Distributed and Low-Power Synchronization Architecture for Embedded Multiprocessors

Chenjie Yu
University of Maryland
College Park, USA
purety@umd.edu

Peter Petrov
University of Maryland
College Park, USA
ppetrov@umd.edu

ABSTRACT

In this paper we present a framework for a distributed and very low-cost implementation of synchronization controllers and protocols for embedded multiprocessors. The proposed architecture effectively implements the queued-lock semantics in a completely distributed way. The proposed approach to synchronization implementation not only completely eliminates the overwhelming bus contention traffic when multiple cores compete for a synchronization variable, but also achieves very high energy efficiency as the local synchronization controller can efficiently determine, without any bus transactions or local cache spinning, the exact timing of when the lock is made available to the local processor. Application-specific information regarding synchronization variables in the local task is exploited in implementing the distributed synchronization protocol. The local synchronization controllers enable the system software or the thread library to implement various low-power policies, such as disabling the cache accesses or even completely powering down the local processor while waiting for a synchronization variable.

Categories and Subject Descriptors: C.1.2 [Hardware]: Multiple Data Stream Architectures (Multiprocessors); C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems

General Terms: Algorithms, Design, Experimentation

1. INTRODUCTION

The ever increasing demands of many modern applications for consolidated functionality, including multimedia, data, communication, security and many other capabilities coupled with increased integration densities have resulted in the adoption and utilization of embedded multiprocessor implementation platforms. Such application domains include smart phones, portable media players, navigation devices, and many others. While trying to meet the performance requirements of such applications, embedded multiprocessor systems have encountered challenges that are specific to these architectures and application domains, such as *energy efficiency* considerations in battery-based devices and *real-time performance* requirements for many time-critical tasks. These domain specific requirements have resulted in new lines of research efforts aiming at adopting and optimizing general-purpose hardware and software organizations to the low-power and real-time requirements of the modern embedded applications.

Multiprocessor architectures for the embedded domain have given rise to some unique problems not present in uni-processor embedded systems, such as inter-core communication, synchronization, and data/code sharing. Furthermore, the typical availability of application-specific information present at design time has enabled a new set of optimization strategies that aim at capturing and exploiting this information at run-time, in order to achieve energy-efficiency and time-deterministic performance for the particular application program or set of tasks to be executed. One such problem that arises in embedded multiprocessors is the typical need for synchronization among the threads executing on the processor cores. Such functionality is needed in many instances where execution progress, data sharing, and communication between the parallel threads need to be synchronized. It is usually the responsibility of the software developer (or in recent developments of parallel compilation environments, the compiler) to properly use the set of available synchronization operations in order to ensure deterministic event order and proper communication between the threads.

Several well known synchronization primitives are usually made available to the software developer/compiler by system libraries or directly by the operating system. Frequently used synchronization primitives include *locks*, *barriers*, *semaphores*, and *monitors*. At hardware level, however, various implementation approaches are being used based on the underlying hardware architecture. Their implementation is often based on certain atomic operations provided by the hardware. Conventional examples of such atomic operation implementations include the pair of *load-linked* and *store-conditional* instructions, or an atomic *test-and-set* instruction. Such atomic primitives ensure that a software implementation of a synchronization primitive would access a certain *synchronization variable* and subsequently modify it accordingly, without the possibility of this variable being modified by another processor core.

While such implementation provides a general-purpose support for a comprehensive set of synchronization primitives, its generality comes with the price of significant power and inter-core communication overheads. It has been known that such synchronization can results in severe bus traffic contention when multiple processors compete for the same synchronization variable [1]. In the case when no local caching is available (or when no cache coherence mechanism exists) the processors need to poll the synchronization variable, thus polluting the interconnect to memory with a large amount of traffic and also expending a significant amount of power. When coherent caches are present, the polling is executed locally by the *spin-lock* primitive, which, however, does not resolve

CODES+ISSS'08, October 19–24, 2008, Atlanta, Georgia, USA.

the power problem; significant bus contention ensues too when a processor releases the synchronization variable, which leads to invalidations in all remote caches. All these problems stem from the need that all the processors compete to read and modify a shared variable (the synchronization variable) in an atomic way. Recently several research projects have proposed centralized solutions [2], where a special hardware controller is introduced that keeps track of the participating tasks and communicates with them. In such a solution, however, all the communication in acquiring, releasing, and granting the synchronization variable is routed through the controller. This results in a performance overhead as the controller needs to compete to access to the shared bus which impacts the time of acquiring a lock and the overall system performance. An alternative would be to create star-like dedicated communication lines to each processor, albeit with significant price in silicon area and routing overheads.

In this paper, we offer a completely distributed synchronization architecture to address the aforementioned problems. Locally to each processor, a light-weight hardware controller is introduced, which captures the synchronization variables of interest to the local processor. In this way, each processor participates in a completely decentralized and distributed protocol of acquiring and releasing a synchronization variable. Each such local controller monitors the bus for "acquires" and "releases" of synchronization variables of local interest and maintains a precise state of the global status of each variable. The proposed organization requires no atomic operations for accessing and modifying main memory as it relies on the inherent serialization nature of the memory bus. The end results is that the semantic of *queued locks* is implemented in a completely distributed manner with a **near zero-latency lock acquisition and release**, while providing for precise and **fine-grained power management** to eliminate a large fraction of the energy wasted while the processor waits on synchronization.

2. RELATED WORK

A large body of research work exists related to the synchronization problems in multiprocessor systems. The performance impact of synchronization due to bus contention and global communication has been recognized from various perspectives [1, 3, 4, 5]. The impact on power has been analyzed and addressed in [6, 7]

In [5], the authors propose a light-weight distributed synchronization method in point-to-point communication applications. The approach encodes the global data dependencies between two processors directly in their memory accesses. In [4] and [8], the authors propose the Lock Cache organization. The lock cache mechanism implements synchronization in a dedicated and centralized hardware controller. With task preemption support from the RTOS, the Lock Cache achieves good performance for *database like applications*. A light weight barrier-based parallelization support for non-cache-coherent MPSoC platforms has been proposed in [3]. A cost-efficient barrier implementation for the specific targeted architecture is outlined. In [6], the authors propose the *thrifty* barrier mechanisms addressing the power problem in general-purpose multiprocessor systems. By carefully predicting and monitoring barrier stall times, processors are placed in low-power modes and speculatively resumed when the barrier release is predicted.

The synchronization architecture proposed in this paper is completely distributed and achieves very fast lock acquisition, eliminates bus contention traffic, and enables very fine-grained power management at each processor node. It is suitable for multithreading applications with unbalanced execution time, as well as applications executing a large number and short critical section regions, which traditionally have incurred very high performance overhead due to synchronization.

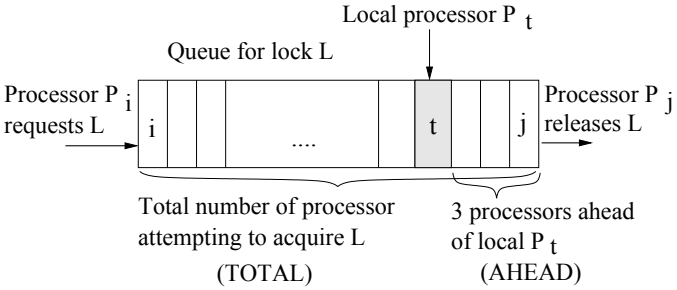

Figure 1: Distributed lock queue information

3. FUNCTIONAL OVERVIEW

Conventional synchronization implementations rely on atomic operations to access and modify memory. Such a support enables the processors to compete for the exclusive access to a synchronization variable and setting it up as "acquired". The processors compete for such an access and whoever succeeds is "granted" the synchronization variable, while the other processors continue their attempts. Such atomic mechanisms in accessing memory locations are used to build various high-level primitives, such as (spin) locks, barriers, monitors, etc. While such synchronization implementations are general-purpose and impose small hardware and ISA constraints (some assume coherent caches), they can be extremely inefficient in terms of both performance and power consumption.

Since the processors have no knowledge as of the global status of the synchronization variables, they all compete for the access to the shared synchronization variable by overwhelming the memory bus with transactions. Even in the presence of coherent caches, at the moment a synchronization variable is released, all the processors waiting for it enter another competing cycle, which results in a burst of bus traffic. Furthermore, when a processor attempts to acquire a synchronization variable when it is not available, it has to keep polling on it and generating bus traffic or in the case of coherent caches, to continuously read it until its remote invalidation. This can be extremely energy inefficient. A centralized solution can alleviate the polling/spinning energy and the bus traffic caused by competing accesses. However, as mentioned in the previous section, it can have a negative impact on performance since lock acquisition and release will always have to be controlled by the remote controller. Such a controller will have to compete for the memory bus or use dedicated communication lines to the processors, which could result in a significant hardware area cost and chip routing complications.

The proposed distributed synchronization architecture addresses the performance, latency, and power problems of the traditional synchronization implementations. Each processor is assigned a local light-weight controller that observes the sequences of remote acquisition attempts and synchronization releases and participates in a very low-cost and efficient distributed protocol for lock acquisition and release. By monitoring the common memory bus, each local controller is able to construct a state (per synchronization variable) representing how many remote processors are waiting for the synchronization variable and have requested it *before the local processor*, as well as the total number of processors currently waiting for the variable. A remote release observed on the bus results in decrementing the number of processors waiting for the variable before the local processor. When this number reaches zero, the local processor ***immediately*** acquires the lock. After the local processor exits its synchronization section, it informs the local controller to release the lock, which results in a bus transaction informing the

```
TOTAL register:
(1)     if (RemoteAcquire)
(2)        TOTAL++;
(3)     if (RemoteRelease)
(4)        TOTAL --;

AHEAD register:
(1)     if (LocalAcquire) {
(2)        send Acquire(Lock) on the bus;
(3)        if(TOTAL == 0)
(4)           Grant local access;
(5)        else
(6)           AHEAD = TOTAL; }
(7)     if (RemoteRelease) {
(8)        if(AHEAD != 0)
(9)           AHEAD --;
(10)          if (AHEAD == 0) Grant local access; }
```

Figure 2: Local queue maintenance

remote processors that the lock has been released. In this way, the next processor in the global queue waiting for that lock will see its local state indicating that there are now zero processors in-front of it and can, thus, immediately acquire the lock.

Fundamentally, a distributed implementation of a *queue lock* mechanism is implemented as each local controller maintains the minimal amount of information needed to represent the relevant queue and the position of the local processor within it. Figure 1 illustrates the data structure that is captures by each local controller, and the minimal information needed to capture the structure with respect to the local processor. Each processor needs to maintain the information regarding its position within the queue; it also needs to be able to update this information as remote processors are requesting and releasing the lock. The relevant information about this can be captured by two variables (counters), shown in Figure 1. The register TOTAL consists with the total number of processors, which are within the queue (have requested the lock, but are still waiting for their turn). The AHEAD register captures the number of processors that are ahead in the lock queue with regards to the local processor. Clearly, when AHEAD becomes zero (down from a non-zero) it is the local processor's turn to acquire the lock.

Maintaining the TOTAL and the AHEAD registers can be achieved by only monitoring the bus for remote acquire attempts and releases to that particular lock. Clearly, acquire and releases generated by the local processor would also have to be taken into account in this process; they also need to be placed on the bus so that the remote processors can accordingly adjust their local state regarding this lock. Figure 2 illustares the functionality required to maintain the TOTAL and the AHEAD registers for each lock, as well as the detection mechanisms for when it is the local processor's turn in acquiring the lock. It is evident that this functionality can be achieved through a rather simple finite-state machine (FSM) controller and the pair of registers per synchronization variable.

The proposed protocol has the distinct advantage of *near zero-latency lock acquisition*. When the lock is globally available, the processor does not have to wait for a synchronization variable to be atomically brought back from memory or to be acquired from some remote centralized controller; the only latency incurred would be the latency for the local controller to acquire the common bus in order to send a lock acquire announcement (Step 2 in the functional description for the AHEAD register). In the case of a lock just being released by the last processor that has requested it before the local processor, the synchronization variable is acquired at the moment the local controller observes the release operation on the common bus. Such lock acquisition is in effect instantaneous as it can be triggered in the same clock cycle during which the remote release was observed. Furthermore, since there is no contention through atomic operations for the synchronization variable, it takes only two bus transactions for a task/processor to acquire and release the synchronization variable regardless of the timings of parallel requests. The acquire request may occur concurrently with remote processors. However, the common bus will serialize the requests and all the local controllers will update their TOTAL and AHEAD counters accordingly.

Since the local synchronization controllers have full information regarding the global status of the synchronization variable, they can provide for *fine-grained power management policies* on the local processor. If, for example, the local task needs to wait for the lock to be acquired (as in the case of spin-locks), the controller can gate either the entire pipeline or the most power consuming components, such as the access to caches. In this way, the processor can be switched into a low-power mode very efficiently, and resumed at the exact moment when the synchronization variable is to be acquired. In the cases of non-trivial wake-up logic, the procedure can be initiated in advance as the local controller has a complete information as of the number of tasks/processors, which are in front of the local task in the lock queue. The local synchronization controllers can also work cooperatively with the OS. The OS allocates the synchronization variable information in the controller's internal structures. Furthermore, the OS can utilize different power-saving policies and resume policies based on (and controlled by) the particular application requirements in order to make the best trade-off between performance and power.

4. SYSTEM ARCHITECTURE

The proposed distributed synchronization architecture requires a hardware support in the form of the local synchronization controllers. We refer to this hardware block as a *Distributed Synchronization Controller (DSC)*; an identical instance of it (of course, with different run-time state) is assigned locally to each processor node in the system and works independently from the other controllers by reacting on the synchronization requests/releases placed on the bus by the remote processors.

There exist several synchronization primitives that have been used in the area of parallel programing and systems. In this paper, we present how our distributed organization implements *locks* and *barriers*. Most of the other synchronization primitives can be derived from these two; thus, they can be either implemented in the DSC in a similar manner, or emulated by software in the synchronization library.

4.1 Synchronization Variable Identification

Since each lock/barrier is assigned an entry in the DSC, a mechanism is needed to identify the locks/barriers and their DSC entry. A synchronization variable is still assigned a memory location within a known page/segment of the memory address space. The page/segment identifier, which corresponds to some group of the address most significant bits is used by the DSCs to determine whether a read/write request to this location is an acquire/release operation to that particular lock. Since our approach does not require that any particular value is written or read from that memory location, its address is used for the sole purpose of broadcasting the lock acquire and releases on the bus by means of normal read and write memory transactions to that address. A group of the least significant bits of this location is used to uniquely identify the corresponding lock/barrier. It is also used to identify the DSC entry for that synchronization variable. We refer to this value as a *LockID* or a *BarrierID*. The number of bits that is actually used to represent a *LockID* is determined by the maximum number of synchroniza-

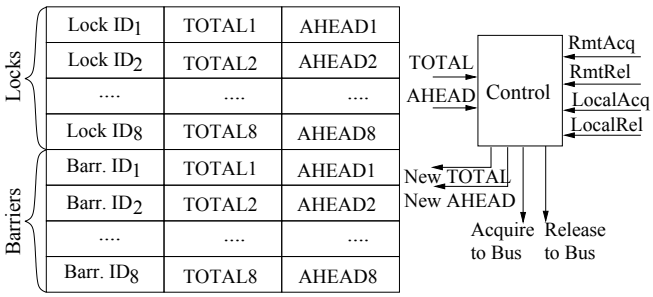

Figure 3: Synchronization Controller

tion variables that will be used in the system. In our experimental benchmarks, including Splash-2, Mediabench, and a number of signal processing applications, this number never exceeds 10, and thus we have adopted 4-bit IDs. The lock or barrier ID, corresponding to least significant bits from the address is used to lookup the DSC for that entry. One approach would be to use a CAM-based parallel lookup. Since the DSC size is very small (16 entries in our study) and the *LockID* used as a key is 4-bits wide, such a parallel lookup will be very fast and power efficient. Another implementation approach would be to use an additional mapping register/table, which will be indexed through the *LockID* and will provide the actual DSC index, if that lock/barrier is relevant to the local processor. If for some application the DSCs entries are exhausted then traditional lock implementation can be used instead for the remaining locks/barriers that cannot fit in the DSC.

Clearly, this approach to lock/barrier identification does not impose any extra requirements on the bus organization, as only traditionally supported read/write operations are used. A lock acquire operation, for instance, is modeled as a normal read from the address of that lock, while a lock release is modeled as a read operation to that location. The particular values written or read are of no importance. If the system bus supports additional control operations, special Acquire and Release transactions can be used, with a parameter corresponding to the lock-ID. In this way, the local DSCs would monitor the bus for such transactions only.

4.2 Distributed Synchronization Controller

The DSC resides at each processor and manages synchronization variables used by the tasks on this processor. It receives synchronization requests from the local processor such as ”acquire a lock”, ”release a lock”, ”enter a barrier”. The DSC also monitors the system bus for relevant relevant synchronization activities from other processors, so as to determin whether to allow the local processor to proceed or wait on synchronization events.

Figure 3 illustrates the DSC internal organization. It consists of a number of entries that correspond to synchronization variables, such as locks and barriers. For each entry, the controller registers the synchronization variable’s ID. This ID can be set when the thread/task is loaded onto this processor by the operating system or the program loader. Here we have assumed that the Lock/Barrier ID is captures in the DSC and a parallel lookup is performed. As discussed above, an alternative implementation is also possible where the identifier is mapped to a DSC index by a small set of registers.

Two counter/registers are associated with each synchronization variable. The TOTAL and AHEAD counters for each lock/barrier used by the local task are allocated and entry in the DSC. These two registers record synchronization activities both from remote processors and from the local processor that access the corresponding synchronization variable.

On a bus transaction representing a remote lock acquire or release, or on a local acquire or release, a simple control logic is used to update the registers. This controller implements the functionality described in Figure 2 for lock implementation. This controller contains two comparators and an increment/decrement module for the TOTAL register, and a decrement unit for the AHEAD register.

Lock implementation. For lock implementation, the TOTAL register counts the total number of remote processors that are waiting to acquire access to the particular synchronization variable. TOTAL is incremented when a remote processor sends an “acquire(lock)” command on the bus - for a typical bus this could be a read command to the location for that lock. Similarly, it decrements when a remote processor broadcasts a release for the lock in the form of a write to the lock address. In this way, when the local processor issues “acquire(lock)” command, it knows immediately how many processors are waiting in the queue, without the need to request information from other processors or from the shared memory. The TOTAL register is initialized to zero when the lock is created by the local thread and allocated into the DSC. When the local processor issues an acquire to that lock, TOTAL is copied into the AHEAD register. The AHEAD register indicates how many other processors are still waiting before the local processor can acquire a lock. It decreases monotonically on observing remote processors releasing the lock. When the AHEAD counter reaches zero, the DSC determines that all remote processors that were before the local in queue have released it and it is safe know for the local processor to immediately be granted the lock. As is evident from this description, the TOTAL and AHEAD registers for a particular lock throughout the system, represent the queue for that local in a consistent way. Each processor keeps track of its place in the queue locally without global cooperation with the other processors. This leaves out the necessity for a centralized control center that handles the entire system, thus making the design entirely distributed and easy to implement in terms of global chip routing and performance/power overheads. Additionally, atomic memory operations, which constitute the fundamental reason for the bus traffic contention problem, are no longer needed to construct locks and other primitives based on lock.

Barrier implementation. The distributed implementation scheme for barriers follows the same concept. The TOTAL register captures the constant that corresponds to the number of threads that must reach the barrier before it is releases. In this scheme, the value in TOTAL does not change during program execution. In the very beginning when the barrier entry is loaded into the DSC, the AHEAD register is initialized with the constant held in TOTAL (number of threads required to reach the barrier). Subsequently, when the DSC observes on the bus that another processor has reached the barrier, it decrements the AHEAD register. Similarly, when the local processor reaches the barrier, the AHEAD counter is decremented and a bus transaction is initiated by the DSC to notify the remote processors that the barrier has been reached locally. This bus transaction can be modelled, for instance, as a read transaction from the memory location of the barrier (of course, no value is expected from that memory location). Similarly as with locks, if the bus can supports extra commands, a new transaction type can be defined for reaching a barrier, which will carry the Barrier-ID needed for the remote DSCs to update their state. When AHEAD reaches zero, it means that all the threads have reached the barrier (and for all of them AHEAD will decrement to zero) and all the DSCs signal their local processor that the barrier can be released. At that moment AHEAD is loaded with the constant from TOTAL and in this way the distributed procedure is re-initialized for that barrier.

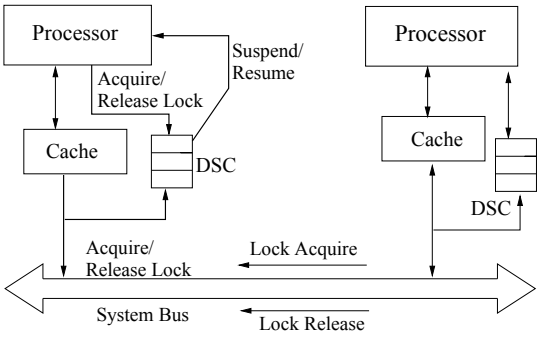

Figure 4: Overall system organization

4.3 Power Management

By having the DSC handle most synchronization operations, it becomes possible to place the local processor in various power down modes, including a complete shut-down, while the local thread is waiting on a synchronization step. The DSC continues its operation while the processor is suspended and resumes the processor execution when the synchronization conditions are met. As the local DSC controllers are exceedingly small and simple, the achieved power savings would be significant, especially in parallel multithreaded programs with unbalanced workloads. We quantify this in our experimental study.

Various power-down scheme can be explored. One approach would be to disable the accesses to the data cache for a spin-lock software implementation, where the proposed distributed synchronization architecture is implemented in an ISA-transparent way. In this case, the software implementation of the lock consists of a small loop, which iteratively attempts to read and set the lock in an atomic way. The DSC will intercept the execution of the atomic operation by matching the address with the known segment/page for synchronization variables and will then proceed with its function. When the lock is not yet available it will indicate that the atomicity of the operation has failed - the software implementation will proceed with next attempt of acquiring the lock. The spin-access to the local data cache and/or remote memory through the bus will be blocked and thus no power will be spent in such energy inefficient operations. Alternatively, the processor ISA is augmented to support dedicated *instructions for lock acquisition and release*. These instructions can be implemented to either block or non-block the pipeline execution. The blocking version would simply stall the pipeline until it is signalled by the DSC to continue execution when the lock has become available to the local processor. Clearly, while the pipeline is stalled the processor will be in a low-power mode as no execution activities be present. The non-blocking version of these instruction would be used by the operating system or thread library to implement higher-level power saving policies. These policies will be briefly described in Section5.

5. COMPILER AND OS SUPPORT

The role of the compiler/software developer is limitted to the instantiation of the locks/barriers and the allocation of unique addresses for each such synchronization variable. As explained in the previous section, this address will be used to form the unique *LockID* or *BarrierID*. In the case of dedicated instructions for the synchronization operation and system bus support for acquire/release commands, the role of the compiler is to generate the globally unique identifier for each lock and barrier. This can be easily achieved with an operating system support. As this is performed during program initialization, no performance overhead will be incurred in practice. The operating system (or thread library) then makes sure to allocate and load the state for each such lock/barrier in the local DSC. This is performed when the worker threads are loaded on the processor nodes. The DSC controllers on different processors can have different sets of synchronization variables allocated to them depending on what the working threads are operating on and the way synchronization is performed amongst them.

An efficient OS support can be implemented when non-blocking versions of the lock acquire and release instructions are supported, then the OS can have a full control in implementing the power saving policies. When the lock or barriers are released, the DSC will simply generate an interrupt and inform the local OS. There are various trade-offs that can be considered. For example, some locks are used in very short critical sections. In this case, the programmer can expect very short stall times and thus inform the OS to just stall the processor pipeline and accesses to the cache structures. Some locks, however, may be used for longer critical sections in unbalanced thread execution. In such cases a thread may stall for a significant amount of time waiting for its turn; consequently, more aggressive power down technique must be considered. In such power-down policies where the processor is brought to very low-power mode through voltage/frequency scaling, the latency of resuming the processor may be non-trivial. Since these parameters are specific to the processor microarchitecture and manufacture process, it may be best for the programmer and the OS to decide what power-saving technique to employ to achieve maximum benefits.

6. EXPERIMENTAL RESULTS

We have conducted a detailed experimental study on a set of multithreaded parallel applications. We have chosen the kernel programs from SPLASH-2 [9], and parallel MPEG encoder/decoder from ALPBench [10]. We have also constructed a set of benchmarks, which are configured as four parallel threads, each performing one stage of computation in a stream processing pipeline. The threads communicate through butterfly buffers and synchronize using standard locks and barriers. The individual tasks constitute of: *FFT*, *ADPCM*, *matrix multiplication*, *data encryption tasks*, *lzo-compression*, *g721*, image processing - the *blur* and the *edge-detection*, and *video processing*. The tasks cover benchmarks from the MediaBench [11] and MiBench [12] suits, as well as from other open-source image and video processing tools. These applications, however, do not exhibit heavy lock/barrier utilization, and will only demonstrate the power benefits of the proposed technique. To evaluate the impact on the performance and system bus traffic, we have constructed two benchmarks, P1 and P2, that stress the synchronization operations of acquiring and releasing locks. The first benchmark, P1, comprises of simple short critical sections executed iteratively, in which contention for locks is frequent; four threads iteratively enter a short critical section to update a shared variable. The second benchmark, P2, consists of interleaved long and short critical sections, where lock contention is less severe.

We have used the M5 [13] simulator to perform our experiments, extended with a library for thread synchronization primitives. The simulated hardware configuration is of four processors connected to a shared memory through a common bus. Each processor features a 32K, 4-way set associative cache, with snoop-based coherence support. The cache power expenditure of the four cache configurations have been obtained through Cacti v4.2 tool [14] for $0.18\mu m$ technology. The energy associated with the additional hardware structures for the proposed methodology are evaluated as follows: We have modeled the TOTAL and AHEAD registers as 2-bit up/down counters, since there are only four processors in the system. The baseline is based on load-linked/store-conditional

	Bus Transactions			Performance (cycles)		
	Baseline	DSC	Reduct.	baseline	DSC	Reduct.
P1	88,506	32,508	63.27%	157,436	133,434	15.25%
P2	3,904	1,833	53.05%	231,918	205,462	11.41%

Table 1: Performance and bus traffic characteristics

	paused cycles	total cycles	p/t
FFT	78,045	228,722,336	0.03%
LU_con	393,799,327	1,117,336,393	35.24%
LU_noncon	151,248,064	788,640,101	19.18%
RADIX	222,772	23,251,486	0.96%
CHOLESKY	78,175	924,734,914	0.01%
Mpegenc	62,020,947	192,784,501	32.17%
Mpegdec	1,312,658	15,388,463	8.53%
APP1	10,594,745	18,069,904	58.63%
APP2	72,411,385	112,554,584	64.33%
APP3	23,963,057	36,836,670	65.05%
APP4	5,899,673	12,160,014	48.52%

Table 2: Performance balance characteristics

paired instructions, which require polling of data cache when a thread is waiting for a lock or barrier.

As seen in Table 1, by applying our efficient synchronization architecture, the cache activities and bus contention due to synchronization are both greatly reduced. The application performance in terms of CPU cycles is also improved, because of reduced cache misses and bus traffic.

Table 2 reports the number of cycles the processors in the system wait for synchronization and the total number of cycles of execution. It is noteworthy that the results vary significantly across the different benchmarks. Some kernels, such as FFT, RADIX, CHOLESKY from splash2, are very well balanced in terms of parallel thread running times. For these benchmarks, stall times caused by blocking on barriers is negligible. In other applications, significant stalling times due to synchronization can be observed and the power reductions for these benchmarks will be higher.

Table 3 reports the energy reductions achieved by employing the proposed synchronization mechanisms. For this study we have modeled a DSC that disabled the accesses to the data cache, while the processor is spinning on the lock. Consequently, the achieved energy reductions are in correlation with the amount of time spent waiting on acquiring the lock or clearing the barrier. The average energy reductions among these benchmarks is 29%, which includes the overhead introduced by the synchronization controllers.

energy(mJ)	improved	baseline	reductions
FFT	28,043	28,057	0.05%
LU_con	139,786	214,661	34.88%
LU_noncon	132,593	161,350	17.82%
RADIX	4,411	4,454	0.95%
CHOLESKY	100,796	100,811	0.01%
Mpegenc	29,531	41,324	28.54%
Mpegdec	3,056	3,305	7.55%
APP1	1,406	3,420	58.89%
APP2	7,625	21,394	64.36%
APP3	2,458	7,014	64.96%
APP4	1,174	2,296	48.86%

Table 3: Achieved energy reductions

7. CONCLUSIONS

We have presented a novel synchronization implementation scheme that achieves good performance and power efficiency. The proposed method features dedicated controllers that are distributed across the system to handle all synchronization related operations. Operating system and compiler support are integrated to provide flexible and optimal control of the controllers. Experimental results show promising power and performance improvements for a broad range of applications.

8. REFERENCES

[1] B. Akgul and V. Mooney, "PARLAK: Parameterized Lock Cache Generator", in *Design Automation and Test in Europe (DATE)*, pp. 1138 – 1139, 2003.

[2] M. Monchiero, G. Palermo, C. Silvano and O. Villa, "Efficient Synchronization for Embedded On-Chip Multiprocessors", *IEEE Transactions on Very Large Scale Integration Systems*, vol. 14, n. 10, pp. 1049–1062, October 2006.

[3] A. Marongiu, L. Benini and M. Kandemir, "Lightweight barrier-based parallelization support for non-cache-coherent MPSoC platforms", in *CASES*, pp. 145–149, 2007.

[4] B. Akgul, J. Lee and V. Mooney, "A system-on-a-chip lock cache with task preemption support", in *Conference on Compilers, Architecture, and Aynthesis for Embedded Systems (CASES)*, pp. 149–157, 2001.

[5] C. Yang and A. Orailoglu, "Light-weight synchronization for inter-processor communication acceleration on embedded MPSoCs", in *CASES*, pp. 150–154, 2007.

[6] J. Li, J. Martinez and M. Huang, "The Thrifty Barrier: Energy-Aware Synchronization in Shared-Memory Multiprocessors", in *International Symposium on High Performance Computer Architecture (HPCA)*, 2004.

[7] O. Golubeva, M. Loghi and M. Poncino, "On the energy efficiency of synchronization primitives for shared-memory single-chip multiprocessors", in *Great Lakes Symposium on VLSI (GLSVLSI)*, pp. 489–492, 2007.

[8] B. Saglam and V. Mooney, "System-on-a-chip processor synchronization support in hardware", in *Design, Automation and Test in Europe (DATE)*, pp. 633–641, 2001.

[9] J. Singh, W-D. Weber and A. Gupta, "SPLASH: Stanford parallel applications for shared-memory", *SIGARCH Computer Architectures News*, vol. 20, n. 1, pp. 5–44, 1992.

[10] M-L. Li, R. Sasanka, S. Adve, Y-K. Chen and E. Debes, "The ALPBench benchmark suite for complex multimedia applications", in *International Symposium on Workload Characterization*, pp. 34–45, October 2005.

[11] C. Lee, M. Potkonjak and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", in *30th MICRO*, pp. 330–335, December 1997.

[12] M.R Guthaus, J. S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge and R.B. Brown, "MiBench: A free, commercially representative embedded benchmark suite", in *WWC*, pp. 3–14, Dec 2001.

[13] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi and S. Reinhardt, "The M5 Simulator: Modeling Networked Systems", *IEEE Micro*, vol. 26, n. 4, pp. 52–60, 2006.

[14] D. Tarjan, S. Thoziyoor and N. Jouppi, "CACTI 4.0: An Integrated Cache Timing, Power and Area Model", Technical report, HP Laboratories Palo Alto, June 2006.