University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

The Elephant and the Mouse: Non-Strict Fine-Grain Synchronization for Many-Core Architectures

Juergen Ributzka, Yuhei Hayashi and Guang R. Gao

CAPSL Technical Memo 099
June 11th, 2010

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

A new synchronization mechanism created under the dataflow model of computation was introduced during the late 1970s and called I-Structure. I-Structure exhibited the following important features: (1) it is a dataflow style synchronization, i.e., synchronization only occurs between an I-Structure producer and consumer operations that are accessing the same memory location; (2) it is fine-grain i.e., it synchronizes at a finer memory granularity than only at the whole data structure level (for instance, it would synchronize at each individual array element instead of barrier synchronization which synchronizes at the data structure level.); (3) it is a lenient (non-strict) synchronization - i.e., an I-Structure load can be issued (non-blocking) even before the corresponding I-Structure store is issued/completed.

This paper reports a study of I-Structures in the context of modern many-core chip architectures. The major points examined include:

- The creation of an I-Structure style design that exploits a lenient synchronization model using a modern many-core architecture - the IBM Cyclops-64 architecture.
- The implementation and integration of our design in the DEEP emulation system that can simulate the entire Cyclops-64 chip at gate level. This allows us to assess the feasibility of its hardware design and implementation.
- The demonstration of the advantages of I-Structure style synchronization - especially its lenient synchronization feature on the Cyclops-64 architecture through an experimental case study using wavefront computation. A quantitative comparison to traditional control-flow based synchronization, such as signal-wait, is reported.

1 Introduction

During the 1970s and 1980s, a novel computational model was introduced by Dennis [1] named Dataflow. Under this model, computation flows according to the availability of data, which means that several operations can be run in parallel if the dependent data is available to them (and there are free resources on which to run them). Under the umbrella of Dataflow, several interesting structures and methods were proposed, like the actors activity template structure for the Moonson Machine [2], static dataflow schemas [3] and the MIT tagged dataflow model [4]. Among these proposed methods, the I-Structure is a very interesting addition. The I-Structure was designed as a lenient fine-grain memory centric (dataflow style) synchronization method in which the requesting operations will wait on the memory construct to be initialized. This behavior allows consumer operations (i.e. loads) to be issued before a producer operation (i.e. store) is issued or completed. The consumer operations will have to wait until the producer operation completes. However, the waiting happens on the I-Structure construct and frees the issuer (i.e. the processor) to do other useful work. This non-blocking issuing behavior is what we call the leniency property of I-structures. Another of the I-structure's properties is that it allows a true data centric style synchronization since it permits the synchronization on an element level (i.e. the I-Structure) instead of depending on certain control flow constructs such as barriers and signal-wait. Finally, it allows the synchronization to occur on finer granularity levels than its control flow based counterparts. Nevertheless, it puts the restriction of

"single assignment on any given location. Due to the overwhelming trend of frequency scaling and uni-processor performance during the 1990s, Dataflow research was gently nudged out of mainstream computing. Due to the emergence of multi and many core designs that have permeated the computer market in the last decade, research on Dataflow models and Dataflow style synchronization have seen a renascence.

Although many synchronization methods exist today, most of them are defined under the control flow style of computation (i.e. they are processor centric). Most of these methods are called coarse-grain since they allow synchronization of structures at a very high level. This incurs high overhead which can be manageable on a small number of cores, but quickly becomes a critical performance killer on a large number of cores. All these synchronization constructs are critical for applications which exhibit data races, i.e., a condition that occurs when two or more memory operations concurrently try to access a single memory element and at least one of them is a write. Data races, if not taken care of, can produce erroneous or unexpected results in a given application. Unfortunately, many of the real applications on High Performance Computing (HPC) exhibit this phenomenon due to the need to use previous computed values on its data space. Some of the most famous applications are stencil like calculations such as the Finite Difference Time Domain (FDTD) and wavefront communication type algorithms like Sweep3D. Some of these problems can be parallelized by program re-structuring or the insertion of coarse-grain synchronization.

One well known synchronization construct is Signal-Wait. Under this model, the producer sends a signal to the consumer after its write has been completed. Such behavior guarantees the producer operation to be completed before the consumer read arrives. However, this also implies that the consumer will have to block and wait for the signal to arrive. Although the way that the wait is implemented (busy-wait versus sleep-and-wakeup approaches) will have a huge impact on its performance, it still incurs an unnecessary substantial overhead for the consumer. Furthermore, this has a negative effect on the processor and tool-chain ability to schedule and reorder instructions. Signal-Wait methods can be implemented in several ways and may need hardware support depending on the architecture. For example, architectures which use Out-of-Order engines will require a memory *fence* instruction so that memory operations will not be incorrectly reordered across the wait and force the results of any memory operations to be visible to the whole system. These strict conditions apply to every memory operation in the processor, even the ones that are not related or do not need synchronization. Such overhead can be reduced by certain program transformations, such as loop unrolling, which allows having a synchronization operation every nth iterations if unrolled n times. Although this increases performance, it also increases the time delay until the next processor can continue program execution. Due to this behavior, it becomes difficult to scale, especially for small problem sizes.

Coarse-grain synchronization constructs, like Signal-Wait, cannot be used to take full advantage of parallelism due to its strict behavior, overhead, the scheduling penalty, and the control-flow centric approach. Thus, many architectures have implemented fine-grain synchronization constructs in hardware. Some examples include the Denalcor HEP [5], Monsoon [2], the Tera/Cray MTA family of processors [6], MDP [7], Cedar, Multicube, KSR1, Alewife/Spar-

cle [8], the M-Machine, the J-Machine, ElDorado (aka Cray XMT) [9] and others. One popular way to implement the fine-grain constructs is to add an extra bit, called the Full/Empty bit, to each word in the memory. This enhancement plus the addition of several extensions to the Instruction Set Architecture to handle the Full/Empty bit allows fast and efficient fine-grain dataflow like synchronization. Since these bits are in each memory word, a synchronized operation will only complete if the memory word is in a pre-determined state (for loads the Full/Empty bit must be "Full and for stores the Full/Empty bit must be Empty). Upon completion of the operation, the state of the memory cell switches to the contrary state (i.e. a load will set the locations bit to Empty and store will set it to Full). These types of operations and the Full/Empty bit mirror the famous M-Structure construct proposed in [10]. The usage of fine-grain synchronization helps to achieve good performance and scalability as we will show in this paper.

Another factor that influences synchronization performance is the strictness of the operation. In general, strictness refers to when a value is evaluated. If the value is evaluated when it is requested, it is called strict. If the value is evaluated when it is needed then it is called non-strict. In particular, strict operations stall or block execution until the operation is completed. Non-strict operations work in an asynchronous fashion and allow execution to continue even though the operation has not yet been completed. The following two examples clarify the differences between strict versus non-strict operations. The first example is a strict version of a synchronizing load (see Figure 1). The load (S1) is issued and its return value is checked to verify if the load was successful or not. This step is repeated until the load is successful. Successful in this case means that the synchronizing store has written the value to memory and set the full bit. The computation and loads (S2, S3, and S4) that follow the synchronizing load (S1) are issued after the synchronizing load has completed successfully. The second example shows a non-strict version (see Figure 2). The synchronizing load (S1) is issued and the execution of independent instructions (S2, S3, and S4) continues until the return value of the synchronizing load is actually required for the computation at statement S5. The second approach can achieve better performance since it allows the overlap of execution of independent instructions (S2, S3, and S4) with the synchronizing load (S1).

```
S1: while(sync_load_strict(&val, &(a[i])) != SUCCESSFUL);
S2: tmp1 = a[i+1];
S3: tmp2 = a[i+2];
S4: tmp1 = tmp1 + tmp2;
S5: result = tmp1 + val;
```

Figure 1: Example of strict synchronization

Even though the addition of the extra bit to each memory word allows the implementation of fine-grain synchronization constructs, its cost might be very high. The Synchronization State Buffer from Zhu et al. [11] mitigates this problem with a trade-off. This trade off is based on the observation that the number of synchronizations at any given time is much smaller than

```
S1: sync_load_non_strict(&val, &(a[i]));
S2: tmp1 = a[i+1];
S3: tmp2 = a[i+2];
S4: tmp1 = tmp1 + tmp2;
S5: result = tmp1 + val;
```

Figure 2: Example of non-strict synchronization

the number of memory locations in the system. Therefore, the use of a small buffer to keep track of the full/empty bits was proposed. However, this approach lacked the non-strictness of the I-Structures and other dataflow-type synchronization constructs.

In this paper, we propose an Extended Synchronization State Buffer (E-SSB) that combines the advantages of a small synchronization buffer with the advantages of non-strict synchronization in a many-core architecture. By adding the non-strictness, this structure behaves more like an I-Structure and it can reap all the benefits of dataflow like synchronization.

For this paper, we use a "real many-core architecture with 160 independent cores to show the feasibility and effectiveness of non-strict fine-grain synchronization. A more detailed description of this many-core architecture, named Cyclops-64, is given in Section 3.1.

Since our fine-grain synchronization extension is not available in the real hardware, we used the hardware description language (HDL) code of the Cyclops-64 architecture and extended it with E-SSB. This enhanced architecture was then emulated on a gate-level accurate emulation platform, which was also used during the original chip verification. A more detailed description of the emulation platform is given in Section 4.1.

Problem Formulation

The problem of the efficient synchronization constructs poses several questions:

- How difficult is it to implement and support non-strict fine-grain synchronization?
- What are the implications on used chip estate?
- What are the performance gains of non-strict fine-grain synchronization?
- How to ensure the correctness of our implementation and the given performance predicting with a very high degree of confidence?

The remainder of the paper is structured as follows: Section 2 presents a case study using wavefront computation. Section 3 describes the design and implementation of non-strict fine-grain synchronization. Section 4 introduces the experimental testbed and shows our results. Section 5 gives a recap of the related work. Section 6 concludes the paper.

2 Case Study: Wavefront

In this chapter we take a closer look at a wavefront computation-style program, which is our motivation for this paper. The C-code of the kernel is shown in Figure 3. First the algorithm initializes the top row and the first column of a 2D array. Next, the remaining elements of the 2D array are calculated based on the previously determined values from the left, top-left and top element. This forms a wavefront computation from the top-left corner to the bottom-right corner as shown in Figure 4.

Due to the dependence of an element on its previously computed neighbors, parallel versions of the wavefront kernel require synchronization constructs to ensure correctness. However, this kernel still exhibits enough parallelism to be efficiently executed on a many-core architecture. A naive approach would be to distribute the rows across the available processors on the chip in a round-robin fashion and enforce data dependencies via synchronization constructs.

```
for(i=0; i<N; ++i) {
  for(j=0; j<N; ++j) {
    a[i][j] = ( a[i-1][j-1] +
                a[i-1][j  ] +
                a[i  ][j-1]
              ) / 3;
  }
}
```

Figure 3: C code of the wavefront kernel

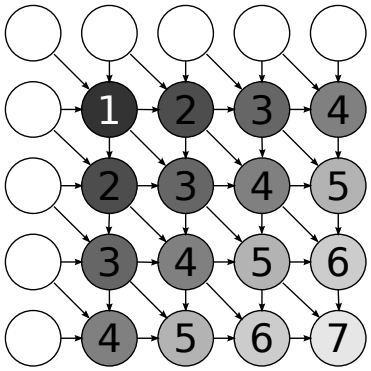


Figure 4: Wavefront Dependencies

We have implemented this case study benchmark for the Cyclops-64 many-core architecture with the following synchronization constructs: Barrier, Signal-Wait, and I-structure style fine-grain synchronization. The different implementations are described in the following sections.

2.1 Wavefront with Barriers

A well-known coarse-grain synchronization construct is the barrier. A barrier enforces order on memory operations and thread execution. Barriers, even if implemented or partially supported in hardware, can incur substantial overhead, which needs to be considered when parallelizing an application. One way to reduce the synchronization overhead is to use a blocking approach. The 2D array is divided into blocks and each row of blocks is processed by one thread. Threads are statically assigned in a round-robin fashion to rows. The spatial distribution of blocks is shown in Figure 5 and its schedule with barriers in Figure 6.

By increasing the block size, the overhead of the barrier can be mitigated, but it also reduces parallelism. The barrier synchronization is a very coarse-grain synchronization construct because it synchronizes all threads. The work allocated to each thread is equal (except for

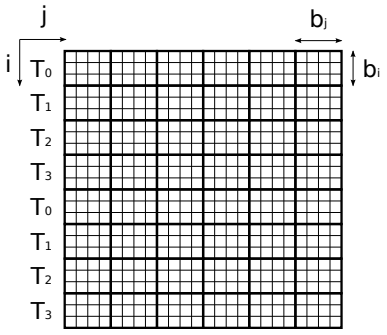


Figure 5: Partition

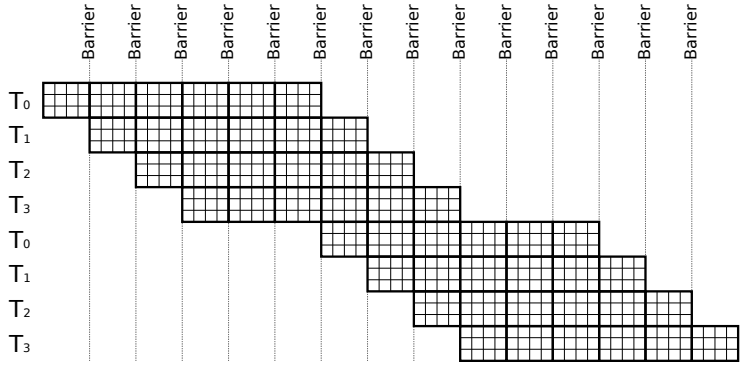


Figure 6: Schedule

the corner cases), but other unpredictable side-effects, like crossbar congestion, will produce a variation in execution time for each thread. This means that if any thread falls behind, all threads must wait for this one thread to complete, even though the wait for certain threads may be wasteful (i.e. no dependencies with the slowest thread). This unneccessary *strict* synchronization takes its toll and the problem further increases with the number of threads.

Even on the Cyclops-64 architecture, which has fast hardware support for barriers, this parallelization strategy did not scale very well with the number of cores. Furthermore, an increase in the problem size, which helps to mitigate the overhead of ramping up and down the wavefront, did not provide significant performance gains. More details can be found in Section 4.4.

2.2 Wavefront with Signal-Wait

Another well known synchronization construct is Signal-Wait. Signal-Wait can be seen as a fine-grain synchronization construct when compared to barriers. Instead of synchronizing a set of threads, it allows a finer control akin to point-to-point synchronization methods. In this parallelization strategy, the producer can signal the consumer when it has finished the write. The consumer will wait until the signal arrives and then read the data it was waiting for. Depending on the architecture the *Signal* and *Wait* functions have different implementations and special hardware support might be required. On an out-of-order architecture, a special operation called a *fence* instruction is required to make sure that the signal from the producer is not sent before the write and the read from the consumer is not issued before the wait. The overhead of Signal-Wait can be reduced by unrolling the loop N times and synchronizing every N elements, much like a smaller scale of the blocked barrier approach.

Experiments on the Cyclops-64 architecture have shown that this fine-grain parallelization strategy is more successful than the barrier approach. More details can be found in Section 4.4.

2.3 Wavefront with I-Structure like Fine-Grain Synchronization

The last synchronization construct we used in this experimental study is an I-structure/dataflow-like fine-grain synchronization method. In fact, we used three different versions of the fine-grain synchronization construct, which are described in detail in Section 3. The important difference between the three versions is that the first two versions are strict/blocking, while the last one is non-strict/asynchronos. We used the same approach as Signal-Wait, but we replaced the *Signal* and *Wait* functions with I-stucture like synchronizing load and store instructions, which are supported in hardware. We expected much better results from these synchronization constructs, because it only synchronized the load and store and not any unrelated memory operations. The first two synchronization constructs proved to be faster than the barrier approach, but were unfortunatelly still slower than the Signal-Wait implementation. This was due to the blocking behaviour of the first two synchronization constructs, which is fatal for in-order-issue processors. Signal-Wait is blocking on the receiver side, but not on the sender side. The first two fine-grain synchronization constructs are blocking on both the sender and receiver sides. To solve this dilemma, we created the third fine-grain synchronization construct, which is non-blocking on both sides - sender and receiver. With this small change, we achieved astonishing results. The third implementation beats all other implementations in every case. We received maximum speedup for any problem size and scaled much better with the number of threads. Even small problem sizes achieved better speedup with this implementation than with any of the previous synchronization constructs. More details of the results are presented in Section 4.4.

Problem Formulation

Our case study has shown that non-strict fine-grain synchronization is beneficial for implementing parallel programs on many-core architectures. In the following sections of the paper we will answer the following questions:

- How difficult is it to implement and support non-strict fine-grain synchronization?

  New features in chips can be simulated and tested in a fast and reliable fashion using functional-accurate simulators, but the real complexity is often misunderstood or just not implementable. To determine the complexity of fine-grain synchronization, we performed an implementation at the hardware description level (HDL) of a real many-core architecture. Section 3.3 gives a more detailed description of the changes that were neccessary to support fine-grain synchronization in the Cyclops-64 many-core architecture.

- What are the implications on used chip estate?

  The real hardware cost of a new architectural feature can, to a certain extent, be estimated by chip architects, but its final resource usage is unkown until an actual implementation has been performed. In Section 3.4 we discuss and describe both the additional hardware resources which are required to support fine-grain synchronization, and how we obtained these results.

- What are the performance gains of non-strict fine-grain synchronization?

  The effort and cost of adding a new architectural feature has to be validated. In the case of our non-strict fine-grain synchronization construct, we expect a substantial performance increase. Otherwise, it may be more useful to use chip real estate for other features or even more cores. In Section 4, we compare and contrast fine-grain synchronization with other already existing synchronization constructs of the Cyclops-64 many-core architecture.

- How to ensure the correctness of our implementation and the given performance prediction with a very high degree of confidence?

  The validation of new features and their true performance is difficult to measure with software simulators only. Software simulators may be cycle accurate, but they are incredibly slow and not useful to validate a full chip or even run a benchmark. Others might be fast, but sacrifice accuracy. In Section 4.1, we describe our emulation system and how we used it to obtain cycle accurate performance results of the whole chip with a very high degree of confidence and its usefulness for whole chip and system software validation.

3 Design and Implementation of Non-Strict Fine-Grain Synchronization

Before we go into the details of the design and implementation of non-strict fine-grain synchronization, we will first introduce the Cyclops-64 many-core architecture. Then we will show our proposed design and its actual implementation for the given many-core architecture.

3.1 The IBM Cyclops-64 Architecture

The IBM Cyclops-64 (C64) architecture is logically partitioned into 80 homogeneous processors which are connected to a 96-port crossbar. A processor contains two Thread Units (TUs), which share one Floating-Point Unit (FPU). Therefore, it is possible to have 160 independent and concurrent threads running at the same time. Every TU is attached to one SRAM bank. Each TU can access all SRAM banks via the crossbar. The SRAM banks can be configured during chip boot-up into two distinct sections. One section of the SRAM bank contributes to the Global Interleaved Shared Memory; the other section can be used as Scratch Pad memory. A TU has a direct, low-latency access to its own Scratch Pad. The Scratch Pad of other TUs can still be accessed through the crossbar. Sequential Consistency is guaranteed for the Global Interleaved Shared Memory, but not for the Scratch Pad. TUs are in-order single-issue and out-of-order completion cores and have a quad-ported register file (two read and two write ports) with 64 $\times$ 64bit General Purpose Registers (GPRs). All TUs share a common signal bus, which provides fast barrier support in hardware. Ten TUs (five processors) share one Instruction-Cache (IC) and four ICs share one crossbar port. There is no Data Cache (DC). Off-chip DDR2 memory is connected through four on-chip DDR2 memory controllers. Each memory controller is connected to its own crossbar port. Each chip can be connected to six

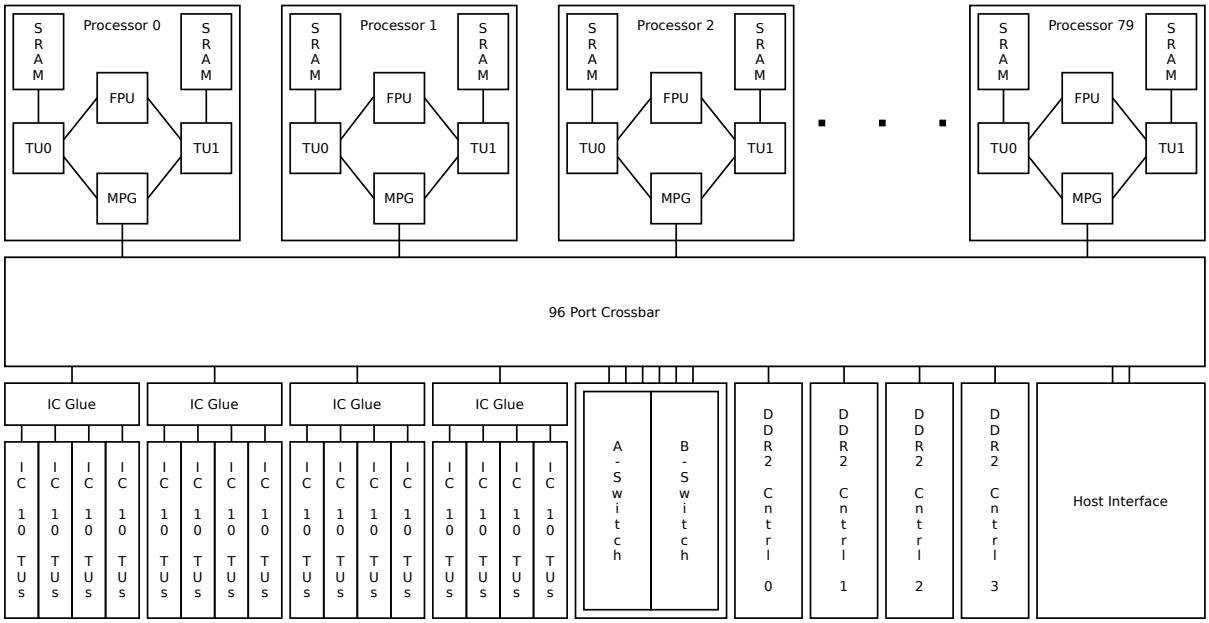


Figure 7: IBM Cyclops-64 (C64) Many-Core Architecture: The architecture consists of 80 processors (Processor 0 -79). Each processor has two Thread Units (TUs) called TU0 and TU1. Both share one Floating-Point Unit (FPU) and one crossbar port (MPG). Each TU is connected to a SRAM bank, which can be accessed by all other TUs via the crossbar. Ten TUs share one Instruction-Cache (IC). The system has four on-chip DDR2 memory controllers to access off-chip memory. The A-Switch is used to connect to the six surrounding neighbors in a 3D-mesh network.

neighboring chips in a 3-D mesh network. The network switch is also integrated into the chip and has six connections to the crossbar. The host interface is connected to two crossbar ports. In summary, the chip's crossbar interconnect possesses a total of 96 ports: eighty for the processors, four ports for the I-cache, four ports for on-chip DDR2 memory controllers, six ports for inter-chip communication, and two ports for the host interface. A logical overview of the chip is shown in Figure 7.

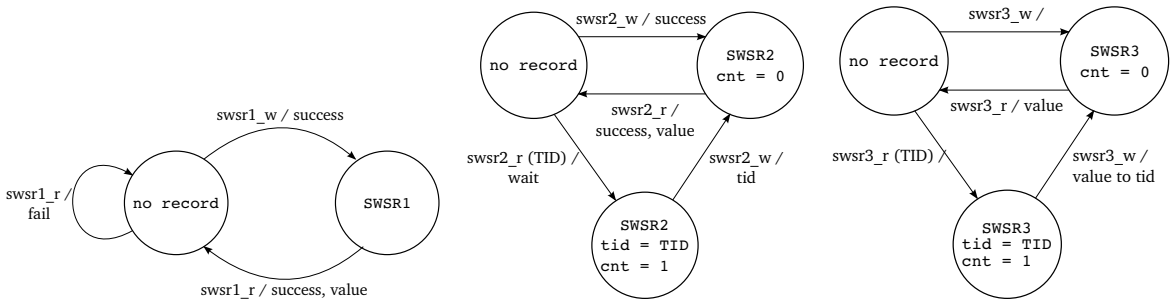
The architecture uses a explicit memory hierarchy similar to the one found in the NVIDIA CUDA or the Cell/B.E. architecture. Moreover, there is no paging or virtual memory support between all the memory hierarchy segments. More information about the C64 architecture and its system software can be found here [12–14].

3.2 Design of Fine-Grain Synchronization Constructs

In this section we will explain the design principles for non-strict fine-grain synchronization and its operational semantics. We used the Synchronization State Buffer (SSB) proposed by Zhu et al. [11] and extended it with non-strict fine-grain synchronization. The major goals in designing our Extended Synchronization State Buffer (E-SSB) was to improve programability and ease

of scheduling for the compiler. Our major interest were the single-writer-single-reader (SWSR) synchronization operations. The original SSB design had two different SWSR modes. Mode 1 employed a busy-wait approach for the reader until the data is ready. The second mode utilized the sleep-wakeup features of the architecture to reduce crossbar traffic and energy consumption. We added a third mode which eliminates the overhead of the synchronization operation with minimal additional hardware cost and added non-strict behaviour. Futhermore we extended all modes to suport any data size (byte, half word, word and double word) and signedness (signed and unsigned) of memory operations.

Figure 8 shows the state diagram of the different SWSR modes.



(a) Mode 1: busy-wait (b) Mode 2: sleep-wakeup (c) Mode 3: non-strict

Figure 8: Single-Writer-Single-Reader (SWSR) State Diagrams

The operational semantics for our non-strict fine-grain synchronization can be found below. For the the first two modes please refer to the original paper [11].

Mode 3 There are two possible scenarios - the write operation arrives first at the memory controller and the read operation second, or vice versa. In the first case, when the writer arrives first, an entry is created in the E-SSB indicating this scenario. No status code is returned to the writer, as compared to Mode 1 and 2. When the load operation arrives its corresponding entry in the E-SSB is obtained (if available) and it is allowed to proceed. Finally, the entry is removed from the E-SSB and the value of the requested memory location is returned to the reader. In the second case, when the reader arrives first, an entry is created in the E-SSB and no data is returned. The reader can continue issuing other instructions, which do not depend on the return value, while waiting for the data to return. When the writer arrives second, the value is stored in memory and also returned to the reader at the same time. Finally, the entry is removed from the E-SSB. Under this mode, the synchronization memory operations appear as normal load and store operations to the processor. The processor only stalls when a dependency is found between the synchronized operation and another operation. If two synchronized reads or writes happen to the same memory location, an interrupt is raised in the thread unit, whcih is responsible for the wrong use of the synchronization construct. Interrupts are also raised for not matching data sizes.

Using these operational semantics, we implemented the E-SSB in the Cyclops-64 architec-

ture at the HDL level. The following section gives an overview of the required changes and implementation decisions.

3.3 Implementation of the Extended Synchronization State Buffer (E-SSB)

In this section we will describe the architectural changes we performed to implement fine-grain synchronization in the Cyclops-64 many-core architecture. The Extended Synchronization State Buffer (E-SSB) required changes mostly in the Thread Unit (TU), because all the required logic related to the on-chip memory interface is located in there. In paticular, changes were requires on the instruction decoder to support the new E-SSB instructions and the storage interface, which is responsible for routing memory request from the netwok and the thread unit. Another module, the crossbar interface, which is shared by two thread units, had to be addapted to support new crossbar packages. Changes to the crossbar itself where not required.

The changes to the instruction decoder were straight forward. Around 59 new instructions were added to support the different synchronization features, including instructions for hardware locks, which are not part of this paper. The Storage Interface (SI) required more extensive changes, because we added the E-SSB in this module. This was the actual E-SSB buffer and the assosciated control logic. The data routing inside the storage interface had also to be adapted to support request from and to the E-SSB logic. In addition, some of the instructions require more than one result register. Due to restrictions in the instruction format, crossbar package format, and in the register file, we use the result register and the next following register as bundled result registers. For example the Single-Writer-Single-Reader instruction swsr1_rd r6,r8 reads a signed double word value from the address specified in register r8. The return code is written to register r6 and the value is written to register r7. The write-back register is selected to be the next register after the return-code register in the register file. The SI in the TU was adapted to handle this special case and to generate crossbar packages for the new instructions if necessary. Memory requests for the SRAM originally could originate from the TU or the network. A simple Least-Recently-Used (LRU) schema is used to arbitrate between these requests. With the addition of E-SSB, the arbitration schema had to be modified. Since both TU and network can produce E-SSB requests, a LRU schema is used at the entrance of E-SSB. We also keep the LRU schema for the normal memory requests to the SRAM, but if E-SSB has a memory request for the SRAM it takes priority over all other memory requests for the SRAM. This approach is still fair, because LRU was already performed at the entrance of E-SSB and due to the serialization effects of the crossbar interconnect.

The E-SSB in this implementation is a 16-entry 8-way associative buffer and 46 bit wide for each entry. The required fields for a E-SSB entry in this architecture are: *State*, *Cnt*, *Address*, *Processor ID (PID)*, *Thread ID (TID)*, *GPR*, *Size*, and *Local*. The size of each field is shown in Table 1.

The E-SSB creates special network return packages to accommodate support for E-SSB return codes, interrupts and performance counter events. The format of the E-SSB return packages is shown in Table 2. *TrCode* specifies the type of package. *Int* is used to raise a

Table 1: E-SSB Entry

Field	State	Cnt	Address	PID	TID	GPR	Size	Local
Size(bit)	4	8	15	7	3	6	2	1

E-SSB interrupt. The interrupt is always risen in the TU that produced it and not in the TU were the E-SSB is located. This is necessary, because even if a TU is turned off, its SRAM can still be accessed by other TUs. *Sync* is used by the performance counter events. It indicates if a E-SSB load arrived before or after the store. *E-SSB Code* is sign-extended to 64bit and written to the register specified in the *GPR* field. *PID* and *TID* are used by the crossbar for routing. *Error* has the same behavior as for normal memory operations and raises an External interrupt. This normally happens when a user level store tries to access protected data or a load/store access is outside the valid memory space. The content of the *Data* field is written to register *GPR*+1.

Table 2: E-SSB Return Package

Field	TrCode	Int	Sync	E-SSB Code	PID	TID	Error	GPR	Data
Size(bit)	6	1	1	3	7	3	1	6	64

3.4 Logic Resource Usage for Extended Synchronization State Buffer (E-SSB)

New architectural features may sometimes be implemented very easily, but the associated hardware cost can be overwhelming and not feasible to be implemented in hardware. We did a comparison of the Cyclops-64 design with and without E-SSB. We converted the HDL code to VHDL and synthesized it with the design compiler, using the generic technology independent libraries (GTECH), to generate a VHDL netlist. Then we used a tool to analyse the VHDL netlist and calculated the number of each design primitive. The design primitives reported for this study are NOT, AND, OR, XOR, Flip-Flops (FF), and SRAM. An exact gate number cannot be given, because this depends on the feature size of the process and the specific component libraries of the semiconductor foundry. The implementation of the first two Single-Writer-Single-Reader Modes (Mode 1 and 2) required additional buffers in the crossbar interface, which is solely responsible for an increase of 76,000 FF in the whole system. We only implemented the first two modes to have a fair comparison for benchmarking. In the final architecture it would not be necessary to implement all three modes and these additional FF will not be required. We still list them here for completeness to represent the current design.

4 Evaluation

In this section we first introduce the experimental testbed, which was used to emulate the Cyclops-64 design. Then we present the results obtained from the experimental testbed, using

Table 3: Logic resource usage of the Cyclops-64 architecture.

Design Primitive	Original	with E-SSB	Increase
NOT	6,946,100	7,364,740	6.03%
AND	10,924,586	11,779,946	7.83%
OR	5,812,398	6,257,358	7.66%
XOR	1,171,951	1,200,671	2.45%
FF	2,140,299	2,350,619	9.83%
RAM(bit)	50,318,560	51,260,640	1.87%

the case study presented in Section 2.

4.1 Experimental Testbed

For experimental performance evaluation, we implemented the proposed Extended Synchronization State Buffer (E-SSB) at the Hardware Description Language (HDL) of the Cyclops-64 (C64) architecture. Moreover, we use the Delaware End-to-end Emulation Platform (DEEP) to emulate this many-core architecture. We selected this FPGA based emulator due to several of its properties. This emulation platform is fast and cycle-accurate if compared with software based methods. It is capable of emulating the whole many-core design with a relative small number of FPGAs (32 Altera Stratic II) thanks to the Delaware Iterative Multiprocessor Emulation System (DIMES) mode. Since the whole Cyclops-64 design cannot be fitted into a single FPGA, neither the ones in DEEP nor any other FPGA on the market today; the design is broken down into sub-modules. These sub-modules fit on a single FPGA, but many FPGA will be required to run the entire system and the communication overhead will be very high. On the other hand, DEEP, running on DIMES mode, takes an iterative emulation approach [15]. Combinatorial logic equivalent sub-modules are implemented on only one (or a few) FPGA(s), and then they are iteratively utilized to emulate all instances of the sub-module. Moreover, stateful elements, like Flip-Flops (FF) and internal RAM blocks, are isolated and kept independent of the sub-module instance. By using this approach, the required number of FPGAs to run the design is drastically reduced. All the steps described above are done automatically by the DEEP software stack. Finally, thanks to its debugging facilities and modes, a design can be quickly debugged and run. For more information about the DEEP system and its various modes of operations (including DIMES), please refer to [16]. In the case of the C64 design (with E-SSB) the average emulation speed is around 20k cycles per second on DEEP.

4.2 Summary of Major Results

From on our experimental results we like to highlight the following key observations:

Observation 1 (See Section 4.3) Non-strict synchronization operations incure no overhead.

Observation 2 (See Section 4.4) Strong scalability even for small problem sizes.

4.3 Overhead

This benchmark tests the overhead of a successful synchronization, that means that the read arrives after the write has completed. This benchmark is exactly the same as the one performed by the original SSB paper [11]. A loop is executed 10,000 times by two threads. One thread performs a write and then synchronizes on a barrier, the other thread synchronizes on the barrier first and then performs a load to the same memory location as the store. This reference time is then compared to a version where the normal load and store operations are replaced with E-SSB operations. The overhead of the different E-SSB operations is shown in Table 4. SWSR 1, SWSR 2 and SWSR 3 are the different E-SSB operations which have been introduced in Section 3.2. From these results we can conclude that our non-strict synchronization operation incurs negligible overhead. This one cycle overhead comes from the E-SSB buffer, which adds one extra cycle on every E-SSB memory operation.

E-SSB Operation	SWSR 1	SWSR 2	SWSR 3
Overhead E-SSB (cycles)	27	33	1

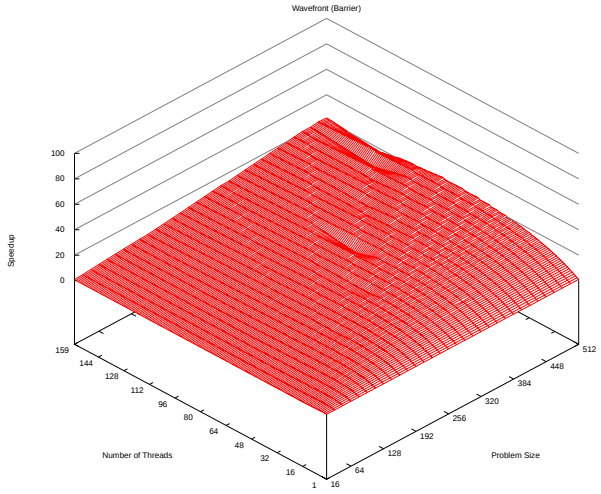
Table 4: Overhead of successful synchronization

4.4 Scalability

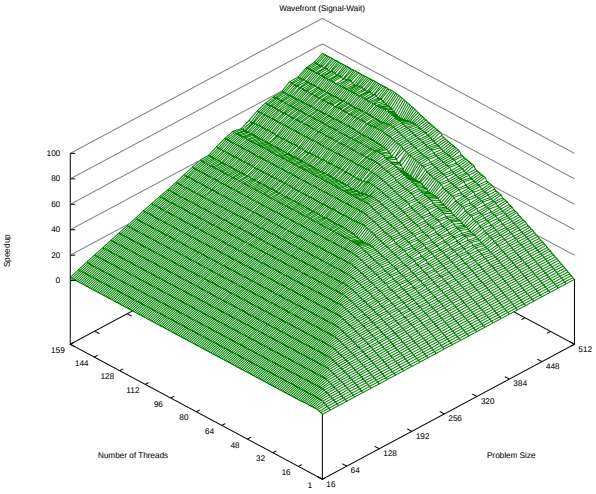
We implemented the wavefront computation kernel in six different version as described in Section 2. The different versions are Serial, Barrier, Signal-Wait, SWSR1, SWSR2 and SWSR3. All kernels were hand-coded in assembly. In all versions, the inner loop is unrolled four times to reduce the overhead of the synchronization and allows for a better overlapping of memory and arithmetic computation. We run the benchmark on the emulation system for problem sizes starting at 16x16 at increments of 16 up to the maximum supported problem size of 512x512 elements. These problem sizes can be emulated on the emulation engine in a reasonable time. For each problem size we run the wavefront benchmark with different number of threads. Starting from one thread all the way up to 159 threads[1] at increments of one. The runtime was calculated only for the kernel and the speedup was calculated based on the results of the serial version. Figure 9 shows the speedup of the different parallel versions. The rifts shown in Figure 9(e) is due to congestion on the crossbar ports and memory banks.

Even though not shown in this paper we would like to mention that we also implemented and run a Successive over-relaxation (SOR) benchmark based on the Jacobi method for solving linear systems. Compared to the wavefront example the SOR benchmark requires the data of

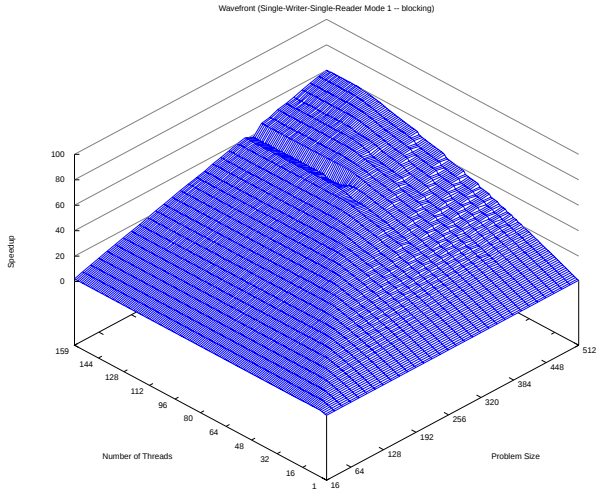
[1]The architecture supports up to 160 hardware threads, but only 159 can be used, because the OS kernel is running on the first thread unit



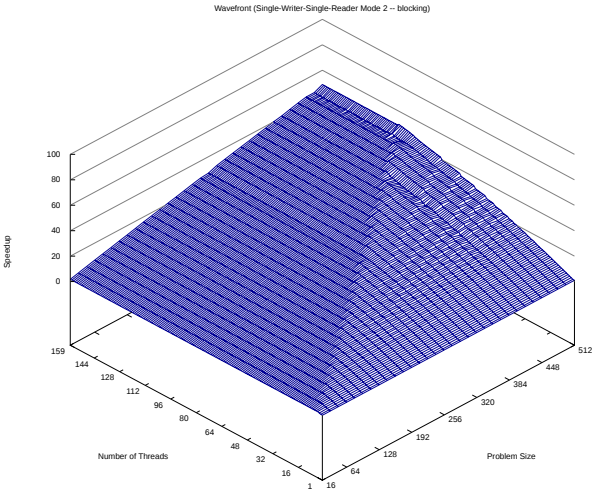
(a) Wavefront (Barrier)



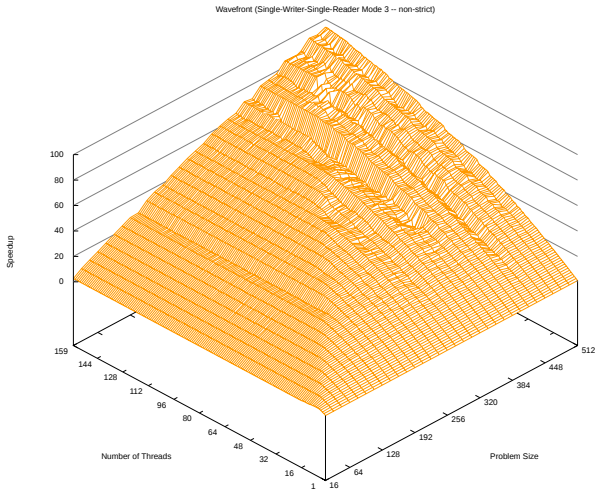
(b) Wavefront (Signal-Wait)



(c) Wavefront (SWSR1)



(d) Wavefront (SWSR2)

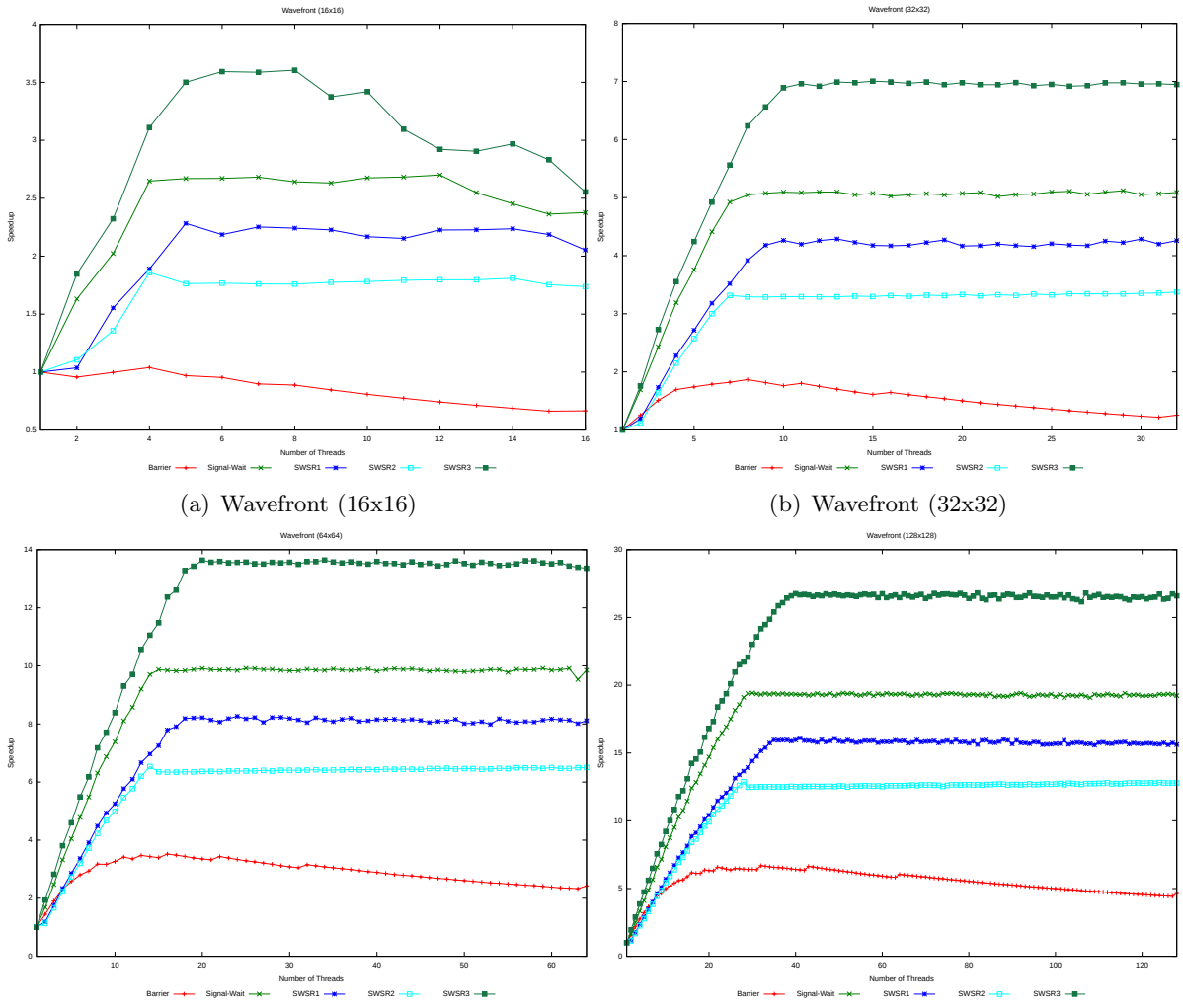


(e) Wavefront (SWSR3)

Figure 9: Wavefront Speedup

its four neighbors (north, south, east, and west). The results are comparable with the wavefront benchmark. We also also implemented a summation tree benchmark to show the advantages of non-strict synchronization for data dependencies which cannot be statically determined during compilation time. Due to space limitations, detailed results for both of these benchmarks are not presented in this paper.

The scalability of the different implementations for small problem sizes is shown in Figure 10. It can be seen that for the given problem sizes SWSR3 scales better with the number of cores and the maximum speedup is always achieved with SWSR3, too.



(a) Wavefront (16x16)

(b) Wavefront (32x32)

(c) Wavefront (64x64)

(d) Wavefront (128x128)

Figure 10: Wavefront Scalability

5 Related Work

Our research was greatly influenced by previous work on fine-grain synchronization constructs by academia and industry. This includes research on dataflow constructs like the I-Structure [1], the Synchronization State Buffer (SSB) [11], and the Terra MTA /Cray XMT [6, 9]. The use of tagged memory, Full/Empty bits, and I-Structure has been explained in Section 1. E-SSB differs in the following aspect from previous work. It enables "virtual tagging" of the whole memory space like SSB, plus it supports all data sizes of the architecture and it is not limited to double-word synchronization. Furthermore, it has been enhanced to support non-strict synchronization. It has the benefits of SSB, that means using less hardware resources, and the non-strict behaviour of I-Structures. Another approach that gained momentum in recent years is Transactional Memory (TM) [17, 18], which also employs a non-blocking synchronization approach. The major difference to our approach is that if a transaction fails, all changes done inside a transaction must be rolled back and the transaction has to be restarted. This results in unneccessary computation everytime a transaction has to be restarted.

6 Conclusion and Future Work

In this paper we presented a new design for non-strict fine-grain synchronization, based on the Synchronization State Buffer (SSB) [11], and its implementation at the Hardware Description Level (HDL) of a "real" many-core architecture. Our experiments were performed on an emulation engine with gate-level accuracy. The results surpassed our expectations and show very good scalability for even small problem sizes. Even for larger problem sizes our non-strict synchronization approach surpasses all other synchronization constructs, like barriers with hardware support and signal-wait. The most noticeable result is that we achieve scalability beyond the 100 core barrier. E-SSB is the first step towards a new paradigm in code generation for many-core architectures and we intend to exploit this in future compiler research.

References

[1] R. Arvind, R. Nikhil, and K. Pingali, "I-Structures: Data Structures for Parallel Computing," *TOPLAS*, vol. 11, no. 4, pp. 598–632, 1989.

[2] G. Papadopoulos and D. Culler, "Monsoon: An Explicit Token-Store Architecture," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 3a, pp. 82–91, 1990.

[3] J. Dennis, "The Evolution of 'Static' Dataflow Architecture," *Advanced Topics in Data-Flow Computing*, pp. 35–91.

[4] K. Traub, "A compiler for the MIT tagged-token dataflow architecture," 1986.

[5] B. J. Smith *et al.*, "Architecture and Applications of the HEP multiprocessor computer system," *Real-Time Signal Processing IV*, vol. 298, pp. 241–248, 1981.

[6] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera computer system," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 3b, pp. 1–6, 1990.

[7] W. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills, "Architecture of a message-driven processor," in *Proceedings of the 14th annual international symposium on Computer architecture*, pp. 189–196, ACM New York, NY, USA, 1987.

[8] A. Agarwal, J. Kubiatowicz, D. Kranz, B. Lim, D. Yeung, G. D'Souza, and M. Parkin, "Sparcle: An evolutionary processor design for large-scale multiprocessors," *IEEE micro*, vol. 13, no. 3, pp. 48–61, 1993.

[9] J. Feo, D. Harper, S. Kahan, and P. Konecny, "Eldorado," in *Proceedings of the 2nd conference on Computing frontiers*, p. 34, ACM, 2005.

[10] P. Barth and R. Nikhil, "M-Structures: Extending a Parallel, Non-strict, Functional Language with State," in *Functional Programming Languages and Computer Architecture*, pp. 538–568, Springer.

[11] W. Zhu, V. Sreedhar, Z. Hu, and G. Gao, "Synchronization State Buffer: Supporting Efficient Fine-Grain Synchronization on Many-Core Architectures," in *Proceedings of the 34th annual international symposium on Computer architecture*, p. 45, ACM, 2007.

[12] Y. Zhang, T. Jeong, F. Chen, H. Wu, R. Nitzsche, and G. Gao, "A study of the on-chip interconnection network for the IBM Cyclops64 multi-core architecture," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, p. 10, 2006.

[13] J. del Cuvillo, W. Zhu, Z. Hu, and G. Gao, "TiNy Threads: A Thread Virtual Machine for the Cyclops64 Cellular Architecture," in *19th IEEE International Parallel and Distributed Processing Symposium, 2005. Proceedings*, p. 8, 2005.

[14] J. Del Cuvillo, W. Zhu, Z. Hu, and G. Gao, "Toward a Software Infrastructure for the Cyclops-64 Cellular Architecture," in *High-Performance Computing in an Advanced Collaborative Environment, 2006. HPCS 2006. 20th International Symposium on*, pp. 9–9, 2006.

[15] H. Sakane, L. Yakay, V. Karna, C. Leung, and G. Gao, "DIMES: An iterative emulation platform for multiprocessor-system-on-chip designs," in *2003 IEEE International Conference on Field-Programmable Technology (FPT), 2003. Proceedings*, pp. 244–251, 2003.

[16] J. Ributzka, Y. Hayashi, and G. Gao, "CAPSL Technical Memo 96: Design and Integration of New Architecture Features into a Many-Core Chip Architecture - A Report on a Novel Architecture/Software Co-Verification Platform," April 2010.

[17] M. Herlihy and J. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th annual international symposium on Computer architecture*, p. 300, ACM, 1993.

[18] B. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Minh, C. Kozyrakis, and K. Olukotun, "The Atomos Transactional Programming Language," *ACM SIGPLAN Notices*, vol. 41, no. 6, p. 13, 2006.