Efficient Synchronization for Distributed Embedded Multiprocessors

Abstract:

In multiprocessor systems, low-latency synchronization is extremely important to effectively exploit fine-grain data parallelism and improve overall performance. This brief presents an efficient synchronization for embedded distributed multiprocessors. The proposed solution works in a completely decentralized request–response manner via explicit message exchange among the processing elements. Scalable lock and barrier synchronization algorithms, which are derived from the inherent distributed characteristics of the underlying architecture, are proposed to enable fair, orderly, and contention-free synchronization. We implement the proposed synchronization model in a distributed 32-core architecture with a commercial cycle-accurate SystemC simulation platform. Experimental results that show our proposed approach achieves ultralow synchronization latency and almost ideal scalability when the core count scales. The proposed architecture of this paper analysis the logic size, area and power consumption using Xilinx 14.2.

Enhancement of the project:

Existing System:

Conventional software approaches usually use the atomic instructions, such as test-and-set, to access shared variables in a busy-wait manner. However, such solution often suffers from a high synchronization overhead and poor scalability caused by remote polling. Therefore, most recent multiprocessors, e.g., Tilera, use cache-coherent system that allows polling to be performed at local cache. However, hardware cache coherency is not always applied in the embedded systems due to the stringent cost and power constraints. In addition, a few emerging multicore architectures, e.g., the IBM cell processor and the ClearSpeed CSX processor, employ explicitly programmable local memory for each processing element (PE) rather than a coherent cache. Therefore, the primary objective of this brief is to implement a synchronization mechanism...
under the assumption that no coherent cache exists, and let it support distributed many-core architectures. Hardware solutions have recently been proposed [5], where a centralized hardware engine attached to the memory controller snoops and manages all synchronization data globally. However, the centralized solution will incur heavy traffic contention when multiple nodes compete for a synchronization token.

**Disadvantages:**

- Latency is high

**Proposed System:**

This brief focuses on the two most common synchronization primitives: 1) lock and 2) barrier. A lock is used to enforce mutually exclusive access to a shared resource, whereas a barrier is used to force a group of processes to gather at a certain point of execution. In this section, we will discuss the detailed implementation of these two primitives.

**Distributed Lock Synchronization**

In the proposed synchronization model, each lock token is assigned a single unique base processor, where a controller is designed to keep track of its precise state. To acquire or release a specific lock, one processor sends a request to the lock’s base via message passing. Upon receiving the request, the base processor responds with an indication that the lock is either free or already owned. In order to achieve a queued contention-free lock, we propose to connect the processors pending on a certain lock as a singly linked list, which is shown in Fig. 1. Local to each processor node, a NEXT register is used to link the next processor waiting for the same lock.
As a detailed demonstration, Fig. 2(a) shows a scenario whereby three PEs acquire lock 1 sequentially. It is assumed that PE0 acquires first by sending a request to PE1, which is the base of lock 1 (Transition 1). Since the lock is initially free, PE1 grants the request (Transition 2) and updates its TAIL register to PE0’s vector bit 0b001. After PE0, PE2’s request (Transition 3) is rejected, because the lock is already held by PE0. In this case, the new tail becomes PE2, whose ID is delivered to PE0’s NEXT register (Transition 5). Finally, the base processor, PE1, acquires the lock locally (Transition 6). Since lock 1 is still unavailable, PE1 updates itself to the TAIL, links PE2’s NEXT (Transition 7), and then suspends.
Barrier Synchronization Protocol

In the proposed distributed synchronization model, barrier protocol is also well supported in a similar way, where each barrier is assigned a single unique base processor for keeping track of its precise state. Like lock protocol, the barrier algorithm also consists of two parts: 1) the requester end and 2) the base end.
Fig. 3. Barrier synchronization protocol.

Fig. 3 shows a barrier synchronization scenario using the proposed model. PE0 and PE2 reach the barrier successively and send their barrier requests to the base PE, respectively (Transitions 1–4). Since PE1 has not arrived yet, the base PE rejects these requests and sets the vector bits of the arrived PEs in the QUEUE register for recording. When PE1 finally arrives (Transition 5), all required PEs have reached the barrier, and thus all the pending processors, whose vector IDs are saved in QUEUE, are signaled to resume (Transition 6).

To implement the proposed synchronization, we use our in-house hybrid shared-memory/message-passing multiprocessor system-on-chip (MPSoC) platform [9], the block diagram of which is shown in Fig. 4.

Processing Element

The PE in this architecture is a 32-bit RISC processor with four-stage pipeline. We implement the PE using Synopsys LISA methodology.

Advantages:

- ultralow synchronization latency
The Master of IEEE Projects

Software implementation:

- Modelsim
- Xilinx ISE