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 *C–Lock*, an energy- and performance-efficient data synchronization method for multicore embedded systems. *C–Lock* achieves balanced energy- and performance-efficiency by combining the advantages of lock-based methods and transactional memory (TM) approaches; in *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, *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 *C–Lock* approach against traditional locks and transactional memory systems and found that *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

1 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 method in traditional processors. However, for single cores, this is turning out to be impractical due to prohibitive power and heat dissipation requirements [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 [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 [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) [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, lockbased 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 *C–Lock*, an energy- and performance-efficient data synchronization method for embedded systems. *C–Lock* delivers TM-like parallelism in race conditions by detecting true data conflicts. The detection is done by

- *S.H. Kim, B. Lee, W.W. Ro and E.-Y. Chung are with the School of Electrical and Electronic Engineering, Yonsei University, Seoul 120-749, Korea. E-mail: kseunghun@gmail.com, bh2@dtl.yonsei.ac.kr, {wro, eychung}@yonsei.ac.kr.*
- *S.H. Lee and M. Jun are with Samsung Electronics, Suwon-si, Gyeonggi-do 443-742, Korea. E-mail: sh1977.lee@samsung.com, minje.jun@gmail.com.*
- *J.-L. Gaudiot is with the Department of Electrical Engineering and Computer Science, University of California, Irvine, CA 92697-2625. E-mail: gaudiot@uci.edu.*

Manuscript received 06 Mar. 2012; revised 12 Mar. 2013; accepted 26 Mar. 2013. Date of publication 07 Apr. 2013; date of current version 15 July 2014. Recommended for acceptance by S.W. Chung.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.84

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 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`. Kägi et al. analyzed the overhead of the synchronization primitives and proposed four mechanisms that reduce the overhead [16]. They concluded that the Queue-On-Lock-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.

```
1  // Shared Data, array A
2  int A[10];
3
4  void incr(int *ptr)
5  {
6    int index;
7    index = foo();
8
9    BEGIN()
10     ptr[index] = bar(ptr[index]);
11   END()
12 }
```

Fig. 1. An example code (simple increment).

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]–[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.

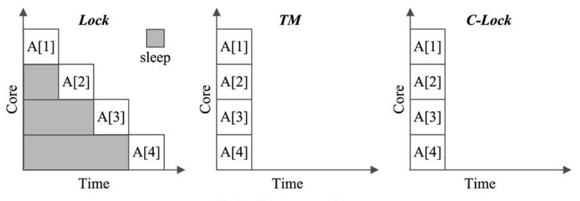
2.3 Hybrid Approach

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

```
// Different indexes at time T
   core 0: ptr[index] == A[1];
   core 1: ptr[index] == A[2];
   core 2: ptr[index] == A[3];
   core 3: ptr[index] == A[4];
```

(a) Function calls to different data



(b) Operation

Fig. 2. Parallel processing with different data.

actively challenging this issue. Our proposed method *C–Lock* can also be categorized as a hybrid approach. Compared to the previous works, *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 *C–Lock* over the other methods will be discussed in Section 5.

3 Motivational Examples

In this section, we will emphasize the advantages of *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 *C–Lock*. `BEGIN` and `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.

3.1 Exclusive Approach and Speculative Approach

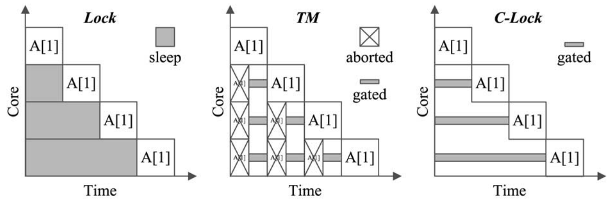
First, let us consider the case where the cores are accessing different data elements as shown in Fig. 2. In this example, core0, core1, core2, and core3 are simultaneously accessing the variables A[1], A[2], A[3], and A[4], respectively, via the function `incr`. With the Lock method, there is no parallelism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

The second case is shown in Fig. 3 where the cores are accessing the same data element A[1]. In this case, the operations are performed sequentially for both Lock and TM, since there are true conflicts among the cores. However, the main drawback of TM is in the fact that TM wastes energy for its speculative execution and consumes additional energy for

1. In the rest of the paper, the term Lock refers to the traditional HW-lock.

```
// Same indexes at time T
   core 0: ptr[index] == A[1];
   core 1: ptr[index] == A[1];
   core 2: ptr[index] == A[1];
   core 3: ptr[index] == A[1];
```

(a) Function calls to the same data



(b) Operation

Fig. 3. Parallel processing with same data.

rollback [shaded box in Fig. 3(b)]. This is an obvious problem considering the requirement of low power consumption in embedded systems although the approach may provide more parallelism than the lock method as shown in the above paragraph.

3.2 Required Features

We have presented the drawbacks of the Lock and TM mechanisms. From the analysis, the ideal operation is the elimination of speculative execution while exploiting parallelism as much as possible. The proposed *C–Lock* system is developed to achieve this purpose.

As shown in Fig. 2, *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), *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, *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 re-designed to be run on TM systems. On the contrary, conventional lock-based programs can be easily transformed for execution with the *C–Lock* approach since the proposed scheme prohibits speculative execution. In addition, *C–Lock* prevents deadlock problem. The mechanism is described in Section 4.2.1.

4 *C-Lock*

In this section, we will describe the concept, implementation, operation, and usage of *C–Lock*. First, we will give a brief

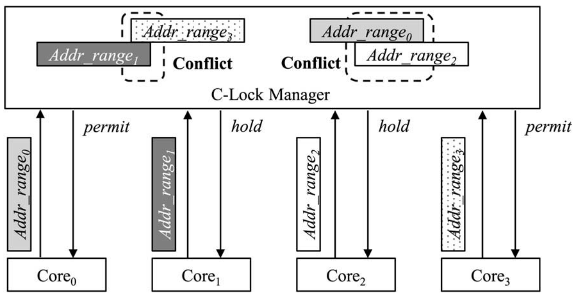
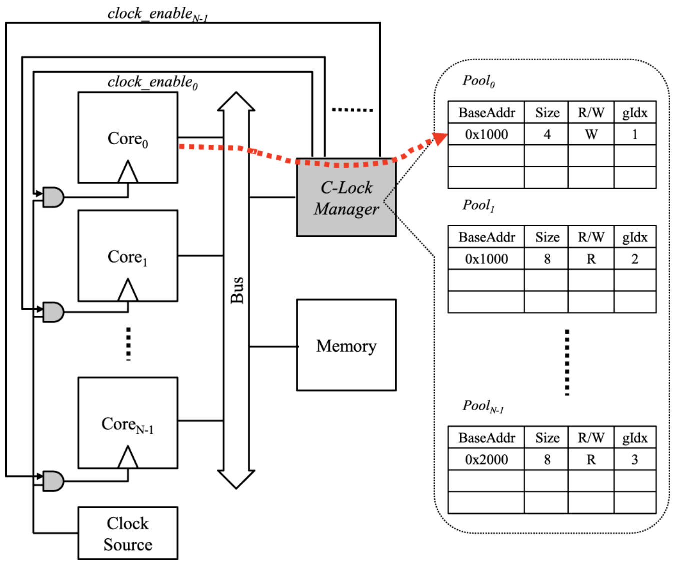


Fig. 4. Concept of $C - Lock$ mechanism.



BaseAddr	Size	R/W	gIdx
0x1000	4	W	1

BaseAddr	Size	R/W	gIdx
0x1000	8	R	2

BaseAddr	Size	R/W	gIdx
0x2000	8	R	3

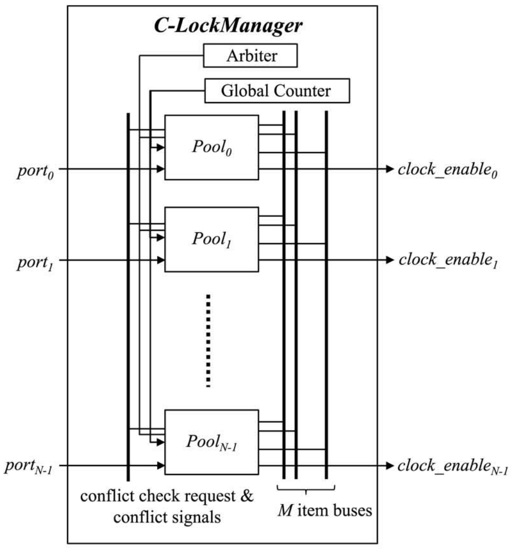
Fig. 5. Top-level architecture of $C - Lock$.

overview of *C–Lock* in Section 4.1. The technical details and implementations of the *C–Lock* hardware will be presented in Section 4.2. Based on the hardware architecture of *C–Lock*, the detailed operation of *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 *C–Lock*.

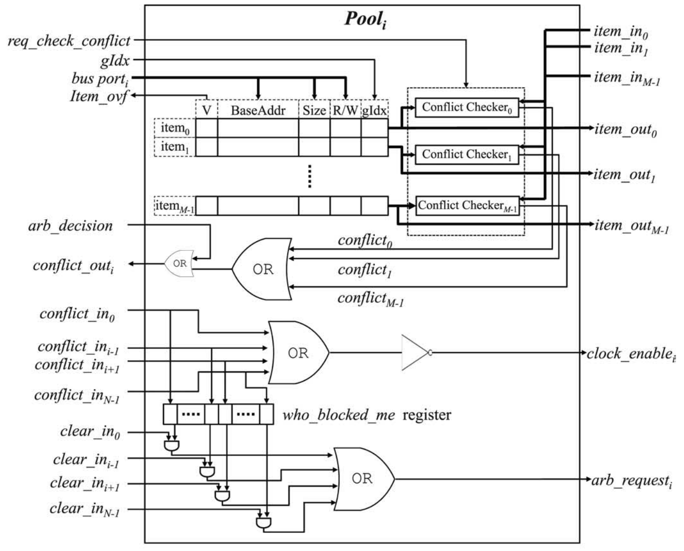
4.1 Overview

The main idea of the *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; $Addr\_range_0$ to $Addr\_range_3$ in the figure. After that, the centralized peripheral *C–Lock Manager* decides whether the ranges overlap or not. If there is an overlap, only one among the cores that cause conflict is permitted to run while the others are stalled with clock-gating until the former ends the execution.

The overall architecture of *C–Lock* is shown in Fig. 5. Two major modifications from the traditional lock schemes are needed to support *C–Lock*; on the hardware side, an additional peripheral called *C–LockManager* is added to the



(a) $C\text{-}LockManager$ internal architecture



(b) The architecture of each $Pool_i$

Fig. 6. $C\text{–}LockManager$ implementation.

system. $C\text{–}LockManager$ is the key component of *C–Lock* which is in charge of detecting true conflicts among the accesses to the shared data, and controlling clock-gating of the cores (Section 4.2).

On the software side, each core is in charge of setting the necessary information to $C\text{–}LockManager$, which includes base address, size, and type of the data it intends to access. When this information is set, the core is allowed to attempt its atomic operation by notifying $C\text{–}LockManager$.

In the next step, $C\text{–}LockManager$ initiates the conflict detection routine and, in case of a conflict, grants permission to only one of the cores while gating the other cores which intend to access the data. Note that multiple cores can get the permissions if the accesses are not involved in any true conflict. After the core which has obtained the permission completes its atomic access, it notifies $C\text{–}LockManager$ to release the permission. This command also triggers *C–Lock* $Manager$'s conflict detection routine. $C\text{–}LockManager$ gives permission to another core by de-asserting the corresponding clock-gating signal.

4.2 $C\text{–}LockManager$ Implementation

The details of $C\text{–}LockManager$ are presented in this subsection. In the following, we assume that there are N cores in the processor, and that each core can record $MItem$s with $C\text{–}LockManager$. $Item$ refers to a storage that contains information for checking true conflicts with the accesses of other cores. One $Item$ consists of the following fields:

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

The internal architecture of $C\text{–}LockManager$ is shown in Fig. 6(a); it is composed of $NPool$s, $MItem$ *busses*, a *global counter*, an *arbiter*, and a couple of signals among the $Pool$s for the purpose of detecting conflicts (signals for requesting conflict check to the other $Pool$s, and for responding to the requests).

4.2.1 Operation of $C\text{–}LockManager$

We first explain in detail the structure of the handling logic of the $Pool$ which is the main part of $C\text{–}LockManager$: it consists of $MItem$ entries and conflict checking and clock-gating logics. The microarchitecture of $Pool$ is shown in Fig. 6(b). Each Core_i initiates the *C–Lock* operation by recording the access information to the corresponding $Pool$ (i.e., $Pool_i$ in $C\text{–}LockManager$). This operation is done by calling the function `ADD_ITEM`, and is a typical bus write operation through a dedicated bus port (port_i) (details are shown later in Section 4.3). Each core can register at most $MItem$s. 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 $C\text{–}LockManager$ 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

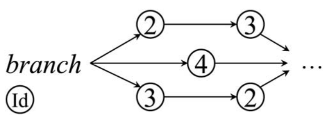


Fig. 7. Lock dependency graph.

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 1 the $arb\_decision$ signal in Fig. 6(b).

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 $MItem$s to the $itembuses$ and requests the other $Pool$s to check for conflicts by comparing the broadcasted $Item$s and their own registered $Item$s. Immediately after, the conflict checking process is performed in the other $Pool$s. The major part of this process is done by the *conflict checker* [shown in Fig. 6(b)]. A conflict checker is dedicated to an $Item$ entry and checks whether any of the M broadcasted $Item$s 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 $Item$s (*rem* stands for remote). Then, $Item_{loc}$ and $Item_{rem}$ have true conflicts if the following conditions are simultaneously present:

- Both $Item$s 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 $Item$s 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 $Item$s are in conflict with the $Item$s in this $Pool$. The signal is OR-gated with the $arb\_decision$ signal to output the final $conflict\_out$ signal.

After that, the $Pool$ which requested conflict checks from the other $Pool$s gathers the results by watching the $conflict\_in$ signals in Fig. 6(b). If any of the other $Pool$s 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 blocking $Pool$s being cleared and reattempt its access. This can effectively avoid the blocked $Pool$s watching the activities from all the other cores. When no conflicts are reported from the other $Pool$s, the core keeps running and executes the atomic access for the registered $Item$s.

As shown in Fig. 6(b), each $Pool$ has its own $Item$s and the number of $Item$s is fixed as M in the proposed scheme. Therefore, if the number of requested $Item$s to be registered is larger than M, some addresses cannot be registered. To solve this problem, $Pool$ is designed to send the $Item\_ovf$ 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 1 to stall the execution of the core which issues $Item\_ovf$ 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 $Pool$s and the *arbiter*. As shown in Fig. 6(b), the conflict checker is dedicated for each $Item$. Therefore, additional control is not required for address comparison. For example, if the $Item$s are shared among the $Pool$s, many multiplexers are needed to connect the all $Item$s 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 $\texttt{END\_C} - \texttt{LOCK}$ 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 $Pool$s 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–Lock Manager* can vary from one to many bus clock cycles. If the data length of an $Item$ is l (excluding valid field because it is automatically set and reset by *C–LockManager*), the number of $Item$s to be registered is m, and the bus width is b, the number of cycles needed for a core to transfer the $Item$s to *C–LockManager* is simply $\lceil (m \times l)/b \rceil$. In our implementation, we assume that setting one $Item$ (including $BaseAddr$, $Size$, and $\overline{R}/W$ 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 $Item$s occurs at the next cycle, while the conflict check process is done in the other $Pool$s 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 $Item$s 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 90 nm logical

2. available at www.synopsys.com

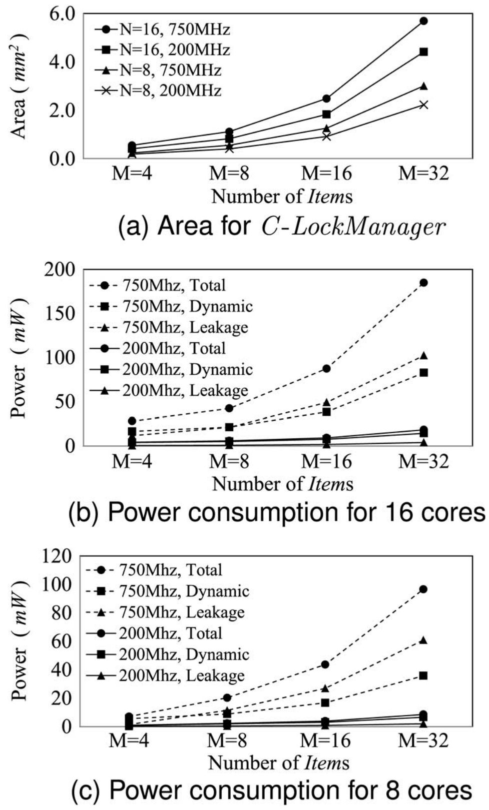


Fig. 8. Area and power consumptions of various $C\text{–}LockManager$ implementations in 90 nm process technology.

```
// bitfield for regCtrl
#define bfBEGIN (1<<0)
#define bfEND    (1<<1)

// flag for regRW
#define bREAD    (false)
#define bWRITE   (true)

// ADD_ITEM macro
#define ADD_ITEM(coreIdx,base,size,rw) \
    {core[coreIdx]->reg = {base,size,rw};}

// BEGIN_C–LOCK macro
#define BEGIN_C–LOCK(coreIdx, C–LockIdx) \
    {core[coreIdx]->reg = C–LockIdx;} \
    {core[coreIdx]->regCtrl = bfBEGIN;}

// END_C–LOCK macro
#define END_C–LOCK(coreIdx, C–LockIdx) \
    {core[coreIdx]->regCtrl = bfEND;} \
    {core[coreIdx]->reg = C–LockIdx;}

char X;
double Y;

void foo() {
    ...
    ADD_ITEM(coreId, &X, sizeof(X),bREAD); //[S1]
    ADD_ITEM(coreId, &Y, sizeof(Y),bWRITE);//[S1]
    BEGIN_C–LOCK(coreId, C–LockId);        //[S2]
    // data process with X and Y;          //[S3]
    END_C–LOCK(coreId, C–LockId);          //[S4]
    ...
}
```

Fig. 9. Macros for $C\text{–}Lock$.

and physical technology libraries were used in the implementation. We implemented various versions of $C\text{–}LockManager$ considering the following conditions; the type aiming at high speed (750 MHz), and the other at optimizing for area and power for a 200 MHz clock frequency (200 MHz is the bus clock speed used in our experiments).

The results show the area overhead and power consumption of $C\text{–}LockManager$ for the various combinations of the number of cores (N) and the number of $Item$ entries (M). As shown in Fig. 8(a), 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. 8(b) and (c). 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 $C\text{–}Lock$; 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 $C\text{–}LockManager$ such that it can separately control the clocks of the frond-end part and the back-end part of each core.

4.3 Software-Hardware Interaction

In order to take advantage of $C\text{–}Lock$, 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 $C\text{–}LockManager$. 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

TABLE 1
Reg_i in $C - LockManager$

Usage	Name	Discription
software	regBase	object base address
	regSize	object size
	regRW	object $\overline{R}/W$
	regC-LockIdx	critical section Id
	regCtrl	begin / end control
internal	isLocked	locking status
	isBusy	running status
	ItemCnt	item count in pool
	LockCoreIdx	core index of locked

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 $C\text{–}Lock$ (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 $C\text{–}Lock$ 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 $C\text{–}LockManager$, when the corresponding $Pool$ successfully registers the incoming information to its $Item$ entry ([$H1$]). After that, the system software starts the $C\text{–}Lock$ 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 $C\text{–}Lock$ $Manager$. From BEGIN_C − LOCK call, $C\text{–}LockManager$ 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 $C\text{–}Lock$ $Manager$ 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 $C\text{–}LockManager$ ([$H5$]), 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 $C\text{–}Lock$ 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 $C\text{–}LockManager$ 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 $C\text{–}LockManager$ ([$H7$]), but also notifies the other $Pool$s that the shared region has been released ([$H8$]). Then, the $Pool$s 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 $C\text{–}Lock$ Operation

In this subsection, we will explain the operation of $C\text{–}Lock$ 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 $Item$s. 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 T_0, $Core_0$, $Core_1$ and $Core_2$ record the $Item$s to the $Pool$s and request their atomic accesses ($Core_3$ continues

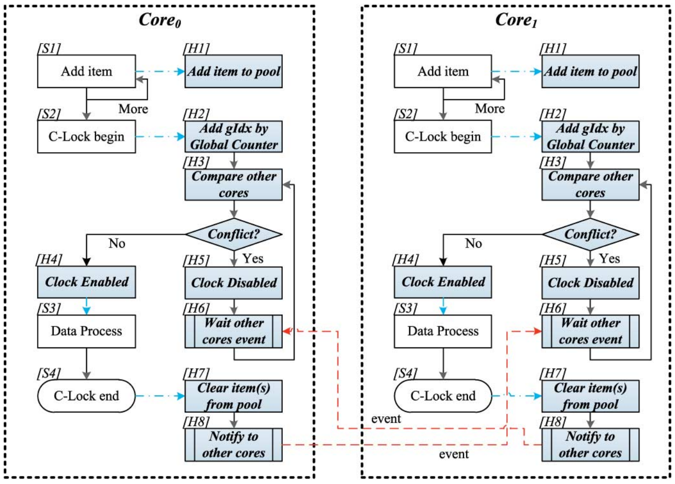


Fig. 10. Software-hardware interaction of $C - Lock$ operation.

performing its local operations). Since the address range $Core_0$ requests do not overlap with others, it is granted access (area labeled "G") and initiate its atomic operation (foo()) without its clock being gated. Also, $Core_1$ and $Core_2$ receive the grant for the operations (bar()) since there is only a *false dependency* between them, i.e., both are read operations.

At T_1, $Core_3$ requests an atomic operation with C − LockId 6. Since we have assumed that all $Item$ entries are empty before T_0, the request can be granted if the $C\text{–}Lock$ system considers only the registered $Item$s at T_0. However, as described in Section 4.2.1, nested lock is informed to the *arbiter*. Therefore, the request of $Core_3$ at C − LockId 6 is denied since C − LockId 5 is already acquired by $Core_0$. As a result, the status of $Core_3$ is changed to Idle. If there is no consideration for the nested lock, the request of Core3 at T_1 will be granted and the status of $Core_0$ will be changed to idle at T_2 due to the overlapped address. In this case, a deadlock may happen if $Core_3$ requests an atomic operation at T_3 as the gray colored text indicates; a conflict occurs due to the $Item$s registered by $Core_0$ at T_0. Also at T_1, $Core_1$ releases its ownership of the address range 304-308 (area labeled "R").

At T_2, $Core_0$ requests the atomic access in a nested manner and it is granted since there is no overlapped address. However, $Core_1$ is denied (area labeled "D") since the request has a true data conflict with the atomic access being held by $Core_0$. As $Core_0$ completes its atomic operation and releases its all ownership of inner nest and outter nest at T_3 and T_4, $C\text{–}LockManager$ performs an arbitration again and grant access to $Core_3$ by enabling the clock.

5 Experiments

In this section we evaluate our proposed $C\text{–}Lock$ 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

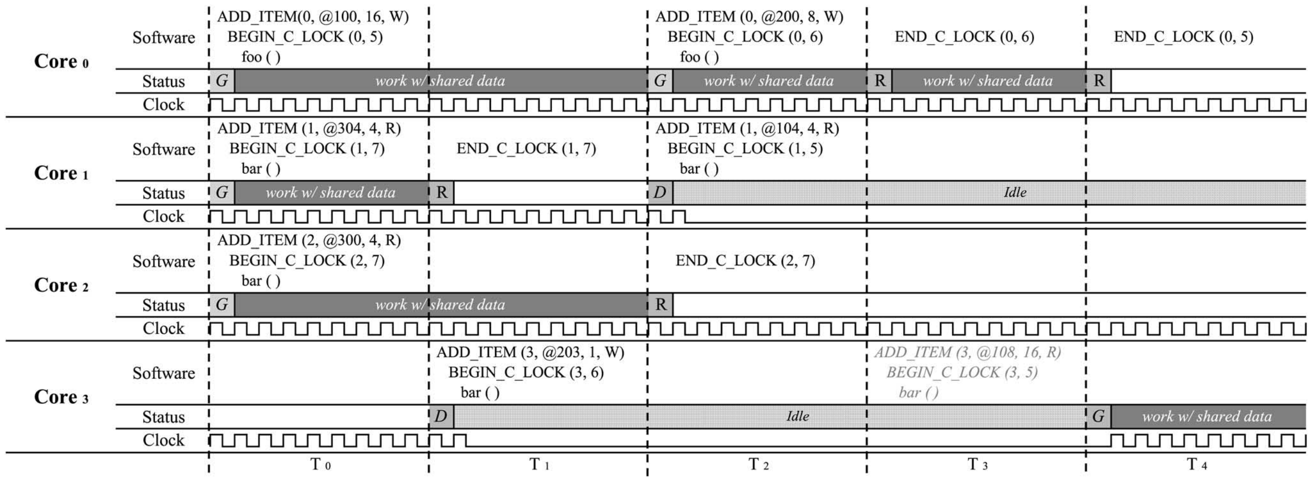


Fig. 11. Description of $C - LockManager$ process.

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. 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 at.* [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 $C\text{–}LockManager$ 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.

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

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%)

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 does 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

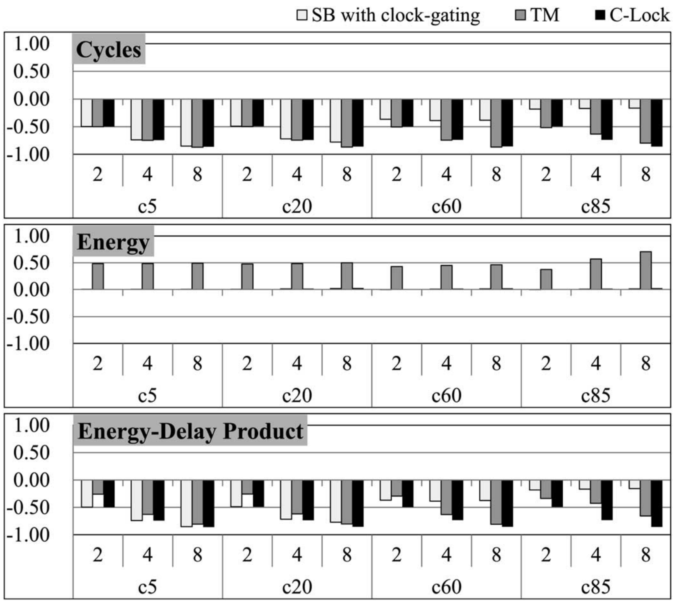


Fig. 12. Comparison results for the matrix benchmarks, using 2, 4, and 8 cores. The values are normalized to the results of the baseline system with 1 core, and substracted by 1.

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

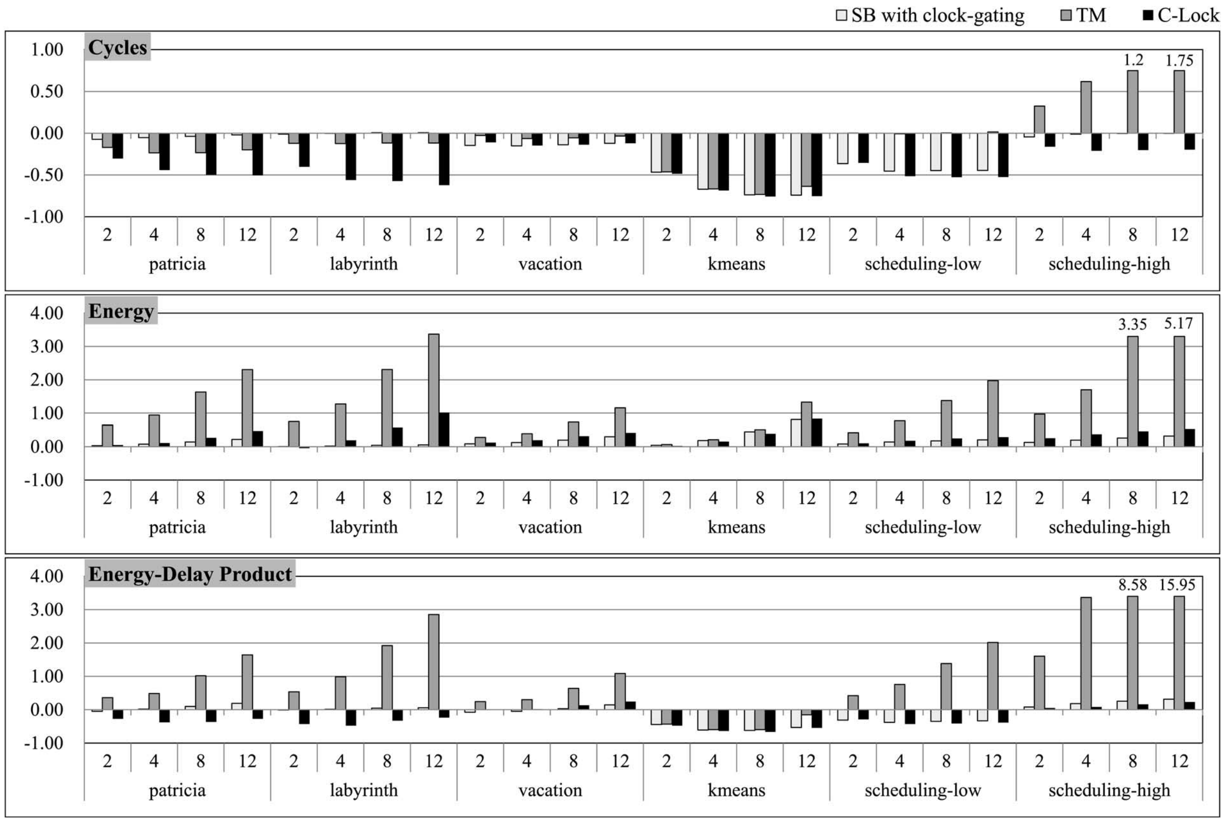


Fig. 13. Comparison results for the applications, using 2, 4, 8, and 12 cores.

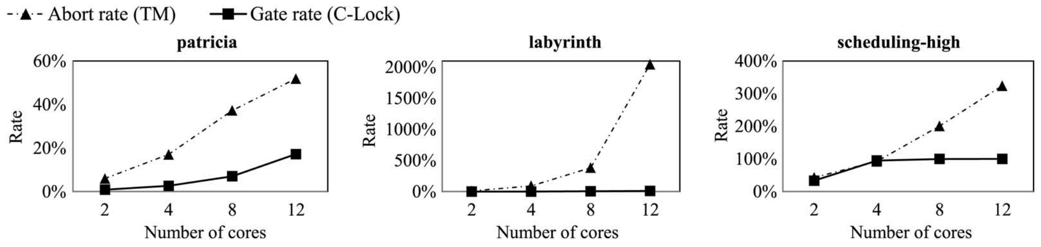


Fig. 14. Abort rate(TM) / gate rate($C-Lock$) of complex applications as cores.

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–low`, 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[3] 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 Conclusion

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 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–Lock Manager*, 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.

Acknowledgment

This work is partly supported by the National Science Foundation under Grant 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) (2009-0077326, 2010-0013202, and 2010-0025423), by the Visiting Professors Program through NRF funded by the MEST (2012S1A2A1A01031420), and by the Mid-career Researcher Program through NRF grant funded by the MEST (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.

3. $\frac{N_{clock-gated}}{N_{crit-section}}$, where $N_{clock-gated}$ represents how many times the core is clock-gated and $N_{crit-section}$ represents how many times the core enters the critical section.

References

[1] D. Geer, “Chip makers turn to multicore processors,” *Computer*, vol. 38, no. 5, pp. 11–13, May 2005.
[2] M. Levy and T. Conte, “Embedded multicore processors and systems,” *IEEE Micro*, vol. 29, no. 3, pp. 7–9, May/Jun. 2009.
[3] M. Hill and M. Marty, “Amdahl’s law in the multicore era,” *Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008.
[4] M. Herlihy and J. E. B. Moss, “Transactional memory: Architectural support for lock-free data structures,” in *Proc. 20th Int. Symp. Comput. Archit.*, 1993, pp. 289–300.
[5] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, and G. Nicolescu, “Parallel programming models for a multi-processor SoC platform applied to high-speed traffic management,” in *Proc. 2nd IEEE/ACM/IFIP Int. Conf. Hardware/Softw. Codesign Syst. Synthesis (CODES+ISSS’04)*, 2004, pp. 48–53.
[6] R. Rajwar and J. Goodman, “Speculative lock elision: Enabling highly concurrent multithreaded execution,” in *Proc. 34th Annu. ACM/IEEE Int. Symp. Microarchit.*, 2001, pp. 294–305.
[7] R. Rajwar and J. Goodman, “Transactional lock-free execution of lock-based programs,” in *Proc. 10th Int. Conf. Archit. Support Program. Languages Oper. Syst.*, 2002, pp. 5–17.
[8] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, “Power/performance hardware optimization for synchronization intensive applications in MPSOCs,” in *Proc. Des. Autom. Test Eur.*, 2006, pp. 606–611.
[9] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, “Efficient synchronization for embedded on-chip multiprocessors,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 10, pp. 1049–1062, 2006.
[10] C. Yu and P. Petrov, “Distributed and low-power synchronization architecture for embedded multiprocessors,” in *Proc. 6th IEEE/ACM/IFIP Int. Conf. Hardware/Softw. Codesign Syst. Synthesis*, 2008, pp. 73–78.
[11] J. Li, J. F. Martinez, and M. C. Huang, “The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors,” in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, 2004, vol. 10, pp. 14–23.
[12] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. J. Irwin, “Exploiting barriers to optimize power consumption of CMPs,” in *Proc. 19th IEEE Int. Symp. Parallel Distrib. Process.*, 2005, p. 5.
[13] O. Golubeva, M. Loghi, and M. Poncino, “On the energy efficiency of synchronization primitives for shared-memory single-chip multiprocessors,” in *Proc. ACM Great Lakes Symp. VLSI*, 2007, pp. 489–492.
[14] C. Ferri, A. Viescas, T. Moreshet, R. Bahar, and M. Herlihy, “Energy efficient synchronization techniques for embedded architectures,” in *Proc. 18th ACM Great Lakes Symp. VLSI*, 2008, pp. 435–440.
[15] C. Ferri, R. Bahar, M. Loghi, and M. Poncino, “Energy-optimal synchronization primitives for single-chip multi-processors,” in *Proc. 19th ACM Great Lakes Symp. VLSI*, 2009, pp. 141–144.
[16] A. Kägi, D. Burger, and J. R. Goodman, “Efficient synchronization: Let them eat QOLB,” in *Proc. 24th Int. Symp. Comput. Archit.*, 1997, pp. 170–180.
[17] R. Rajwar, A. Kagi, and J. Goodman, “Improving the throughput of synchronization by insertion of delays,” in *Proc. 6th Int. Symp. High-Perform. Comput. Archit.*, 2000, pp. 168–179.
[18] N. Shavit and D. Touitou, “Software transactional memory,” in *Proc. 14th Symp. Principles Distrib. Comput.*, 1995, pp. 204–213.
[19] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III, “Software transactional memory for dynamic-sized data structures,” in *Proc. 22nd Symp. Principles Distrib. Comput.*, 2003, pp. 92–101.
[20] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, “Transactional memory coherence and consistency,” *ACM SIGARCH Comput. Archit. News*, vol. 32, no. 2, 2004, p. 102.
[21] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie, “Unbounded transactional memory,” in *Proc. 11th Int. Symp. High-Perform. Comput. Archit.*, 2005, pp. 316–327.
[22] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood, “LogTM: Log-based transactional memory,” in *Proc. 12th Int. Symp. High-Perform. Comput. Archit.*, 2006, pp. 254–265.
[23] T. Moreshet, R. I. Bahar, and M. Herlihy, “Energy reduction in multiprocessor systems using transactional memory,” in *Proc. Int. Symp. Low Power Electron. Des.*, 2005, pp. 331–334.
[24] C. Ferri, T. Moreshet, R. Bahar, L. Benini, and M. Herlihy, “A hardware/software framework for supporting transactional memory in a MPSoC environment,” *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 47–54, 2007.
[25] C. Ferri, S. Wood, T. Moreshet, R. I. Bahar, and M. Herlihy, “Embedded-tm: Energy and complexity-effective hardware transactional memory for embedded multicore systems,” *J. Parallel Distrib. Comput.*, vol. 70, no. 10, pp. 1042–1052, 2010.

[26] S. Sanyal, S. Roy, A. Cristal, O. S. Unsal, and M. Valero, "Clock gate on abort: Towards energy-efficient hardware transactional memory," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2009, pp. 1–8.

[27] T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis, "Adaptive locks: Combining transactions and locks for efficient concurrency," *J. Parallel Distrib. Comput.*, vol. 70, no. 10, pp. 1009–1023, 2010.

[28] B. D. Carlstrom, J. Chung, H. Chafi, A. McDonald, C. C. Minh, L. Hammond, C. Kozyrakis, and K. Olukotun, "Executing Java programs with transactional memory," *Sci. Comput. Programm.*, vol. 63, no. 2, pp. 111–129, 2006.

[29] M. Martin, C. Blundell, and E. Lewis, "Subtleties of transactional memory atomicity semantics," *IEEE Comput. Archit. Lett.*, vol. 5, no. 2, pp. 17–20, Jul.–Dec. 2006.

[30] S. Akhter and J. Roberts, *Multi-Core Programming*. Intel Press, Intel Corporation, 2111 NE 25th Avenue, JF3-330, Hillsboro, OR 97124-5961, 2006.

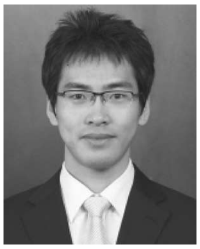
[31] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "Mparm: Exploring the Multi-Processor SoC Design Space with SystemC," *J. VLSI Signal Process.*, vol. 41, no. 2, pp. 169–182, 2005.

[32] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *Proc. 10th ACM Symp. Parallel Algorithms Archit.*, 1998, pp. 119–129.

[33] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *Proc. IEEE Int. Symp. Workload Char.*, 2008, pp. 35–46.

[34] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. IEEE Int. Workshop Workload Char.*, 2001, pp. 3–14.

[35] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.



Seung Hun Kim received the BS and MS degrees in electrical and electronic engineering from Yonsei University, Seoul, Korea, in 2009 and 2011, respectively. He is currently PhD 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 from Yonsei University, Seoul, Korea, in 2003 and 2012, respectively. He is currently a senior engineer with Samsung Electronics, Suwon, Korea. His research interests include sytem-on-chip architecture.

Minje Jun (M'08) received the BS, MS, and PhD degrees in electrical and electronic engineering from Yonsei University, 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 BS degree in electrical and electronic engineering from Yonsei University, Seoul, Korea, in 2010, where he is currently working toward the PhD degree in electrical and electronic engineering. His research interests include low power design and memory architecture.

Won Woo Ro received the BS degree in electrical engineering from Yonsei University, Seoul, Korea, in 1996, and the MS and PhD degrees in electrical engineering from the University of Southern California, Los Angeles, in 1999 and 2004, respectively. He worked as a research scientist in the Electrical Engineering and Computer Science Department, University of California, Irvine. He currently works as an associate professor with the School of Electrical and Electronic Engineering, Yonsei University, Seoul, Korea. Prior to joining Yonsei University, he has worked as an assistant professor in the Department of Electrical and Computer Engineering, California State University, Northridge. His industry experience also includes a college internship at Apple Computer, Inc., and a contract software engineer in ARM, Inc. His current research interests include high-performance microprocessor design, compiler optimization, and embedded system designs.

Eui-Young Chung (SM'99–M'06) received the BS and MS degrees in electronics and computer engineering from Korea University, Seoul, Korea, in 1988 and 1990, respectively, and the PhD degree in electrical engineering from Stanford University, Stanford, CA, in 2002. From 1990 to 2005, he was a principal engineer with SoC R&D Center, Samsung Electronics, Yongin, Korea. He is currently a professor with the School of Electrical and Electronic Engineering, Yonsei University, Seoul. His research interests include system architecture and VLSI design, including all aspects of computer-aided design with the special emphasis on low-power applications and flash memory applications.

Jean-Luc Gaudiot received the Diplôme d'Ingénieur from the École Supérieure d'Ingénieurs en Electrotechnique et Electronique, Paris, France, in 1976, and the MS and PhD degrees in computer science from the University of California, Los Angeles, in 1977 and 1982, respectively. He is currently a professor and chair of the Electrical and Computer Engineering Department, the University of California, Irvine. Prior to joining UCI in January 2002, he was a professor of electrical engineering with the University of Southern California since 1982, where he served as director of the Computer Engineering Division for 3 years. He has also done microprocessor systems design at Teledyne Controls, Santa Monica, California, in 1979–1980, and research in innovative architectures at the TRW Technology Research Center, El Segundo, California, in 1980–1982. He consults for a number of companies involved in the design of high-performance computer architectures. His research interests include multithreaded architectures, fault-tolerant multi-processors, and implementation of reconfigurable architectures. He has published over 200 journal and conference papers. His research has been sponsored by NSF, DoE, and DARPA, as well as a number of industrial organizations. In January 2006, he became the first editor-in-chief of *IEEE Computer Architecture Letters*, a new publication of the IEEE Computer Society, which he helped found to the end of facilitating short, fast turnaround of fundamental ideas in the Computer Architecture domain. From 1999 to 2002, he was the editor-in-chief of the *IEEE Transactions on Computers*. In June 2001, he was elected chair of the IEEE Technical Committee on Computer Architecture, and re-elected in June 2003 for a second 2-year term. He is a member of the ACM and of the ACM SIGARCH. He has also chaired the IFIP Working Group 10.3 (Concurrent Systems). He is one of three founders of PACT, the ACM/IEEE/IFIP Conference on Parallel Architectures and Compilation Techniques, and served as its first program chair in 1993, and again in 1995. He has also served as program chair of the 1993 Symposium on Parallel and Distributed Processing, HPCA-5 (1999 High Performance Computer Architecture), the 16th Symposium on Computer Architecture and High Performance Computing, Foz do Iguacu, Brazil, the 2004 ACM International Conference on Computing Frontiers, and the 2005 International Parallel and Distributed Processing Symposium. He was elevated to the rank of AAAS Fellow in 2007.