Light-weight Synchronization for Inter-processor Communication Acceleration on Embedded MPSoCs*

Chengmo Yang and Alex Orailoglu
Computer Science and Engineering Department
University of California, San Diego
9500 Gilman Drive, La Jolla, CA 92093
{c5yang, alex}@cs.ucsd.edu

ABSTRACT

Advances in semiconductor technologies have placed MPSoCs center stage as a standard architecture for embedded applications of ever increasing complexity. Efficient utilization of the ample hardware resources requires applications to be decomposed into fine-grained threads, engendering in turn a large amount of interprocessor communications. While fine-grained on-chip interconnects can reduce the data transfer overhead, the traditional synchronization mechanisms, such as spin locks and barriers, still cause significant contention in polling shared variables. To overcome this issue, in this paper we propose a light-weight distributed synchronization mechanism which statically encodes the semantically correct order of accesses to each shared variable. A sharp reduction in the number of code bits is attained through a reference coloring algorithm, which furthermore enables an implementation within negligible hardware overhead. This light-weight synchronization mechanism allows dependent threads to frequently exchange data during execution, in turn enabling the exploration of fine-grained parallelism for applications with complex dependences.

Categories and Subject Descriptors: C.2.0 [Computer-communication Networks]: General –*Data communications*

General Terms: Performance

Keywords: Synchronization, interprocessor communication

1. INTRODUCTION

As the advances in VLSI fabrication technologies offer a tremendous amount of computational power, the Multiprocessor System-on-Chip (MPSoC) [1, 2] is quickly becoming a standard organization for high-end embedded systems. Unlike traditional single processor architectures, this decentralized execution model aims to exploit more coarse-grained parallelism, thus requiring applications to be explicitly parallelized into multiple threads either by the programmer or by the compiler. However, this parallelization of programs usually imposes interprocessor communication and synchronization, at a frequency and volume increasing super-linearly as the number of threads grows. Current MPSoCs usually employ an on-chip network or a global shared memory [3] to implement interprocessor communications. Unfortunately, both communication schemes typically impose a significant overhead of hundreds or even thousands of clock cycles [4, 5], thus constituting a significant bottleneck for this decentralized execution model.

Because of the significant communication overhead associated with current MPSoCs, the decomposition of an application has traditionally accorded a high priority to the reduction of communication frequency, resulting in the inability of coupled applications to fully utilize the scaling number of on-chip *Processing Elements* PEs available in future MPSoCs. This crucial limitation can be illustrated more clearly by considering a representative embedded application, MPEG-4 [6], in which a video stream is composed of a sequence of *video object planes* (VOPs) that display strong inter-VOP dependences. More specifically, VOPs can be classified into three types: *intraframe* (I-) VOPs, forward predicted (P-) VOPs, and bidirectional-predicted (B-) VOPs. A typical coding pattern is presented in Figure 1, with arrows representing the inter-VOP dependences. As can be seen, the strong dependences preclude the independent encoding/decoding of each VOP, thus significantly limiting the amount of parallelism that can be exploited by the traditional application parallelization strategy.

The acceleration of coupled application execution necessitates a cheap yet efficient communication scheme. Recent advances in network-on-chip (NoC) [7] have enabled sizable reductions of data transmission overhead. However, interprocessor communication still displays significant overhead as it employs traditional synchronization mechanisms, such as *spin locks* and *barriers*, to ensure mutual exclusion through continuous polling of a shared variable. These synchronization mechanisms not only impose large contention on the on-chip network, but also require memory accesses to be serialized, thus creating significant overhead in communication.

To overcome this issue, in this paper we propose a light-weight distributed synchronization mechanism for shared memory MPSoCs. Rather than explicitly inserting synchronization

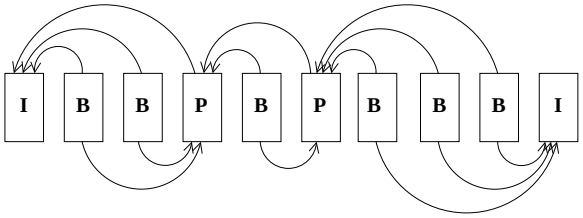
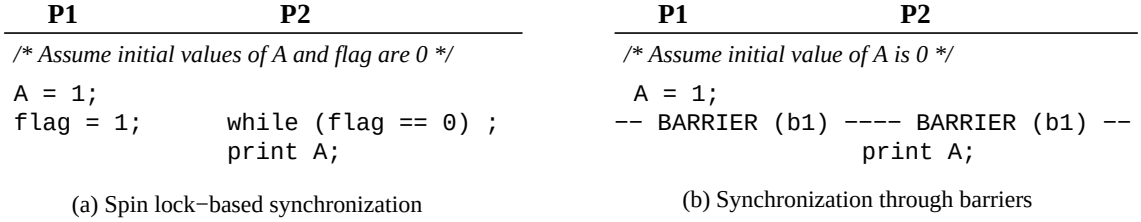


Figure 1: Limited application parallelism: inter-VOP dependences in MPEG-4

*This work is supported in part by a grant from Cal-(IT)^2.

CASES'07, September 30–October 3, 2007, Salzburg, Austria.



(a) Spin lock−based synchronization

(b) Synchronization through barriers

Figure 2: Explicit event synchronization to impose memory access order

variables to serialize the transmission of data through a shared memory location, we propose a mechanism to encode dependence information within each memory access, thus enabling synchronization to be combined together with data communication. Furthermore, by utilizing statically extracted application information, a sharp reduction in the number of code bits needed is attained through the proposed reference coloring algorithm, thus enabling an implementation within negligible hardware overhead.

The remainder of this paper is organized as follows. Section 2 discusses the limitations of current synchronization techniques. Section 3 presents in detail the conceptual idea and the implementation of the proposed light-weight synchronization mechanism. Section 4 provides an experimental evaluation of our work, while Section 5 summarizes this paper.

2. BACKGROUND AND MOTIVATION

In a decentralized architecture, the accesses to a shared variable emanating from different processors may be performed out-of-order, thus requiring explicit synchronization points to be inserted by the programmer into a parallel program to correctly serialize accesses to shared resources. Figure 2 presents two common synchronization primitives used by parallel applications, the *spin locks* which ensure atomic access to the data, and the *barriers* which ensure that a group of cooperating threads all have reached the barrier before any of them is able to advance.

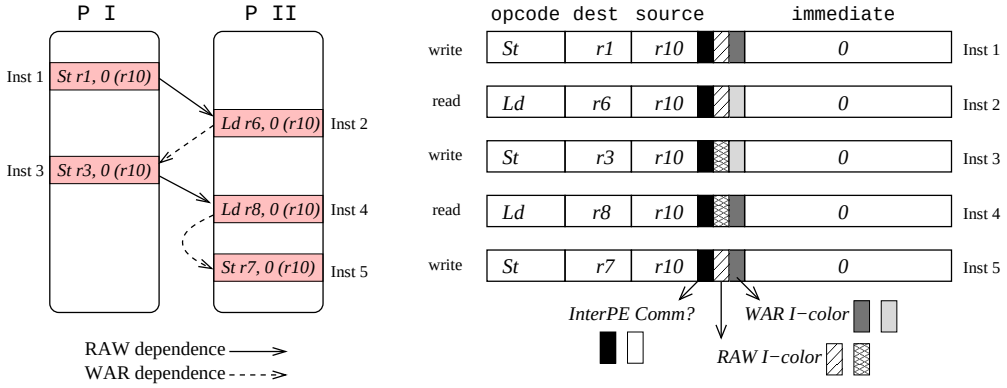
Given the serializing nature of synchronization operations, their performance often limits the achievable concurrency of parallel applications. Unfortunately, one crucial limitation of traditional synchronization mechanisms is that they ensure mutual exclusion through continuous polling of a shared variable, such as the reading of *flag* and *b1* from **P2** in Figure 2. This mechanism not only imposes large contention on the on-chip network, but also requires memory accesses to be serialized. To reduce this polling overhead, a number of optimization techniques have been proposed in the conventional multiprocessor arena. For instance, simple spin locks can be optimized by delaying each consecutive probe of the lock [8], or by buffering the threads waiting for the lock [9]. For barriers, efficient implementations for large scale distributed systems have been proposed in [10]. The idea consists of representing a single synchronization variable by means of a set of variables organized in a tree data structure, thus enabling a group of threads to be synchronized from the tree leaves in parallel.

While these optimization techniques can reduce network contention, the polling of shared variables cannot be completely eliminated because it directly derives from the fundamental nature of the traditional synchronization mechanisms, that is, the use of explicit synchronization variables to serialize the transmission of data in communication. On the other hand, a detailed examination shows that the serialization of data communication accesses does not necessarily require the use of explicit synchronization variables. More specifically, because each communication is composed of a write operation followed by a read operation to the same memory location, the dependence information between these memory accesses can be statically extracted and explicitly encoded. These code words can be written/read together with the data in transfer (variable A in Figure 2). Furthermore, a dynamic checking of the encoded dependence information enables the identification of the status of the data in communication, based on which the read operation can be suspended to achieve semantically correct communication. In this way, the accesses to explicit synchronization variables, such as *flag* and *b1* in Figure 2, can be completely eliminated.

Statically encoding data dependences of each communication to reduce synchronization overhead is eminently suitable for embedded systems, as the compiler can effectively exploit the limited set of well-defined applications. However, an effective encoding mechanism is still necessitated in order to capture the dependences within a highly constrained number of code bits, as otherwise the overhead of writing/reading the code words would be comparable to the overhead of writing/reading an explicit synchronization variable. Previous studies [11] have shown that most applications display highly similar and static communication patterns in that each thread regularly communicates with a small and fixed subset of the rest of the threads. More crucially, a large portion of the communication is performed via point-to-point communication, that is, consistent communication between the same two processors. This property of restrictive communication patterns enables the design of a highly effective encoding mechanism. In this paper we propose a novel ***reference coloring algorithm***, which can encode global dependence information in arbitrary access contexts within only a **2-bit** overhead for each memory access in a point-to-point communication. This negligible encoding overhead furthermore enables an easy incorporation of the proposed synchronization scheme into most embedded system architectures by a slight extension of the data transfer instructions.

3. ENCODING-BASED SYNCHRONIZATION

As PEs in shared memory MPSoCs communicate through accessing the same memory location, the data dependences between memory accesses can be classified into two types: *local dependences* that occur in instruction pairs lying within the same PE, and *global dependences* that exist between load/store instructions accessing the same memory location from different PEs. Due to unpredictable run-time events, such as cache misses in the data subsystem and branch execution in the control subsystem, two PEs may access the shared data out of order. Semantically correct communication necessitates obviously in-order execution of all globally dependent instructions emanating from various PEs. The challenge here is to



(a) Access sequence to CR1

(b) Encoding of RAW and WAR I−colors

Figure 3: Encoding of global data dependences

ensure in-order execution of global accesses while incurring neither significant amounts of hardware overhead nor performance degradation.

In general, a pair of global dependent memory access instructions can be composed of either a store and a load *(RAW dependence)*, or a load and a store *(WAR dependence)*, or a store and a store *(WAW dependence)*. Considered from the aspect of interprocessor communications, a *RAW dependence* ensures that the read operation in each communication obtains the correct data, while a *WAR dependence* ensures that the data of an incomplete communication will not be overwritten by a write operation in a subsequent communication. A *WAW dependence* between consecutive memory accesses, on the other hand, implies a redundant usage of the shared memory location, as the value stored by the first write operation is not consumed by any read operation.

The absence of redundant usage of globally shared memory locations is presumed in the proposed synchronization framework, a task easily achieved by standard compiler techniques. In the proposed synchronization framework, the redundant usage of global shared memory locations is presumed to be eliminated by the compiler. In general two types of access patterns can be classified as redundant usage of a global shared memory location:

- **Two consecutive *store* instructions**. As the value stored by the first *store* is not consumed by any *load* instruction, the first *store* is redundant.
- **Multiple *load* instructions from the same PE that depend on the same *store*.** As the first *load* instruction will load the data in communication into a local register of that PE, subsequent *load* instructions emanating from the same PE are redundant.

3.1 Encoding of global data dependence

Once the two redundant cases have been eliminated, dependent threads executed on different PEs will communicate by using the store/load instructions in an **alternating order** to access a shared memory location. This property can be observed clearly in Figure 3*a*, which presents an instruction sequence executed on PEs $P\ I$ and $P\ II$ to access the memory location $MEM[r10]$. More crucially, this highly regular access pattern enables a highly efficient encoding technique to preserve the data dependence information. Figure 3*b* presents the incorporation of the proposed static encoding technique into standard data transfer instructions.[1] As can be seen, our static encoding technique uses one bit to distinguish global load/store instructions, together with two additional bits to encode global *RAW* and *WAR* dependences.

To preserve semantic correctness of a *RAW dependence*, a read operation (e.g. *Inst 4* in Figure 3*a*) should be blocked if it attempts to access the data earlier than the corresponding producer (e.g. *Inst 3* in Figure 3*a*). This can be achieved through forcing each producer to write a distinct "signature" together with the data in communication, and forcing each read to verify the proper signature before it obtains the data. One straightforward solution would consist of the explicit specification of the address of the corresponding producer in each read. However, the encoding overhead of this solution is nontrivial, as the instruction address typically incurs at least a 32-bit overhead. Furthermore, writing/reading a 32-bit signature at run-time may impose an overhead comparable to the write/read of an explicit synchronization variable.

We propose instead a more efficient encoding solution by exploiting the regularity of access patterns for the shared data. More specifically, because only two PEs are involved in each point-to-point communication, as long as two adjacent write operations can be differentiated, RAW violations can be precluded. Accordingly, we propose a reference coloring algorithm which alternatingly uses two *RAW I-colors* during the static compilation process to make sure adjacent write operations have distinct *RAW I-colors*. This property can be observed be examining the behavior of *Inst 1*, *Inst 3* and *Inst 5* in Figure 3b. During execution, each write operation will write its *RAW I-color* together with the data in communication, enabling each read operation to check the RAW color to ensure the completion of the execution of its producer. Accordingly, each read operation is assigned the same *RAW I-color* as its producer, as can be observed from *Inst 2* and *Inst 4* in Figure 3b.

The preservation of *WAR dependences* encounters additional challenges, as traditionally each producer may have more than one consumer. More specifically, traditionally no write operation can be performed until all the read operations in a previous communication have been executed. However, for point-to-point communication each producer has a single corresponding consumer only, thus enabling a further reduction

[1]The format of the data transfer instructions shown in the figure is used by a wide range of embedded architectures [3], such as *ARM*, *Hitachi SuperH*, and *Mitsubishi M32R*.

in the number of code bits needed. More specifically, WAR violations can be prevented in the same way as RAW violations, through the usage of two *WAR* ***I-colors***. The encoding results can be observed in Figure 3b, wherein *Inst 2* and *Inst 3* share the same *WAR* ***I-color***, which differs from the *WAR* ***I-color*** shared by *Inst 4* and *Inst 5*.

A pseudocode for a slight extension to the compiler in order to incorporate the necessary updates for generating the suggested I-colors can be undertaken as described below.

1. **for** each memory location used in point-to-point communication {
2. Order all the accesses to that location.
3. **if** two consecutive accesses are read operations, write operations, or a write followed by a read emanating from the same PE, /* *they constitute redundant references that need to be pruned* */
4. **then** prune them.
5. **else**
6. Mark all the remaining accesses as "inter-PE comm"
7. Assign one RAW I-color and one WAR I-color to each write operation, with the property that two adjacent write operations have distinct RAW and WAR I-colors,
8. Color each read operation using the same **RAW** I-color as the write operation immediately *preceding* it, while using the same **WAR** I-color as the write operation just *following* it.
9. **end for**

It can be easily seen that for this algorithm no more than two RAW and two WAR I-colors are needed, implying that a total of **two** bits, one RAW and one WAR I-color bit, suffice to encode all the dependences. These two bits, together with the bit that is used to indicate whether a memory access is involved in inter-processor communication, constitute the only static encoding overhead of the proposed synchronization mechanism.

3.2 Dynamic checking and blocking

The reference coloring scheme discussed in the last section explicitly encodes the dependence information between memory accesses involved in interprocessor communication. It should be noted that the applicability of the proposed encoding-based synchronization mechanism is quite flexible and broad. The proposed synchronization mechanism can be incorporated into any type of communication implemented through accessing a shared hardware resource, for example, a shared centralized memory, a shared cache, or even registers shared between PEs, such as the registers used in the *Multiscalar* architecture [12], the SKY architecture [13], and the RAW architecture [14].

Dynamically when a memory access instruction is executed, if the static encoded "*inter-PE comm*" bit indicates that the specific instruction is involved in an interprocessor communication, the status of the data in communication will be checked and updated based on the statically encoded *I-colors*. More specifically, two extra bits, denoted as the *RAW* ***R-color*** and the *WAR* ***R-color*** bits, are added to record the status of the data in communication. This can be clearly seen in Figure 4. Moreover, in order to eliminate a continuous polling of the **R-color** bits, a *PEID* field is also added to record whether a PE is waiting to access the data in communication, thus enabling a light-weight mechanism to await a blocking PE. As

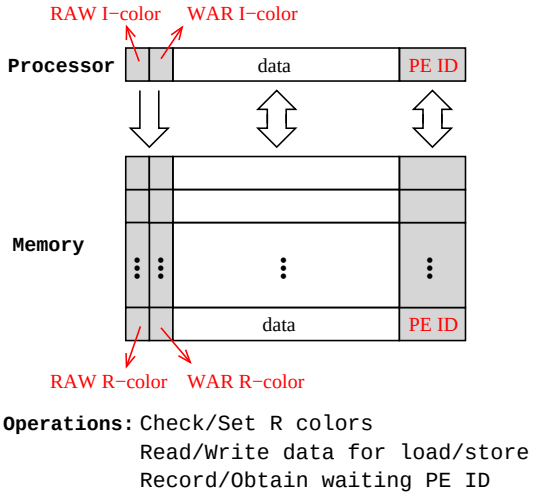


Figure 4: Hardware extension and operations

only two PEs are involved in each point-to-point communication, at most one PE needs to be blocked, implying that one *PEID* field suffices.

In the process of executing a global load/store instruction, two synchronization functions need to be performed in order to record the status of the data in communication: the **checking** and the **setting** of the *R-colors*. Furthermore, if an instruction attempts to access the data in a semantically incorrect order, two extra synchronization functions need to be performed: the **blocking** and the **unblocking** of the specific instruction. The following two cases delineate the detailed functions performed when executing load and store instructions, respectively.

Load instruction: Before reading the data, the PE checks if the *RAW* ***R-color*** bit has the same color as the *RAW* ***I-color*** statically encoded in the load instruction. If so, the instruction can proceed to execution. Otherwise the instruction needs to be stalled, and the PE's ID will be recorded in the *PEID* field. The blocking of the load continues until a subsequent store instruction has updated the *RAW* ***R-color***.

Once the execution of the load instruction has been completed, the PE sets the *WAR* ***R-color*** bit of the memory location to the *WAR* ***I-color*** encoded in the load instruction. Furthermore, if the *PEID* field shows that a store instruction emanating from the alternative PE is waiting to update the data, the store will be unblocked.

Store instruction: Before reading the data, the PE checks if the *WAR* ***R-color*** bit has the same color as the *WAR* ***I-color*** statically encoded in the store instruction. If so, the instruction can proceed to execution. Otherwise the instruction needs to be stalled, and the PE's ID will be recorded in the *PEID* field. The blocking of the store continues until a subsequent load instruction has updated the *WAR* ***R-color***.

Once the execution of the store instruction has been completed, the PE sets the *RAW* ***R-color*** bit of the memory location to the *RAW* ***I-color*** encoded in the store instruction. Furthermore, if the *PEID* field shows that a load instruction emanating from the alternative PE is waiting to obtain the data, the load will be unblocked.

4. EVALUATION

Since the commercial absence of the associated compiler technology precludes a precise analysis of the ensuing ben-

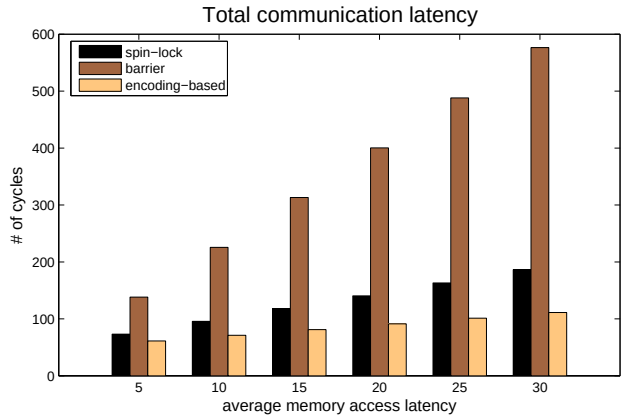


Figure 5: Total communication latency, assuming the average number of extra cycles spent in waiting for the consumer thread of 50

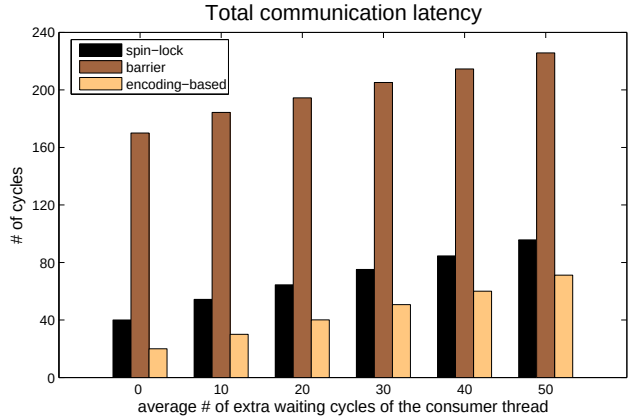


Figure 6: Total communication latency, assuming an average memory access latency of 10 cycles

efits, we evaluate the proposed light-weight synchronization mechanism by theoretically comparing the number of memory accesses necessitated in the proposed synchronization to the number of memory accesses necessitated in conventional spin-lock and barrier synchronization presented in Figure 2. We randomly generate a sequence of 1000 point-to-point communications, for which the average communication latency is computed for each synchronization scheme.

In general, the communication latency is a function of the memory access latency, the total number of memory accesses involved in communication, as well as the number of extra cycles spent in waiting for the consumer thread. In our experimental framework, the memory access latency is varied from 5 to 30 cycles, while the average number of extra cycles spent in waiting for the consumer thread is varied from 0 to 50. The results are plotted in Figures 5 and 6. As can be seen, the proposed synchronization scheme outperforms both the spin-lock and the barrier synchronization schemes in reducing communication overhead. This is because the proposed encoding-based synchronization scheme significantly reduces the number of memory accesses needed in point-to-point communications. As all the memory accesses involved in synchronization and communication need to be serialized through sequential bus transactions which require tens of cycles, the significant reduction in the number of memory accesses directly implies a significant performance improvement enabled by the proposed encoding-based synchronization scheme.

5. CONCLUSIONS

We have described a light-weight distributed synchronization mechanism to accelerate the inter-processor communications for shared memory MPSoCs. The synergistic collaboration between the compiler, responsible for statically identifying and encoding global data dependences between memory accesses involved in inter-processor communication, and the hardware extension of the conventional storage organization provide a novel synchronization framework for future MPSoCs. The proposed light-weight synchronization mechanism approach not only reduces the control and communication overhead associated with global storage architectures, but also allows dependent threads to frequently exchange data during execution, in turn enabling the exploration of fine-grained parallelism for applications with strong dependences.

6. REFERENCES

[1] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," *Proc. 7th ASPLOS*, pp. 2–11, Sept. 1996.

[2] W. Wolf, "The future of multiprocessor systems-on-chips," In *Proc. 41st DAC*, pp. 681–685, June 2004.

[3] J. L. Hennessy and D. A. Patterson, "Computer architecture: A quantitative approach (4th edition)," *Morgan Kaufmann Publishing Co.*, Jan. 2007.

[4] W. Wulf and S. McKee, "Hitting the memory wall: Implications of the obvious," *ACM Computer Architecture News*, 23(1):20–24, 1995.

[5] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Instroduction to the Cell multiprocessor," *IBM J. Research & Development*, 49(4/5):589–604, 2005.

[6] K.-S. Shin, W.-O. Yoon, J.-H. Jung, and S.-B. Choi, "MPEG-4 stream transmission and synchronization for parallel servers," *IEEE Multimedia*, 13(1):24–36, Jan. 2006.

[7] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli, "NoC synthesis flow for customized domain specific multiprocessor systems-on-chip," *IEEE Trans. on Parallel and Distributed Systems*, 16(2):113–129, Feb. 2005.

[8] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, 1(1):6–16, Jan. 1990.

[9] A. Kagi, D. Burger, and J. R. Goodman, "Efficient synchronization: Let them eat QOLB," In *Proc. 24th ISCA*, pp. 170–180, 1997.

[10] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. on Computer Systems*, 9(1):21–65, Jan. 1991.

[11] J. S. Vetter and F. Mueller, "Communication characteristics of large-scale scientific applications for contemporary cluster architecture," In *Proc. 16th IPDPS*, pp. 27–36, April 2002.

[12] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," *Proc. 22nd ISCA*, pp. 414–425, June 1995.

[13] R. Kobayashi, M. Iwata, Y. Ogawa, H. Ando, and T. Shimada, "An on-chip multiprocessor architecture with a non-blocking synchronization mechanism," *Proc. 25th EUROMICRO Conference*, vol. 1, pp. 432–440, Sept. 1999.

[14] M. B. Taylor, J. Kim, *et.al.*, "The RAW microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, 22(2):25–35, 2002.