element rather than to a process, at the cost
of dramatically increased programming complexity. The common issue encountered with conventional explicit synchronization schemes such as locks, semaphores, mutexes, etc. is indeed programming complexity. The parallel programs based on these schemes (an abstraction containing explicit synchronization) must be aware of its details in order to avoid races or deadlocks.

2.2 TM Approach

TM provides sufficient programmability to the programmers by abstracting the details of the synchronization. Consequently, the programmers rather focus on the functionality. Even though TM simplifies the programming model and maximizes concurrency, transactions may suffer from interference which causes them to abort and from heavy overheads for memory accesses. It should be noted that, in recent years, there has been increasing interest in both software transactional memory (STM) [18], [19] and hardware transactional memory (HTM) [20], [21], [22].

Moreshet et al. [23] evaluated the energy cost of managing memory contention in a multiprocessor environment with a special emphasis on the conflict scenarios within transactions. They showed that TM has an advantage over locks in terms of energy consumption, but that this advantage largely depends on the architecture of the system, the contention level, and the conflict resolution policy.

Ferri et al. proposed a hardware TM called Embedded-TM [24], [25], which aims at balancing energy efficiency and simplicity in an embedded system. However, the energy efficiency of TM strongly depends on the accuracy of the speculation. Indeed, whenever the speculation is wrong, it consumes non-negligible energy for the associated transaction abort and restart. Sanyal et al. [26] proposed a shutdown method to tackle this issue; they dynamically turned off a processor by gating all its clocks, whenever any transaction running on the processor is aborted. Even though the shutdown scheme somewhat mitigates the waste of energy when the speculation is wrong, there is no way to completely compensate for the energy already consumed by the speculatively executed parts.

2.3 Hybrid Approach

There has been a hybrid approach to combine the merits of lock and TM. Adaptive locks [27] is a hybrid method which dynamically selects TM or a mutex lock to improve performance. However, it only focuses on improving program execution time; in fact, the energy consumption is not discussed. That is, the system allows speculative execution that may cause a power-consuming rollback operation. In addition, there is no power saving mechanism for the processors waiting for the execution of a critical section. Also, introducing adaptive locks requires additional adaptive logic as well as run-time cost-benefit analysis, which causes additional overhead.

To summarize, the traditional lock-based schemes are inadequate from a performance perspective, while TM methods are not well designed from an energy perspective. For these reasons, it is necessary to design a data synchronization method which exploits the advantages of both methods. In fact, some methods categorized in the hybrid approach are actively challenging this issue. Our proposed method \textit{C-Lock} can also be categorized as a hybrid approach. Compared to the previous works, \textit{C-Lock} has a unique feature; it normally behaves like a lock scheme for energy efficiency, but it shows a transactional behavior for checking data conflicts. For further energy saving, it performs clock-gating during the stalls. As mentioned in the above paragraph, the power consumption problem has not been discussed in the earlier hybrid approaches such as adaptive locks. The effectiveness of \textit{C-Lock} over the other methods will be discussed in Section 5.

3 Motivational Examples

In this section, we will emphasize the advantages of \textit{C-Lock} by comparing it to the Lock$^1$ and TM. We use a simple piece of code as an illustrative example (see Fig. 1) to describe the difference between lock, TM, and \textit{C-Lock}. \texttt{BEGIN} and \texttt{END} at line 9 and 11 define a critical section in the code. Although the optimized critical section implementation may be different for each synchronization mechanism [28], the example is presented to show the distinct features of the exclusive approach and the speculative approach.

```c
1 // Shared Data, array A
2 int A[10];
3
4 void incr(int *ptr)
5 {
6   int index;
7   index = foo();
8   BEGIN()
9     ptr[index] = bar(ptr[index]);
10   END()
11 }
```

Fig. 1: An example code (simple increment)

3.1 Exclusive Approach and Speculative Approach

First, let us consider the case where the cores are accessing different data elements as shown in Fig. 2. In this example, core0, core1, core2, and core3 are simultaneously accessing the variables $A[1]$, $A[2]$, $A[3]$, and $A[4]$, respectively, via the function \texttt{incr}. With the Lock method, there is no parallelism among the operations simply because the critical section in the function \texttt{incr} is controlled by the same \texttt{lockId}. On the other hand, the cores execute their operation simultaneously in TM.

1. In the rest of the paper, the term Lock refers to the traditional HW-lock.
the elimination of speculative execution while exploiting parallelism as much as possible. The proposed mechanism is developed to achieve this purpose.

As shown in Fig. 2, \textit{C-Lock}, the address range is used for detecting true dependencies so as to decide whether to execute or hold the operation. Since all the cores are accessing different variables (i.e., non-overlapped address range), \textit{C-Lock} detects no conflict. Thus their operations can be performed simultaneously, achieving TM-like parallelism. Also, the system permits only one access at a time if there is a true conflict among the cores. Moreover, the cores without access permission move into the clock-gated state to reduce dynamic power consumption. Consequently, \textit{C-Lock} scheme yields higher energy efficiency than TM and provides higher performance than Lock.

In fact, lowering the programming complexity is also important in parallel processing. For example, a fine-grained locking approach requires a large amount of programming effort while the method provides more parallelism than a coarse grained approach. TM has thus been proposed to provide a convenient programming interface for synchronization. However, direct transformation of traditional lock-based critical sections into transactions does not guarantee correct execution of the program [29]. Therefore, the programs should be redesigned to be run on TM systems. On the contrary, conventional lock-based programs can be easily transformed for execution with the \textit{C-Lock} approach since the proposed scheme prohibits speculative execution. In addition, \textit{C-Lock} prevents deadlock problem. The mechanism is described in Section 4.2.1.

4 \textit{C-Lock}

In this section, we will describe the concept, implementation, operation, and usage of \textit{C-Lock}. First, we will give a brief overview of \textit{C-Lock} in Section 4.1. The technical details and implementations of the \textit{C-Lock} hardware will be presented in Section 4.2. Based on the hardware architecture of \textit{C-Lock}, the detailed operation of \textit{C-Lock}, which is the interaction between its software part and the hardware part will be explained in Section 4.3. At the end of the section, an example will be given in Section 4.4 which will show the detailed operation of \textit{C-Lock}.

4.1 Overview

The main idea of the \textit{C-Lock} system is to exploit available parallelism with true conflict detection and to minimize dynamic power consumption with clock-gating for the idle cores. Fig. 4 shows the concept of the proposed mechanism. Before the execution of the critical section, every core sends the address range to be accessed; \textit{Addr\_range}_0 to \textit{Addr\_range}_3 in the figure.
the accesses of other cores. One Item consists of the following fields:

- *BaseAddr*: base address
- *Size*: access size
- *R/W*: read/write
- *gIdx*: global index for conflict detection
- *V*: one bit valid field for indication of the validity of Item

The internal architecture of C-LockManager is shown in Fig. 6a; it is composed of N Item entries and conflict checking and clock-gating logics. The microarchitecture of Pool is shown in Fig. 6b. Each Core, initiates the C-Lock operation by recording the access information to the corresponding Pool (i.e., Pool, in C-LockManager). This operation is done by calling the function ADD_ITEM, and is a typical bus write operation through a dedicated bus port (port) (details are shown later in Section 4.3). Each core can register at most M Items. In our implementation, the handling logic of Pool manages the status of the entries by checking the valid fields thus puts the incoming Item to an empty entry. Therefore, the program does not need to identify which Item entry it is accessing.

During the above procedure, the arbiter can inspect the possible deadlock. In principle, programmers are responsible for the correct program execution. In other words, deadlocks should be avoided at software design time [30]. However, the proposed C-Lock system also offers deadlock prevention to lessen the programming effort. For this operation, we have assumed that the compiler could provide the lock dependency graph shown in Fig. 7. In detail, the compiler figures out whether each C-LockId is nested or not using the BEGIN_C-LOCK macro at compile time; BEGIN_C-LOCK is described in Section 4.3. The analysis result, the dependency information of each lock, is notified to the arbiter in C-LockManager before the program is executed. As an example in Fig. 7, a deadlock may occur if two threads have different sequences for the nested locks: C-LockId 2 and 3. The arbiter is informed of these nested locks; it defines them as a nested lock group. When the arbiter receives the C-LockId from the request for an atomic operation of the Pool, it allows only one C-LockId to be acquired by the cores within the group. For example, if C-LockId 2 and 3 are nested as shown in Fig. 7 and C-LockId 2 is already acquired by a given core, then a request from the Pool with C-LockId 3 is denied by setting to 0 the arb_decision signal in Fig. 6b.

### 4.2 C-LockManager Implementation

The details of C-LockManager are presented in this subsection. In the following, we assume that there are N cores in the processor, and that each core can record M Items with C-LockManager. Item refers to a storage that contains information for checking true conflicts with
An atomic access is triggered when the core sends the begin command to the corresponding Pool through the bus. Next, the Pool requests a grant to the conflict checking operation from the arbiter. This procedure is necessary since multiple cores can trigger their atomic accesses at the same time if C-LockManager is connected via multiple buses (e.g., bus matrix). If the Pool gets the grant, it sets the gIdx fields of the newly registered Item entries to the current global index values which is broadcasted by the global counter. At the same time, the granted Pool signals the global counter to increment the global index value.

After that, the Pool broadcasts all the M Items to the itembuses and requests the other Pools to check for conflicts by comparing the broadcasted Items and their own registered Items. Immediately after, the conflict checking process is performed in the other Pools. The major part of this process is done by the conflict checker (shown in Fig. 6b). A conflict checker is dedicated to an Item entry and checks whether any of the M broadcasted Items causes true conflicts with its own Item. As an illustration, imagine that Item_loc is the Item to which the conflict checker is dedicated (loc stands for local), and Item_rem is one of the broadcasted Items (rem stands for remote). Then, Item_loc and Item_rem have true conflicts if the following conditions are simultaneously present:

- Both Items are valid
- Their address ranges overlap
- At least one of them is a write operation
- gIdx of Item_loc is smaller than gIdx of Item_rem

The first two conditions are obvious, while the third one filters out the false dependency (i.e. Read-after-Read). The fourth condition detects possible data hazard; if the fourth condition holds, it means that the Item_rem is registered later than Item_loc (since global counter is an ascending counter) and, therefore, executing Item_rem prior to Item_loc may cause data hazard in the requested memory region.

Each conflict checker performs the above operation for all the broadcasted Items and finally produces out the conflict signal by simply pairwise ORing the results. Again, by ORing all the conflict signals from the conflict checkers, the Pool finally makes the signal which indicates whether any of the broadcasted Items are in conflict with the Items in this Pool. The signal is AND-gated with the arb_decision signal to output the final conflict_out signal.

After that, the Pool which requested conflict checks from the other Pools gathers the results by watching the conflict_in signals in Fig. 6b. If any of the other Pools reports conflict, it means the requested atomic access cannot be executed at this time, and therefore, the Pool disables the clock of the corresponding core (i.e. deasserts the clock_enable signal). Also, the conflict_in signals are stored in the who_blocked_me register so that the Pool can watch the events of the blocked Pools being cleared and reattempts its access. This can effectively avoid the blocked Pools watching the activities from all the other cores. When no conflicts are reported from the other Pools, the core keeps running and executes the atomic access for the registered Items.

As shown in Fig. 6b, each Pool has its own Items and the number of Items is fixed as M in the proposed scheme. Therefore, if the number of requested Items to be registered is larger than M, some addresses cannot be registered. To solve this problem, Pool is designed to send the item_out signal to the arbiter if there is no empty space for the Item. Then, the arbiter sends the arb_decision
signal back to the core for synchronization. In details, if any of the core is executing a critical section, the arbiter sets the `arb_decision` signal to 0 to stall the execution of the core which issues `item_cfg` signal. If not, the arbiter permits the core to execute the critical section.

Also, the proposed design reduces the complexity of the interconnections in the Pools and the arbiter. As shown in Fig. 6b, the conflict checker is dedicated for each Item. Therefore, additional control is not required for address comparison. For example, if the Items are shared among the Pools, many multiplexers are needed to connect the all Items with the conflict checker and the arbiter should decide the connection for every grant operation.

When the core completes its atomic access, it asks `C-LockManager` to clear the corresponding Item entries by calling the function `END_C-LOCK` which effectively sends out an end control command to the corresponding Pool (details are described later in Section 4.3). When the Pool clears the Item entries, it also notifies the other Pools that its previous atomic access has been completed. If there is any other Pool which was blocked by this Pool, it would reattempt its access first by requesting the grant from the arbiter.

### 4.2.2 Overhead Analysis

Depending on the width of the various fields in `Item` and the width of the system bus, setting the `Item` entry in `C-LockManager` can vary from one to many bus clock cycles. If the data length of an `Item` is 1 (excluding valid field because it is automatically set and reset by `C-LockManager`), the number of `Items` to be registered is `m`, and the bus width is `b`, the number of cycles needed for a core to transfer the `Items` to `C-LockManager` is simply `⌈(m \times l)/b⌉`. In our implementation, we assume that setting one `Item` (including `BaseAddr`, `Size`, and `R/W` fields) is done in two clock cycles. When the `begin` command is received, the arbitration takes one cycle and, if the Pool is granted access, setting the `gIdx` field is performed within the same cycle. Issuing the conflict check request and broadcasting `Items` occurs at the next cycle, while the conflict check process is done in the other Pools at the following cycle. Finally, at the next cycle, the Pool can determine whether to gate the clock of the core or not according to the conflict check results. In total, at least six cycles are required from the moment when the core begins setting `Item` to the moment when the gating of the clock is decided. The cycle count may increase if the Pool was not granted access from the arbiter or the core attempts to set multiple `Items` for the atomic access.

We implemented `C-LockManager` with Verilog HDL, to analyze the hardware overhead of the proposed scheme. We performed a topographical synthesis with Synopsys Design Compiler and performed static timing and power analysis with Synopsys PrimeTime\(^2\). Synopsys 90nm logical and physical technology libraries were used in the implementation. We implemented various versions of `C-LockManager` considering the following conditions; the type aiming at high speed (750MHz), and the other at optimizing for area and power for a 200MHz clock frequency (200MHz is the bus clock speed used in our experiments).

The results show the area overhead and power consumption of `C-LockManager` for the various combinations of the number of cores (`N`) and the number of `Item` entries (`M`). As shown in Fig. 8a, the required area is proportional to the number of cores and entries. The main factor of the increased area is the number of registers for the `Item` in the Pool. For this reason, some significant large amounts of leakage power are caused compared to the designs with higher frequency as shown in Fig. 8b and Fig. 8c. On the contrary, the designs with lower frequency that are shown as continuous line, are less affected from the increase area. Consequently, the later designs are more practical. Above all, the increasing ratio of the power consumption over the number of `Items` is remarkably small compared to the designs with higher frequency. From the results shown in Fig. 8, the design with lower frequency operate at the same clock frequency as the bus (in our experiments), while the power consumption is less than one ninth and the area is 25% smaller in average for 16 kinds of implementations.

In addition, supporting out-of-order execution may
complicate the implementation and management of C-Lock; the back-end of the core (re-order buffer (ROB) and components for handling commit) should not be clock-gated until the ROB entries which are being committed are finally committed. Therefore, to support out-of-order execution, modifications are needed in C-LockManager such that it can separately control the clocks of the frond-end and the back-end part of each core.

4.3 Software-Hardware Interaction

In order to take advantage of C-Lock, the software is in charge of setting the information for the atomic access as well as triggering and clearing operations. These tasks can be handled simply by writing the commands to the Pool to which the core is dedicated, via the corresponding port in C-LockManager. Note that the base addresses of the ports are determined at the system level so that they can be easily calculated with the coreIds. Consequently, from the perspective of the system software, the atomic accesses can be performed by setting the registers visible by the system software shown in Table 1, with the macro functions shown in Fig. 9. A simple code of the macros is displayed in Fig. 9. Also, it should be noted that an advanced Integrated Development Environment tool can be developed to provide information for the macro functions to lessen the burden of the software programmer. For example, a list of available C-LockIds or the access history (read/write) of shared variables can be informed to the programmer.

Now that the roles and implementations of hardware and software are described, we can explain the interaction between software and hardware in C-Lock (Fig. 10). Note that only two cores are used in the figure for simplicity, but that the number of processors can be easily extended. An atomic access in C-Lock is initiated when the system software calls the function ADD_ITEM to register the information of the atomic access ([S1]). This step is initiated from the hardware side, i.e., by C-LockManager, when the corresponding Pool successfully registers the incoming information to its Item entry ([H1]). After that, the system software starts the C-Lock operation by setting the begin command at the regCtrl register, i.e., by calling the function BEGIN_C-LOCK, [S2]. The function notifies current coreId and C-LockId to C-LockManager.

<table>
<thead>
<tr>
<th>Table 1: Regs in C-LockManager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>software</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>internal</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

![Fig. 9: Macros for C-Lock](image)

From BEGIN_C-LOCK call, C-LockManager determines whether to let the core proceed with its atomic access or to stop the core until the conflict is resolved, as described in Section 4.2 ([H2] to [H5]). That is, the function call triggers the request for the critical section access. Also, the triggering should be separated from the Item registration since the proposed scheme allows multiple calls of the ADD ITEM function before the execution of a critical section. If C-LockManager detects no conflict for the access request, the core proceeds with its tasks with the shared data ([S3]). Otherwise, the core is set to the clock-gated state by C-LockManager ([H5]), while the corresponding Pool waits for the other cores which have blocked its associated core to complete their accesses ([H6]).
The address comparison scheme of the proposed C-Lock method is not appropriate to detect a conflict of nondeterministic memory access or dynamic allocation in the critical section. In this case, the critical section should be exclusively executed. For that purpose, the programmer needs to specify the mutual exclusion by setting the constant value in the arguments of the ADD_ITEM function. A line of code such as ADD_ITEM (coreId, 1, 1, bWRITE) can achieve a mutually exclusive operation for nondeterministic memory access or dynamic allocation. As a future work, we will investigate more elaborated mechanism to solve this limitation.

When the atomic access is finished, the system software notifies C-LockManager to invalidate the corresponding Item entries by calling the function END_C-LOCK ([S4]). The function END_C-LOCK not only clears the Item entries in C-LockManager ([H7]), but also notifies the other Pools that the shared region has been released ([H8]). Then, the Pools which have been waiting for this event go back to arbitration to check for conflicts, and the procedure repeats from [H2].

### 4.4 An Example of C-Lock Operation

In this subsection, we will explain the operation of C-Lock with a detailed example (see Fig. 11). It is assumed that there are four cores (N = 4) and that each core can register more than two Items. The rows labeled “Core” show the action of the software (“Software”), the operational status (“Status”), and the clock activity (“Clock”) of the cores at each time slot.

At T0, Core0, Core1, and Core2 record the Items to the Pools and request their atomic accesses (Core3 continues performing its local operations). Since the address range Core0 requests do not overlap with others, it is granted access (area labeled “G”) and initiates its atomic operation (foo()) without its clock being gated. Also, Core1 and Core2 receive the grant for the operations (bar()) since there is only a false dependency between them, i.e., both are read operations.

At T1, Core3 requests an atomic operation with C-LockId 6. Since we have assumed that all Item entries are empty before T0, the request can be granted if the C-Lock system considers only the registered Items at T0. However, as described in Section 4.2.1, nested lock is informed to the arbiter. Therefore, the request of Core3 at C-LockId 6 is denied since C-LockId 5 is already acquired by Core0. As a result, the status of Core3 is changed to Idle. If there is no consideration for the nested lock, the request of Core3 at T1 will be granted and the status of Core0 will be changed to idle at T2 due to the overlapped address. In this case, a deadlock may happen if Core3 requests an atomic operation at T3 as the gray-colored text indicates; a conflict occurs due to the Items registered by Core0 at T0. Also at T1, Core1 releases its ownership of the address range 304-308 (area labeled “R”).

At T2, Core0 requests the atomic access in a nested manner and it is granted since there is no overlapped address. However, Core1 is denied (area labeled “D”) since the request has a true data conflict with the atomic access being held by Core0. As Core0 completes its atomic operation and releases its all ownership of inner nest and outer nest at T3 and T4, C-LockManager performs an arbitration again and grant access to Core3 by enabling the clock.

### 5 Experiments

In this section we evaluate our proposed C-Lock method using several benchmark applications. We first describe the experimental setup, followed by a detailed discussion of the results.

#### 5.1 Experimental Settings

In our experiments, the baseline system is implemented as a clock-gating applied SB [8] method considering the power-saving approach of Yu and Petrov [10] which is mentioned in Section 2; the processors that are registered in SB are clock-gated. By presenting performance comparison to the baseline system, we will show that the advantage of C-Lock is not only from the clock-gating but also from the synchronization mechanism.

---

Fig. 11: Description of C-LockManager process
Also, we evaluated C-Lock against the transactional memory systems shown in [14]. Since the balance between throughput and energy is critical for embedded systems, we selected the energy-delay product (EDP) as the performance metric.

We implemented the proposed C-Lock and the baseline system onto the MPARM simulation framework, presented by Benini et al. [31]. MPARM is a cycle-accurate virtual platform written in SystemC, which consists of processors, interconnect, memories, and peripherals. With its cycle-accurate power models, it provides much useful information, such as the total number of cycles, system energy consumption, abort rate, etc. When measuring the execution clock cycles and energy consumption, we used static task mapping for the benchmark programs to concentrate on the data synchronization while avoiding the influence of any other factors. Specifically, each task is mapped on its own processor hence there is no overhead for task switching or task migration [14].

The simulation platform used in the experiments is composed of:

- a configurable number of ARM cores with own caches,
- a main memory,
- snoop devices,
- an interconnection [14], and
- a C-LockManager with the clock-gating feature for synchronization.

The benchmark applications are chosen from the STAMP benchmark suite [33], the MiBench suite [34], thread scheduling model (TSM) in [32], and microbenchmarks for MPARM. They are summarized in Table 2. The portion of the time spent in the critical section is shown as a percentage. In the case of the matrix benchmarks and thread scheduling model, the portion of time can be varied.

In fact, the numbers next to ‘C’ stand for the percentile ratio of the time spent in the critical section to the total execution time in the matrix benchmarks. The program consists of a sequence of atomic operations executed on a shared matrix, logically subdivided into overlapping regions. To guarantee that concurrent accesses to overlapping regions do not conflict, processors should obtain an exclusive access to each region.

| TABLE 2: The benchmark applications |
|-------------------------------|-------------------|---------------------|
| Suite                        | Name              | Description (critical section %) |
| Micro-benchmarks             | C5                | Matrix arithmetic (5%)        |
|                              | C20               | Matrix arithmetic (20%)       |
|                              | C60               | Matrix arithmetic (60%)       |
|                              | C85               | Matrix arithmetic (85%)       |
| MiBench                      | patricia          | IP prefix matching (59%)      |
| STAMP                        | labyrinth         | Routes paths in a 3D maze (77%) |
|                              | vacation          | Travel reservation system (10%) |
|                              | kmeans            | K-means clustering (4%)       |
| TSM [32]                     | scheduling-low    | Work-stealing modeling (42%)  |
|                              | scheduling-high   | Work-stealing modeling (72%)  |

The other benchmarks represent more complex applications than the matrix benchmarks. The patricia program executes a prefix matching of IP addresses for network applications. The parallelized MPARM version of the program is presented in Ferri et al.’s work [25]. The labyrinth is a program which finds the shortest paths between pairs of starting and ending points in a 3D maze. Both programs include the critical sections that are implemented using a coarse-grained scheme. We have chosen the programs so as to show the effectiveness of exploiting parallelism versus lock and eliminations of the speculative execution versus TM. The kmeans benchmark is a partitioning program where the objects to be partitioned are equally subdivided among the threads. Vacation emulates a non-distributed travel reservation system. Each thread tries to access a central database system which keeps a record for available plane tickets or available hotel rooms using a transaction manager. In addition to the traditional benchmark programs, we have modeled the thread scheduling algorithm in Arora et al. [32]. The algorithm is based on a work-stealing [35] method using a deque structure. Each thread interacts with the work queues of other threads when there is no work stored in its own queue. The programs are modeled as two cases; the suffixes (low and high) indicate the probability of having conflicts during the work-stealing process.

5.2 Experimental Results

5.2.1 Results for the Matrix Benchmarks

The effectiveness of the proposed scheme is in exploiting available parallelism with low power consumption. In detail, C-Lock prevents unnecessary exclusive execution using the access address range comparison and the system does not perform a power-wasting speculative execution. Also, the clock-gating feature reduces the dynamic power of the cores that are not granted for a critical section access. The following simulation results will show these advantages.

We first present the experimental results for the matrix benchmarks. Fig. 12 shows the results using various numbers of cores (2, 4, and 8) for the three metrics: execution cycles, energy, and EDP. Note that the values are normalized to the baseline system with 1 core and 1 is subtracted from it. With this data presentation, a negative value can be simply interpreted as ‘improved’ and a positive value as ‘degraded.’ The length of the bar indicates how much the metric is improved or degraded. Also, note that the results with single core are omitted since they are identical for all the three methods; the reason is that we use static task mapping to each core without multi-threading within the core.

Let us first discuss the execution cycles. For C5 and C20, all the three methods the baseline, TM, and C-Lock show almost the same results. This is because the portion of critical section is not large enough to show a noticeable difference among the three methods even when 8
cores are used. However, given a large enough critical section, the gap between the three methods becomes more prominent, as shown for C60 and C85. The baseline system shows the poorest performance scaling as the number of cores increases, since it blocks core(s) whenever more than one cores attempt to access the memory, even though their accesses do not cause conflicts. TM shows better performance scaling than the baseline but not as good as C-Lock. Even though both TM and C-Lock check true conflicts for their atomic access, the result demonstrates that the conflict checking and blocking mechanism of C-Lock is superior to the speculative execution and abort mechanism of TM.

As for the energy consumption, TM shows the worst characteristics due to its aborted execution overhead; for example, TM consumes 1.6 times more energy compared to C-Lock in C85. The baseline system and C-Lock show almost same results in energy consumption since clock-gating features are applied in both systems. The clock of a core is stopped when the processor is registered in the SB entry or there is any data conflict. The systems automatically resume the clock when the execution of the previously registered processor is finished or the data access is granted.

Although the baseline system achieves less energy consumption similar to C-Lock, the system shows poor EDP due to the long execution delay. The EDP results of the three methods are shown at the bottom of Fig. 12. The results show that the advantage of C-Lock becomes more prominent as the portion of critical section increases. C-Lock shows 86% of EDP reduction in case of 8-cores with C85 whereas TM shows 66% and the baseline results in 16%.

The effectiveness of the proposed C-Lock method is also noticeable as the number of cores increases. As shown in Fig. 12, EDP reduction of C-Lock as the number of cores increase 2 to 8 is 36% in C85 while the value of the baseline is -2% and TM is 32%.

5.2.2 Results for Complex Benchmarks

In this subsection, we evaluate C-Lock with more complex applications: patricia and labyrinth. The results are shown in Fig. 13, and the values are calculated in the same way as Fig. 12.

As shown in the figure, C-Lock provides energy- and performance-efficient execution of the applications in most of the cases compared to the baseline system and TM. On average, the proposed C-Lock method shows 20% and 41% more performance improvement than the baseline and TM respectively. Especially, remarkable improvements are achieved in patricia, labyrinth, and scheduling-high as demonstrated by the following results; as the number of cores increases from 1 to 12, the maximum performance improvement of the baseline is 7% in patricia, 1% in labyrinth, and 4% in scheduling-high. On the other hand, the performance of C-Lock is gradually improved by up to 51%, 62%, and 21%, respectively.

In the case of patricia and labyrinth, TM shows better performance than the baseline, but the gain is much lower than C-Lock, even though it also guarantees data consistency. In addition, severe performance degradation is occurred in scheduling-high while the C-Lock method reduces the program execution cycles. Such a big difference between TM and C-Lock comes from how and when these two methods use the true dependence checking feature; C-Lock uses the feature before entering the critical section, and blocks and clock-gates the core(s) other than the granted one until the data conflict is resolved. Therefore, if a core is blocked and clock-gated, it never wakes up and attempts access again until the access to the shared data finally becomes available. On the other hand, TM speculatively enters a critical section and, if a conflict is detected, aborts the execution and tries again and again until the execution of the critical section ends without any conflict.

There is a possibility that the critical section is executed numerous times due to frequent aborts in TM; it might cause overall performance degradation. In the case of kmeans, the performance improvement in execution time as the number of cores increases is smallest in TM among three synchronization methods even though the application has outstanding scalability. We believe this is due to the frequent aborts in the application. As a matter of fact, the result of 12-cores case is worse than 8-cores while the other two approaches do not; the amount of the improvement in 12-cores with kmeans is 75% whereas the values of the baseline is 74% and TM is 64%. Also, TM shows the worst result in execution time with vacation. The performance improvement of the baseline and C-Lock is 12% while TM is 3% in 12-cores case.
In addition, C-Lock shows better results in scheduling-low compared to the baseline and TM. Although the baseline system has advantages over TM due to the exclusive operation, the amount of performance improvement is smaller than C-Lock since the system does not fully exploit parallelism. The negative effect of this limitation gets worse as the portion of the critical section increases. As shown in the execution cycles of scheduling-low and scheduling-high, the advantage of parallel processing in the baseline considerably decreases compared to C-Lock as the amount of time spent in the critical section increases. For example, the performance improvement in scheduling-low is 45% and scheduling-high is 0.2% using 12-cores with the baseline model. On the other hand, the values of C-Lock are 53% and 20%, respectively.

As for the energy consumption, the TM scheme shows the worst results among three approaches due to the rollback operations even though the program execution time is reduced. The main reason of this inefficiency is shown in Fig. 14. As shown in the figure, the abort rate of TM is much higher than the gate-rate of C-Lock for patricia, labyrinth, and scheduling-high. Especially in labyrinth, compared to the dramatic increase in the abort rate of TM as the number of cores increases, the gate-rate is kept relatively constant. Consequently, the advantage of C-Lock is shown prominently for labyrinth and scheduling-high. C-Lock can reduce the energy consumption by 2.16 times in labyrinth and 4.02 times in scheduling-high compared to TM, respectively.

6 Conclusions

C-Lock is an energy- and performance-efficient data synchronization method for multicore embedded systems. C-Lock can save system energy by gating clocks of some cores which request shared data but are blocked.

\[ N_{\text{lock-gated}} \cdot N_{\text{crit-section}} \]  

where \( N_{\text{lock-gated}} \) represents how many times the core is clock-gated and \( N_{\text{crit-section}} \) represents how many times the core enters the critical section.
since the data are being occupied by another core(s). In order to minimize the performance loss due to conflict (stall, thereby), C-Lock checks the true dependencies among the cores by examining their address range, access type, and so on, unlike the traditional lock. These properties of C-Lock combine the advantages of locks and TM and offer the most efficiency; the experiments show that C-Lock can reduce EDP by a multiplicative factor up to 1.94 compared to the baseline and 13.78 compared to TM. These results demonstrate that the proposed C-Lock approach can provide a power efficient memory consistency model. The proposed scheme may require significant work from the programmer; this can be regarded as a trade-off of the improved performance including power efficiency. On the other hand, the hardware area overhead and the power and execution time overhead of the proposed approach are not significant. The high efficiency of C-Lock relies mostly on the special hardware C-LockManager, with marginal support from the software.

In the near future, we plan to extend C-Lock by adding out-of-order core support and examining it for more benchmarks. Also, although the modification needed on the software side is marginal, we plan to develop a compiler assistance for C-Lock so that the manual modification on the software can be more reduced.

ACKNOWLEDGMENTS

This work is partly supported by the National Science Foundation under Grant No. CCF-1065448, by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (MEST) (No. 2009-0077326, 2010-0013202, and 2010-0025423), and by the Mid-career Researcher Program through NRF grant funded by the MEST (No. 2010-0026822). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Foundations.

REFERENCES


Seung Hun Kim received the B.S. and M.S. degrees in electrical and electronic engineering from Yonsei University, Seoul, Korea, in 2009 and 2011, respectively. He is currently Ph.D. student in Embedded Systems and Computer Architecture Laboratory, School of Electrical and Electronic Engineering, Yonsei University. His research interests include the Transactional Memory systems and multi-core architecture.

Sang Hyong Lee received the BS and MS degrees in electrical and electronic engineering in 2003 and 2012, respectively, from Yonsei University, Seoul, Korea. He is currently a senior engineer with, Samsung Electronics, Suwon, Korea. His research interests include System-on-Chip architecture.

Minje Jun (M'08) received the B.S., M.S., and Ph.D. degrees in Electrical and Electronic Engineering from Yonsei University in Seoul, Korea, in 2006, 2008, and 2013, respectively. He is now with System LSI Division, Semiconductor Business, Samsung Electronics. His research interests include Network-on-Chip and DRAM-stacked SoC architectures with the special emphasis on their design automation.

Byunghoon Lee received the B.S. degree in electrical and electronic engineering in 2010 from Yonsei University, Seoul, Korea, where he currently working toward the Ph.D degree in electrical and electronic engineering. His research interests include low power design and memory architecture.