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

The Elephant and the Mice: The Role of Non-Strict Fine-Grain Synchronization for Modern Many-Core Architectures

Juergen Ributzka, Yuhei Hayashi, Joseph B. Manzano and Guang R. Gao

CAPSL Technical Memo 99
Revised on March 31, 2011

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

The Cray XMT architecture has incited curiosity among computer architects and system software designers for its architecture support of fine-grain in-memory synchronization. Although such discussion go back thirty years, there is a lack of practical experimental platforms that can evaluate major technological trends, such as fine-grain in-memory synchronization. The need for these platforms becomes apparent when dealing with new massive many-core designs and applications.

This paper studies the feasibility, usefulness and trade-offs of fine-grain in-memory synchronization support in a real-world large-scale many-core chip (IBM Cyclops-64). We extended the original Cyclops-64 architecture design at gate level to support the fine-grain in-memory synchronization feature. We performed an in-depth study of a well-known kernel code: the wavefront computation. Several versions of the kernel were used to test the effects of different synchronization constructs using our chip emulation framework. Furthermore, we tested selected OpenMP kernel loops against existing software-based synchronization approaches.

In our wavefront benchmark study, the combination of fine-grain dataflow-like in-memory synchronization with non-strict scheduling methods yields a thirty percent improvement over the best optimized traditional synchronization method provided by the original Cyclops-64 design. For the OpenMP kernel loops, we achieved speeds of three to fourteen times the speed of software-based synchronization methods.

1 Introduction

During the 1970's and 1980's, a novel computational model was introduced by Dennis et al. [1] named Dataflow. Under this model, computation "flows" according to the availability of data, which means that several operations can run in parallel if the dependent data is available to them (and there are free resources to run them). Under the umbrella of Dataflow, several interesting structures and methods were proposed, like the actor's 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 non-strict 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 a consumer operation (i.e. read) to be issued before a producer operation (i.e. write) is issued or completed. The consumer operation will have to wait until the producer operation completes. However, the waiting happens on the I-Structure construct and frees the processor (i.e. producer and/or consumer) to do other useful work. This non-blocking issuing behavior is what we call the leniency property of the I-Structure. Another property of the I-Structure is that it allows a true data centric 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 or 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 1990's, Dataflow research was gently

nudged out of mainstream computing. Due to the emergence of multi-core 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 renaissance.

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 that exhibit data races, 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) [5, 6] and wavefront communication type algorithms like Sweep3D [7]. Some of these problems can be parallelized by program re-structuring or by 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) can 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's and the toolchain's ability to schedule and reorder instructions. The reason for this is that although the signal and the memory operation are decoupled, they need to be scheduled in a very restricted manner, affecting other unrelated memory operations. 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 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 data sets.

Coarse-grain synchronization constructs like signal-wait cannot take full advantage of parallelism due to their 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 [8], Monsoon [2], the Tera MTA family of processors [9], MDP [10], Cedar [11], Multicube, KSR1, Alewife/Sparcle [12], the M-Machine [13],

the J-Machine [14], ElDorado (aka Cray XMT) [15] and others. One popular way to implement the fine-grain constructs is to add an extra bit, called the full/empty bit, to each memory location. This enhancement, along with the addition of several extensions to the Instruction Set Architecture (ISA) to handle the full/empty bit, allows fast and efficient fine-grain dataflow-like synchronization. Since these bits are in each memory location, a synchronized operation will only complete if the memory word is in a pre-determined state (e.g. for loads the full/empty bit must be "full" and for stores the full/empty bit must be "empty"). Upon completion of the operation and according to the instruction type, the state of the memory location might change to a different state or stay the same. 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 the point of evaluation. If the value is evaluated when it is requested, it is called strict. If the value is evaluated when it is needed, it is called non-strict or lenient. 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.

Even though the addition of the extra bit to each memory location allows the implementation of fine-grain synchronization constructs, its cost might be very high. The Synchronization State Buffer (SSB) from Zhu et al. [16] 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 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/leniency 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. We implemented the E-SSB at the gate-level using the hardware description language (HDL) code of the original Cyclops-64 architecture and extended it with our E-SSB implementation. A more detailed description of the Cyclops-64 architecture is given in Section 3.1. 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

In the following sections we answer these 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 3.4 gives a more

detailed description of the changes that were necessary 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 unknown until an actual implementation has been performed. In Section 3.5 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 do we 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 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 the system's usefulness for whole chip and system software validation.

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 1. 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 2.

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

Figure 1: C Code of the Wavefront Kernel

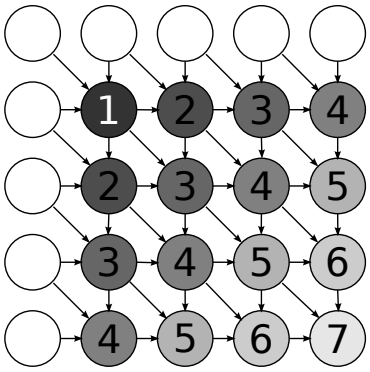


Figure 2: Wavefront Dependencies: Elements are calculated based on the values from the left, top-left and top element.

We have implemented this case study benchmark for the Cyclops-64 many-core architecture with the following synchronization constructs: barrier, signal-wait, and fine-grain in-memory 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 the issuing of 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 schedule with barriers is shown in Figure 3.

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 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

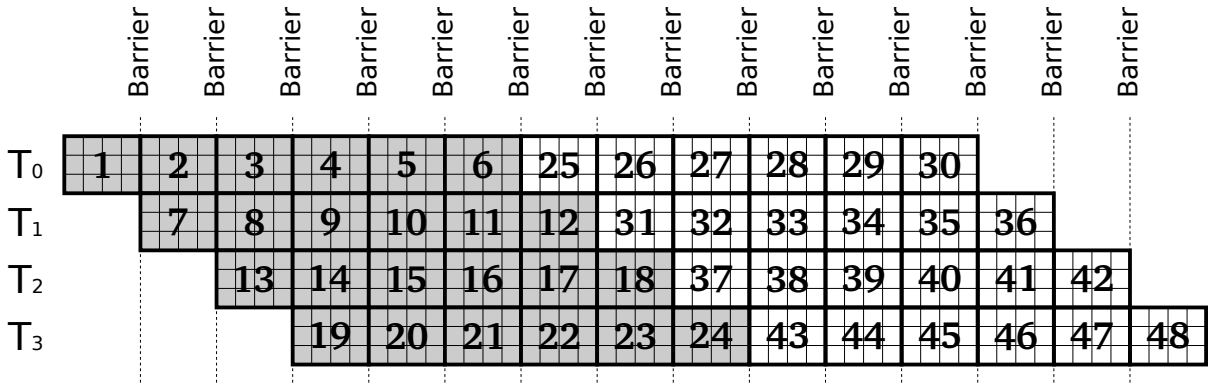


Figure 3: Wavefront Schedule with Barriers: The 2D array is partitioned into blocks, which are distributed across threads.

(i.e. no dependencies with the slowest thread). This unnecessarily *harsh* synchronization takes its toll and the problem is further aggravated 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 either. Figure 4 shows the speedup for the barrier version of the benchmark with a maximum speedup of 24x. More details can be found in Section 4.2.

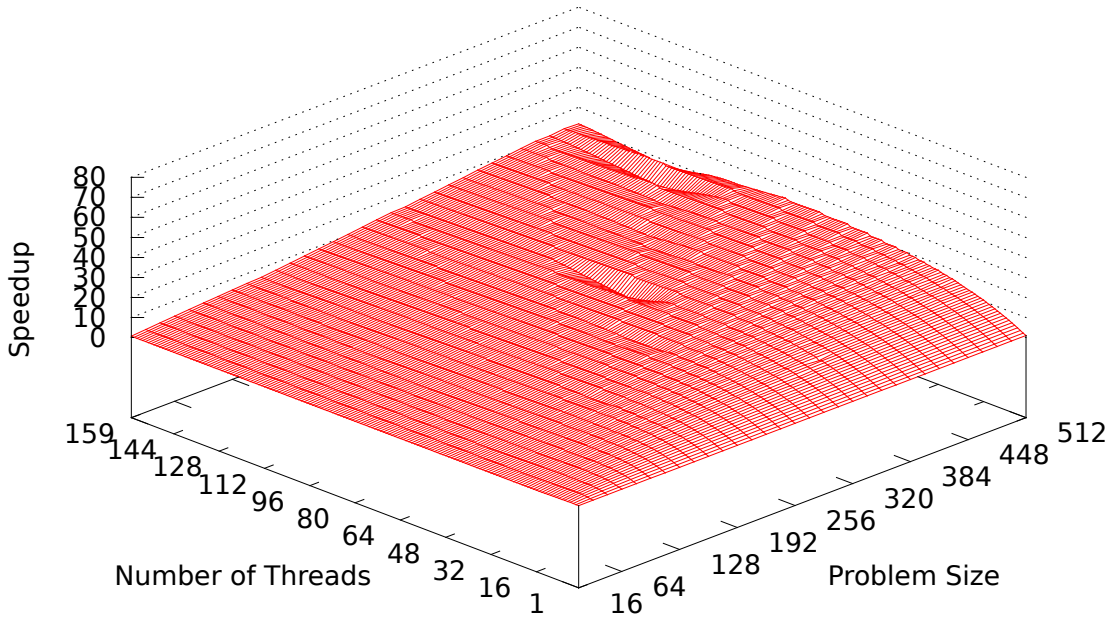


Figure 4: Wavefront Speedup (Barrier)

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. Figure 5 shows the speedup for the signal-wait version of the benchmark with a maximum speedup of 72x. More details can be found in Section 4.2.

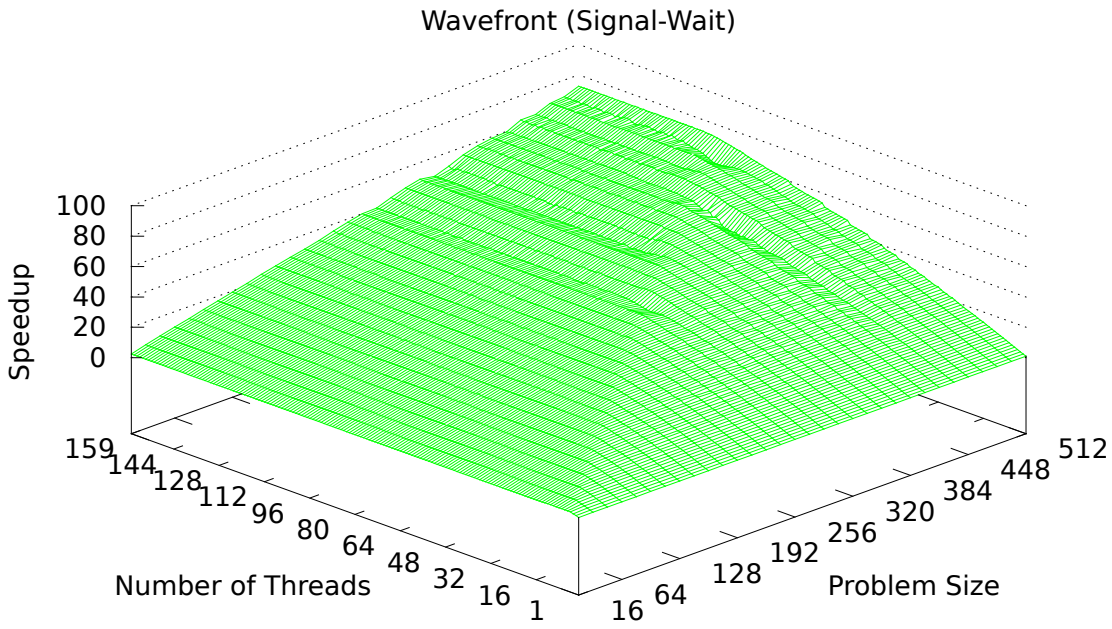


Figure 5: Wavefront Speedup (Signal-Wait)

2.3 Wavefront with Fine-Grain In-Memory Synchronization

The last synchronization construct we used in this experimental study is an dataflow-like fine-grain in-memory synchronization method. In fact, we used three different versions of this 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 blocking, while the last one is asynchronous/non-blocking. We used the same approach as signal-wait, but we replaced the *Signal* and *Wait* functions with 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 unfortunately still slower than the signal-wait implementation. This was due to the blocking behavior 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 change, we achieved promising 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. Figures 6 to 8 show the speedups for the different fine-grain synchronization versions of the benchmark with a maximum speedup of 60x, 50x and 94x, respectively. More details of the results are presented in Section 4.2.

3 Design and Implementation of Fine-Grain Synchronization

Before we go into the details of the design and implementation of fine-grain synchronization, we will first introduce the Cyclops-64 many-core architecture. We will then 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 and 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 contributes to the Global Interleaved Shared Memory; the other section can be used as Scratch Pad Memory (SPM). A TU has direct, low-latency access to its own SPM. The SPM of other TUs can still be accessed through the crossbar. Sequential Consistency is guaranteed for the Global Interleaved Shared Memory, but

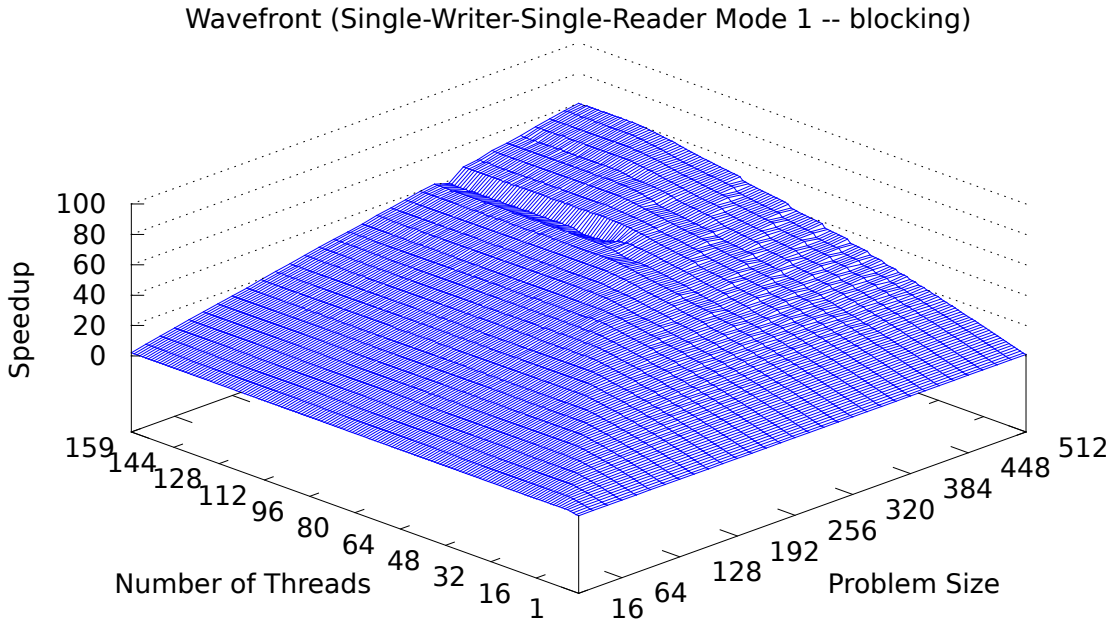


Figure 6: Wavefront Speedup (Single-Writer-Single-Reader 1)

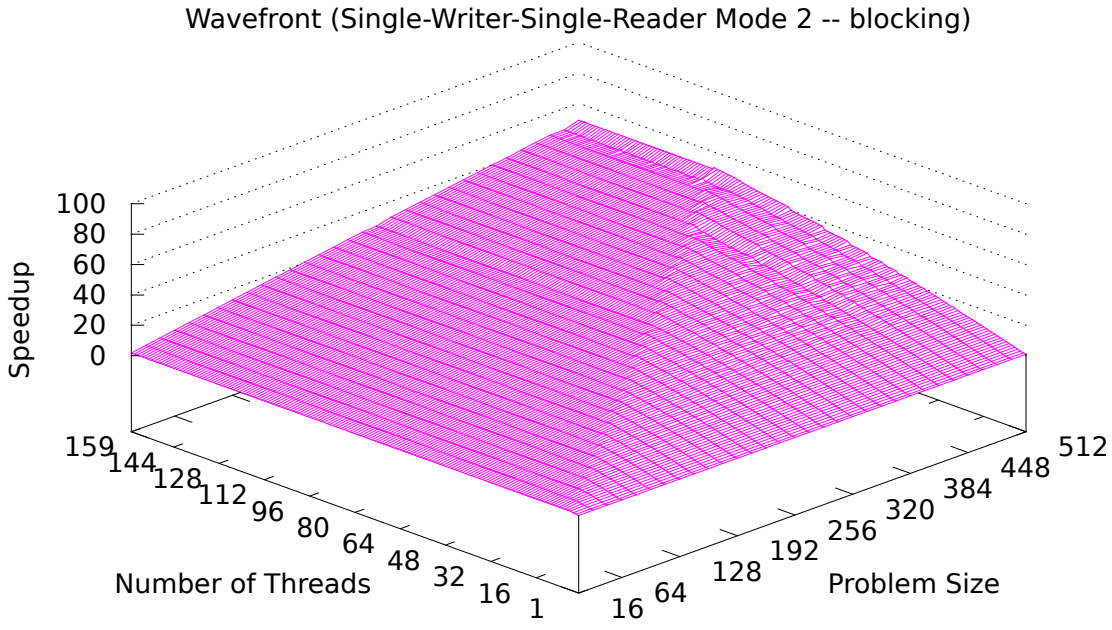


Figure 7: Wavefront Speedup (Single-Writer-Single-Reader 2)

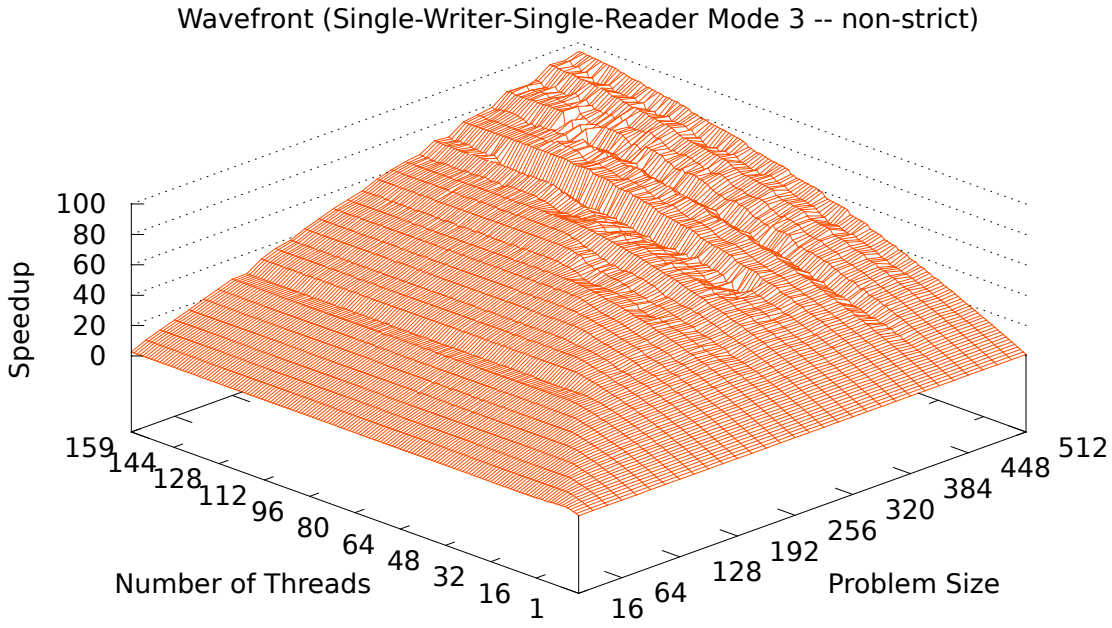


Figure 8: Wavefront Speedup (Single-Writer-Single-Reader 3)

not for the SPM. TUs are in-order single-issue cores and use scoreboarding for out-of-order completion. They have a quad-ported register file (two read ports and two write ports) with 64×64 bit 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. Off-chip DDR2 memory is connected through four on-chip DDR2 memory controllers and each memory controller is connected to its own crossbar port. Each chip can be connected to six neighboring chips in a 3-D mesh network. The network switch is also integrated into the chip and has seven connections to the crossbar. The host interface is connected to one crossbar port. In summary, the chip's crossbar interconnect possesses a total of 96 ports: eighty for the processors, four ports for the IC, four ports for on-chip DDR2 memory controllers, seven ports for inter-chip communication, and one port for the host interface. A logical overview of the chip is shown in Figure 9.

The architecture uses an 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 elsewhere [17–19].

3.2 SSB: A Recap

The Synchronization State Buffer (SSB) proposed by Zhu et al. is based on the observation that in any synchronized program only a small number of synchronized variables are needed at any

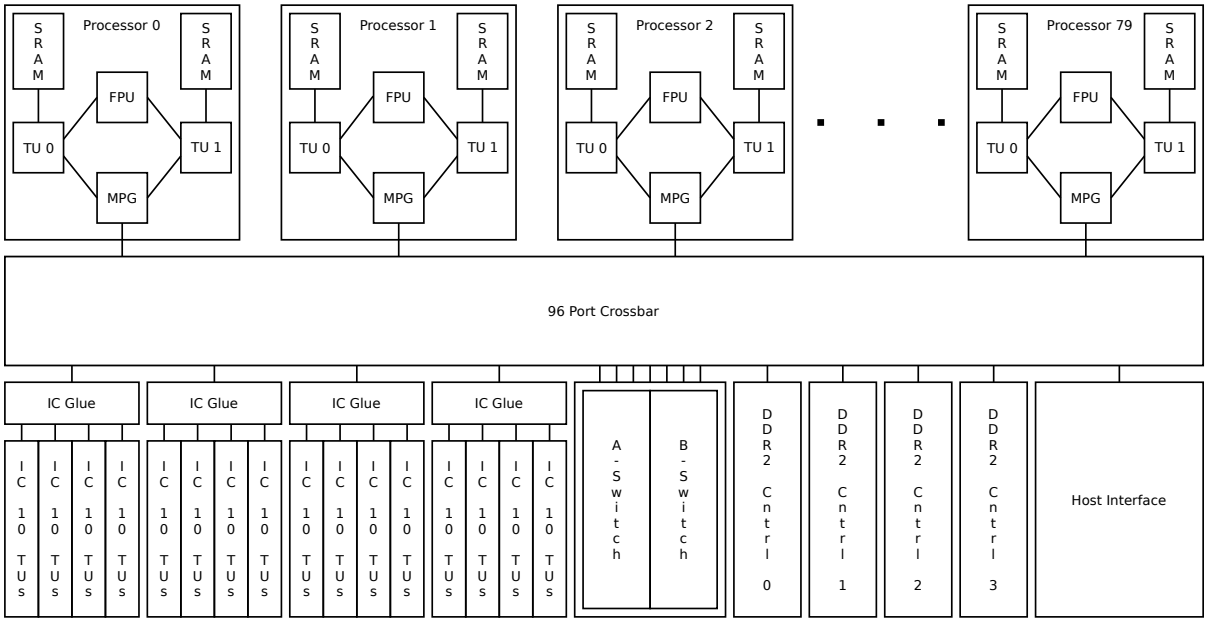


Figure 9: IBM Cyclops-64 (C64) Many-Core Architecture: The architecture consists of 80 processors (Processor 0 -79). Each processor has two Thread Units (TUs) called TU 0 and TU 1. Both share one Floating-Point Unit (FPU) and one crossbar port (MPG). Each TU is connected to a SRAM bank, which can be 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.

point in time [16]. This means that a small buffer (added to each memory controller) is sufficient to keep the synchronization metadata of these variables. This reduces the overhead of keeping N bits for each memory word in the system as presented in other solutions [15]. Moreover, this buffer can store additional metadata for a specific variable for enabling such features as memory-based pointer forwarding and debugging/tracing capabilities.

In this paper we will only describe the usage of the metadata as full/empty bits in the context of Single-Writer-Single-Reader (SWSR) synchronization. The information saved in an SSB entry is implementation dependent, but it requires at least four parts in the original SSB: (1) a state field to indicate the current synchronization mode; (2) a counter field; (3) a thread identifier field; and (4) an address field to specify the memory address to which the entry applies.

The original SSB design had two different SWSR modes. Mode 1 employed a busy-wait approach for the reader until the data was ready. The second mode utilized the sleep-wakeup features of the architecture to reduce crossbar traffic and energy consumption. The operational semantics for the SSB synchronization constructs are described as follows:

SSB 1: Busy-Wait If the writer is first, then an entry is created in the SSB and the status "SUCCESS" is returned to the writer. When the load arrives, it is allowed to proceed and the

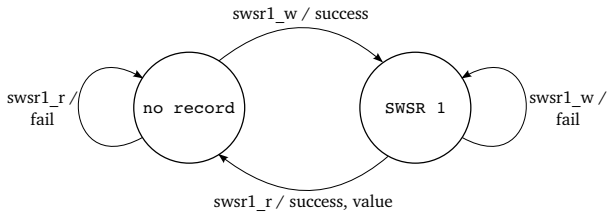


Figure 10: SSB 1: Busy-Wait

entry is removed from the SSB. The value and the status "SUCCESS" are returned to the reader. If the reader is first, then no entry is created and the status "FAIL" is returned to the reader. The reader has to retry until the status "SUCCESS" is returned. The corresponding state diagram is shown in Figure 10.

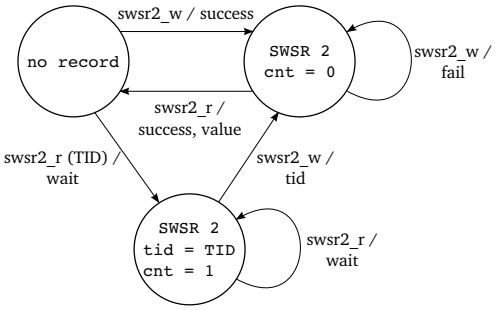


Figure 11: SSB 2: Sleep-Wakeup

SSB 2: Sleep-Wakeup If the writer is first, then an entry is created in the SSB and the status "SUCCESS" is returned to the writer. When the load arrives, it is allowed to proceed and the entry is removed from the SSB. The value and the status "SUCCESS" are returned to the reader. If the reader is first, then an entry is created and the status "WAIT" is returned to the reader. The reader goes to sleep and waits to be woken up by the writer. When the writer arrives, the Thread ID (TID) of the waiting reader is returned. The writer sends the wakeup signal to the waiting reader. The reader has now to retry the load again. This time it will succeed and the entry is removed from the SSB. The corresponding state diagram is shown in Figure 11.

If the buffer is full and a synchronization operation tries to add a new entry, an interrupt is generated and the software runtime will take control of the buffer. There is no automatic eviction of entries and flush to memory as a cache would do.

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

In this section we will explain the design principles for non-strict fine-grain synchronization and its operational semantics. We implemented the original SSB and extended it with non-strict

fine-grain synchronization. The major goal in designing our Extended Synchronization State Buffer (E-SSB) was to improve programmability and ease-of-scheduling for the compiler. Our major interest were the Single-Writer-Single-Reader (SWSR) synchronization operations. We added a third mode which eliminates the overhead of the synchronization operation with little additional hardware cost and added non-strict behavior. For the remainder of this paper we will refer to these three different modes as SSB 1, SSB 2 and SSB 3, respectively. Furthermore, we extended all modes to support any data size (byte, half word, word and double word) and signedness (signed and unsigned) of memory operations. To support these new features we extended the SSB entry with the following fields: (5) register identifier; (6) size; and (7) signdness.

The operational semantics of the non-strict synchronization is defined as follows:

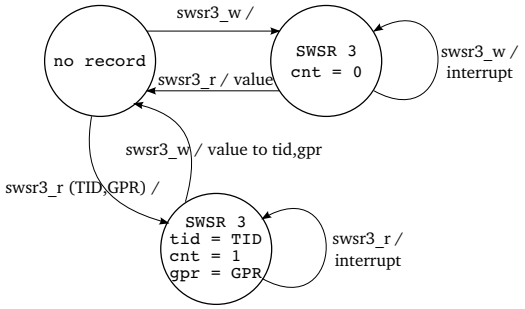


Figure 12: SSB 3: Non-Strict

SSB 3: Non-Strict If the writer arrives first, an entry is created in the E-SSB indicating this scenario. No status code is returned to the writer, as opposed to Modes 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. If the reader arrives first, an entry is created in the E-SSB to store the Thread ID and register id of the requesting thread unit, and no data is returned. While waiting for the data to return, the reader uses scoreboarding to determine if it can continue issuing other instructions that do not depend on the return value. When the writer arrives, 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. The corresponding state diagram is shown in Figure 12.

Figure 13 illustrates the advantages of the non-strict behavior of an SSB 3 load operation. Even if the load operation is outstanding, other operations can be issued in order until a dependency is met. Register dependencies are enforced by the scoreboard. Another advantage of the non-strict version is that there are no additonal crossbar packages, meaning that zero overhead is incurred. An SSB 3 load or store operation produces the exact same number of crossbar packages as their normal memory load and store counterparts.

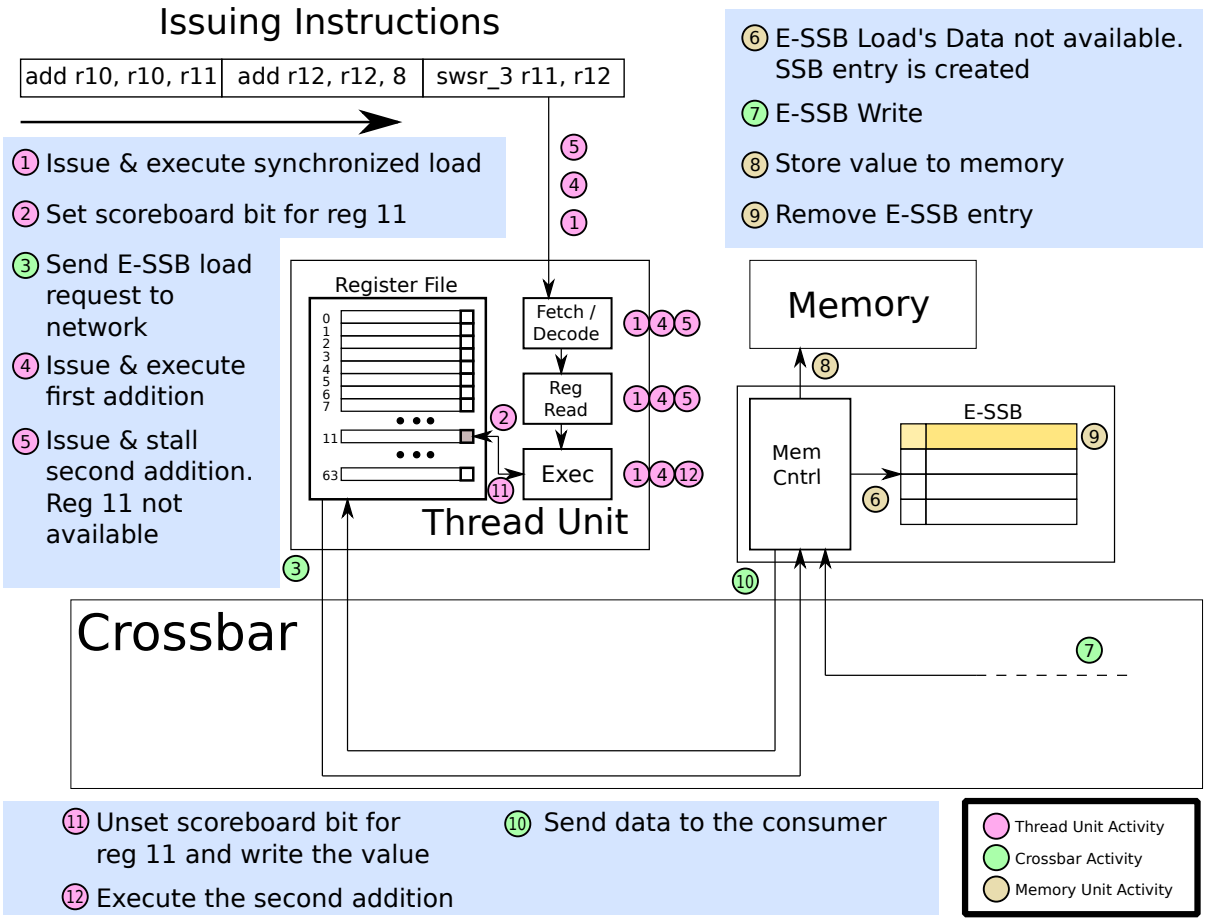


Figure 13: SSB 3 Read Example

Using these operational semantics, we implemented the E-SSB in the Cyclops-64 architecture at the HDL level. The following section gives an overview of the required changes and implementation decisions.

3.4 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 there. In particular, changes were required on the instruction decoder to support the new E-SSB instructions and the storage interface, which is responsible for routing memory requests from the network and the thread unit. Another module, the crossbar interface, which is shared by two thread units, had to be adapted to support new crossbar packages. Changes to the crossbar itself were not required.

The existing design allowed for an easy extension of the instruction decoder to support the new synchronization instructions. The Storage Interface (SI) of the TU required more

extensive changes, because we added the E-SSB in this module. This was the actual buffer for the metadata and the associated control logic. The SI orchestrates the data routing between different requests coming from the network, TU, and E-SSB and the responses coming from the network and memory controller.

Some of the original SSB instructions require more than one result register. One register is required for the return code and one for the data. Due to restrictions in the instruction format, crossbar package format, and the register file, we use the result register and implicitly, the following register as bundled result registers. For example, the SSB 1 instruction `swsr1_rd rt,ra` reads a signed double word value from the address specified in register ra. The return code is written to register rt and the value is written to register rt+1. 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. The actual implementation of the meta-data buffer is a 16-entry 8-way associative buffer and is 47 bits wide for each entry. The required fields for an E-SSB entry in this architecture are: state (4 bits), counter (8 bits), address (15 bits), processor id (7 bits), thread id (3 bits), register id (6 bits), size (2 bits), signdness (1 bit), and bits for implementation dependent features (in this case one bit).

The E-SSB creates special network return packages to accommodate support for E-SSB return codes, interrupts and performance counter events. The interrupt is always raised in the TU that produced it and not in the TU where 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.

3.5 Logic Resource Usage of the Extended Synchronization State Buffer (E-SSB)

New architectural features may sometimes be implemented very easily, but the associated hardware cost can be overwhelming and may not be feasible to implement 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. We then used a tool to analyze 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 (SSB 1 and SSB 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.

Table 1: 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%

4 Evaluation

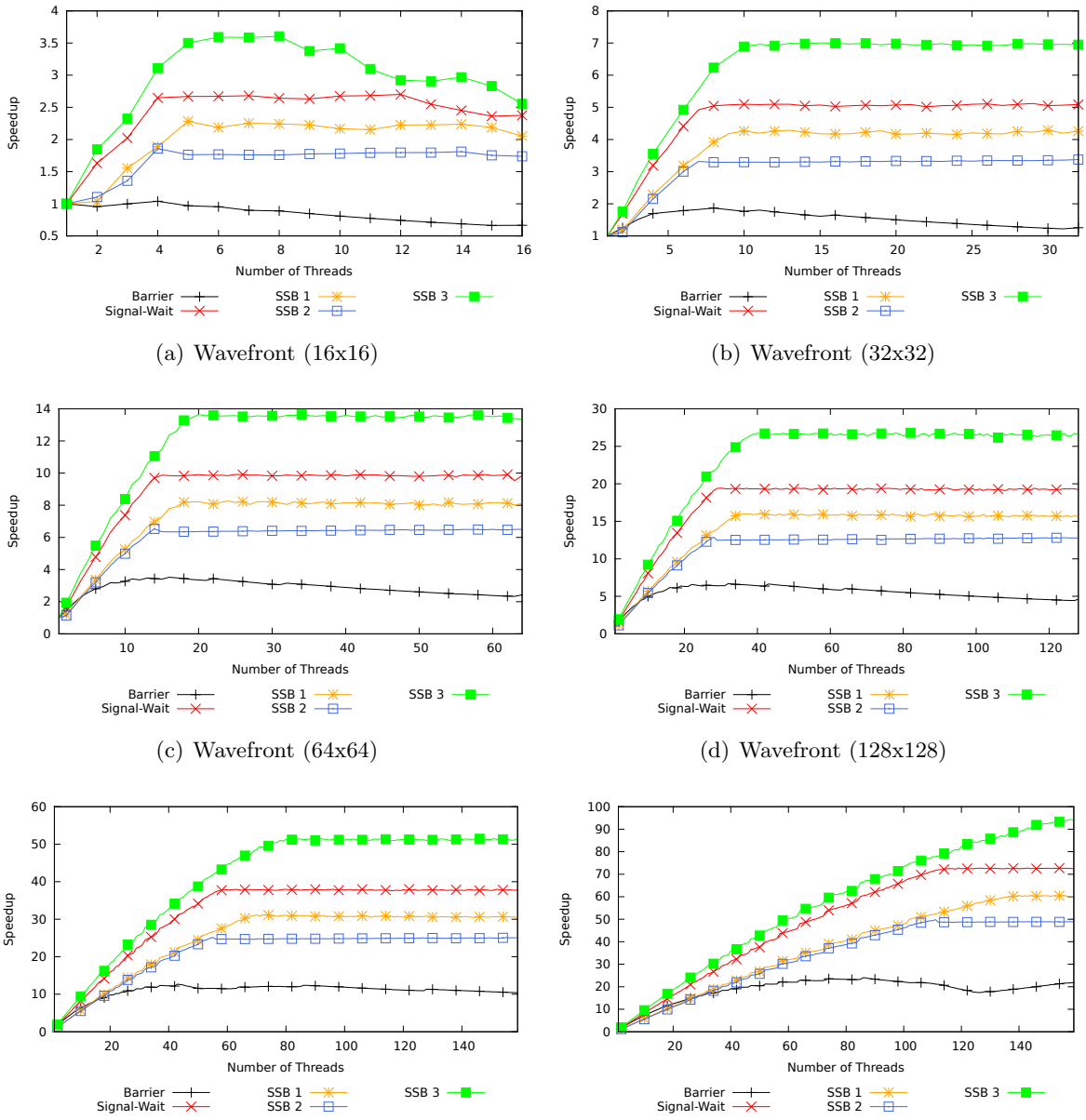
In this section we first introduce the experimental testbed, which was used to emulate the Cyclops-64 design. Then we present the results obtained from the experimental testbed, using the wavefront computation kernel and selected OpenMP kernel loops.

4.1 Experimental Testbed

For experimental performance evaluation, we implemented the proposed Extended Synchronization State Buffer (E-SSB) at the Hardware Description Language (HDL) level of the Cyclops-64 (C64) architecture. Moreover, we use the Delaware Enhanced 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 compared to software based methods. It is capable of emulating the whole many-core design with a relatively small number of FPGAs (32 Altera Stratix II) thanks to the Delaware Iterative Multiprocessor Emulation System (DIMES) mode. Since the whole Cyclops-64 design cannot fit into a single FPGA, neither the FPGAs in DEEP nor any other on the market today, the design is broken down into sub-modules. These sub-modules fit on a single FPGA, but many FPGA would be required to run the entire system and the communication overhead would be very high. On the other hand, DEEP, running in DIMES mode, takes an iterative emulation approach [20]. Combinatorial logic equivalent sub-modules are implemented on only one (or a few) FPGA(s); they are then 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 emulation 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 Ributzka et al. [21]. In the case of the C64 design (with E-SSB) the average emulation speed is around 20k cycles per second on DEEP (without using its tracing capabilities).

4.2 Experimental Results

Wavefront



(a) Wavefront (16x16)

(b) Wavefront (32x32)

(c) Wavefront (64x64)

(d) Wavefront (128x128)

(e) Wavefront (256x256)

(f) Wavefront (512x512)

Figure 14: Wavefront Speedup

We implemented the wavefront computation kernel in six different versions. The different versions are serial, barrier, signal-wait, and SSB Modes 1 to 3. 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 to allow for a better overlapping of memory operations and arithmetic

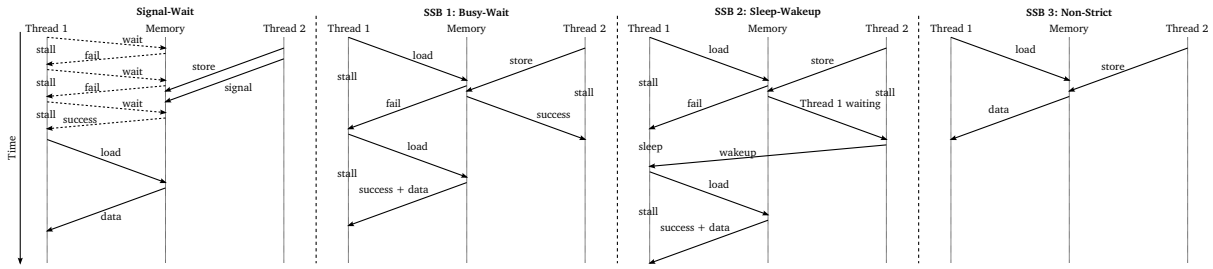


Figure 15: Synchronization Delay Illustration

computation. We run the benchmark on the emulation system for problem sizes starting at 16x16 elements at increments of 16 up to the maximum supported problem size of 512x512 elements. For each problem size, we run the wavefront benchmark with different numbers of threads, starting with one thread and going up by increments of one to 159 threads[1]. The runtime was calculated only for the kernel and the speedup was calculated based on the results of the serial version. Figure 14 shows the speedups of the different parallel versions.

Barrier: Even though the hardware-enabled barrier is very efficient, the speedup of the application is limited. The weakest link is the slowest thread. All other threads have to wait for it before they can continue doing useful work. Using barriers for these kinds of workloads is not necessarily a good choice, and dynamic scheduling approaches have achieved better results. We are aware of this, but we chose to demonstrate the barrier implementation for two important reasons. First, the barrier is supported in hardware and we wanted to compare different hardware supported synchronization constructs. Second, from a programming point of view the barriers seems to be an easy and efficient construct, because the work for each thread is the same. We wanted to show that this thinking cannot be applied anymore to many-core architectures and that congestion, bank conflicts, etc., can have unpredictable impacts on a thread's execution. The barrier version of the benchmark achieved a maximal speedup of 24x.

Signal-Wait: The signal-wait version can be implemented very efficiently on the Cyclops-64 architecture by taking advantage of the extensive atomic memory operation support and the local, low-latency scratch pad memory resulting in a speedup of 72x. Figure 15 illustrates the synchronization delay of the different benchmark versions. For all examples in this illustration, Thread 1 (consumer) always tries to read the shared data, whereas Thread 2 (producer) is producing this shared data. The first example shows the synchronization delay for signal-wait. The dashed arrows represent accesses to scratch pad memory via the back-door each thread unit has to its own scratch pad memory. This access is much faster, because it does not have to go through the crossbar. Solid arrows represents memory operations that go through the crossbar and therefore take more time. Since the consumer spins on its own local synchronization variable, changes to this variable are observed with little delay. Once the signal from the producer arrives, the consumer can continue execution without any further synchronization related stalls. This allows the overlap of computation and memory operations after the wait. The producer does

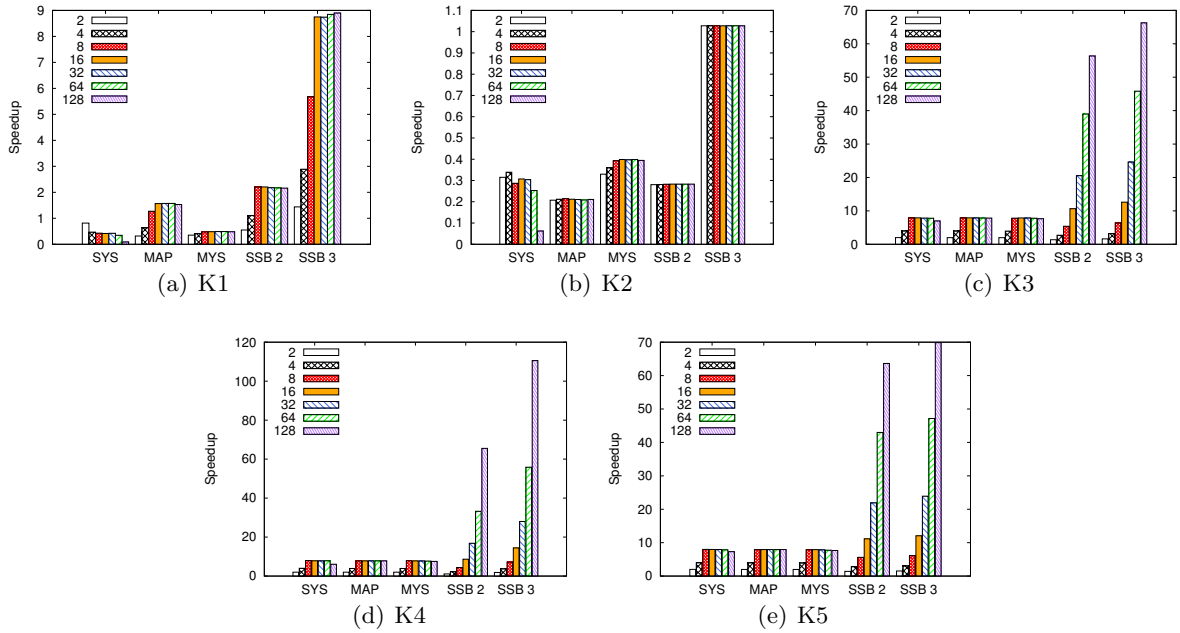
[1]The architecture supports up to 160 hardware threads, but only 159 can be used, because the OS kernel is running on the first thread unit.

not need to stall at all. This makes signal-wait a very efficient synchronization construct on Cyclops-64.

Fine-grain In-Memory Synchronization: The different SSB versions of the benchmark achieved speedups of 60x, 50x and 94x respectively. We took a closer look at the benchmarks by using performance counters. In summary, we can say that the benchmark is not memory bound. The SSB 1 (busy-wait) version has a synchronization failure rate of 150%. That means every synchronizing load operation has to be repeated 1.5 times on average, because the data had not been written yet by the producer. The SSB 2 (sleep-wakeup) version on the other hand had a failure rate of only 1-2%. Nevertheless, the SSB 1 (busy-wait) approach still achieved better speedups. The second approach generates fewer memory operations and also saves power, but the price is a longer synchronization delay, which hinders parallelism and therefore performance. The SSB 3 (non-strict) version has a failure rate of 25%, but that only means that the load arrived before the store. No additional overhead or memory transactions were required to correct this, because the memory controller had already taken care of it. The second illustration in Figure 15 shows that SSB 1 employs a similar busy-waiting approach as signal-wait, but it has to go through the crossbar every time. Furthermore, the producer and the consumer have to stall and cannot overlap any other computation or memory operations until the memory operation on their side has successfully completed. The SSB 2 sleep-wakeup approach in the next example even further aggravates this problem, because now the producer has to wake up the consumer and the synchronization delay increases further. The last SSB mode solves all the problems of the previous versions by performing the synchronization completely in the memory controller. No further action is required from the producer or the consumer. In this mode synchronizing memory operations act like normal memory operations for the thread unit and the synchronization is transparent to them. This allows aggressive scheduling of synchronizing and non-synchronizing memory operations and arithmetic instructions.

OpenMP Kernel Loops

The kernel loops are extracted from SPEC OpenMP benchmarks, such as 314.mgrid and 318.galgel. As in the original SSB paper, we compare our SSB versions against the software-based approaches proposed by Kejariwal et al. [22]. All loops exhibit the same characteristics, namely, that dependencies between loop iterations are positive and constant. They also fulfill our requirement of Single-Writer-Single-Reader, so our SSB synchronization constructs can be applied. Figure 16 shows the speedup of the different parallel versions against the sequential version. SSB 3 clearly outperforms all other versions, both software and hardware based. Another interesting aspect is that we do not lose performance when we increase the number of threads. K1's and K2's speedups are severely limited, but this is understandable and expected. K1 only performs a single arithmetic operation in the loop and therefore the speedup is clearly limited by it and the only form of parallelism can be obtained from the number of iterations that can be performed in parallel without dependence. K2's story is even worse, because the iteration dependence is 1. That means none of the iterations can be performed in parallel. Nevertheless, SSB 3 is still able to obtain instruction level parallelism between iterations through



(a) K1 (b) K2 (c) K3

(d) K4 (e) K5

Figure 16: OpenMP Loops Speedup

its fine-grain non-strict behavior and does not suffer any performance degradation as the other approaches. K3, K4 and K5 do not only provide sufficient iteration level parallelism due to a larger dependence distance of 8, but also a larger kernel that provides a great source of cross-iteration instruction level parallelism that can only be leveraged by SSB 3.

4.3 Analysis Breakdown

In this section we take an in-depth look at the different versions of the tested wavefront benchmarks. This in-depth look consists of breaking down the collected information into different important activities and overhead such as cycles spent on useful work, synchronization overhead, loop overhead, arithmetic stalls, and stalls due to synchronized and unrelated memory operations, among others. To obtain this instruction mix, we enabled the program tracing feature on the emulation engine and obtained the detailed trace of all 160 thread units. Figure 17 shows a break down of the instruction mix of the different benchmark versions. We used the maximum problem size (512x512) for the benchmark to fully utilize the whole system. The histogram shows the accumulated cycles spent by all thread units to complete the work. The serial version uses of course only one thread unit, whereas all the other versions use 159 thread units. To obtain the actual execution time, each version has to be divided by the number of threads used. "Work" contains all instructions necessary to perform the actual required computation. This includes the arithmetic instructions and the memory operations to obtain the data. "Stall Arithmetic" contains all the stall cycles in the kernel due to dependence on an unfinished arithmetic instruction. "Stall Arithmetic" can bee seen as part of "Work", because it depends on the given schedule and the arithmetic instruction latency of the architecture. "Loop Overhead" contains pointer increments, loop exit checks, branches and branch delays.

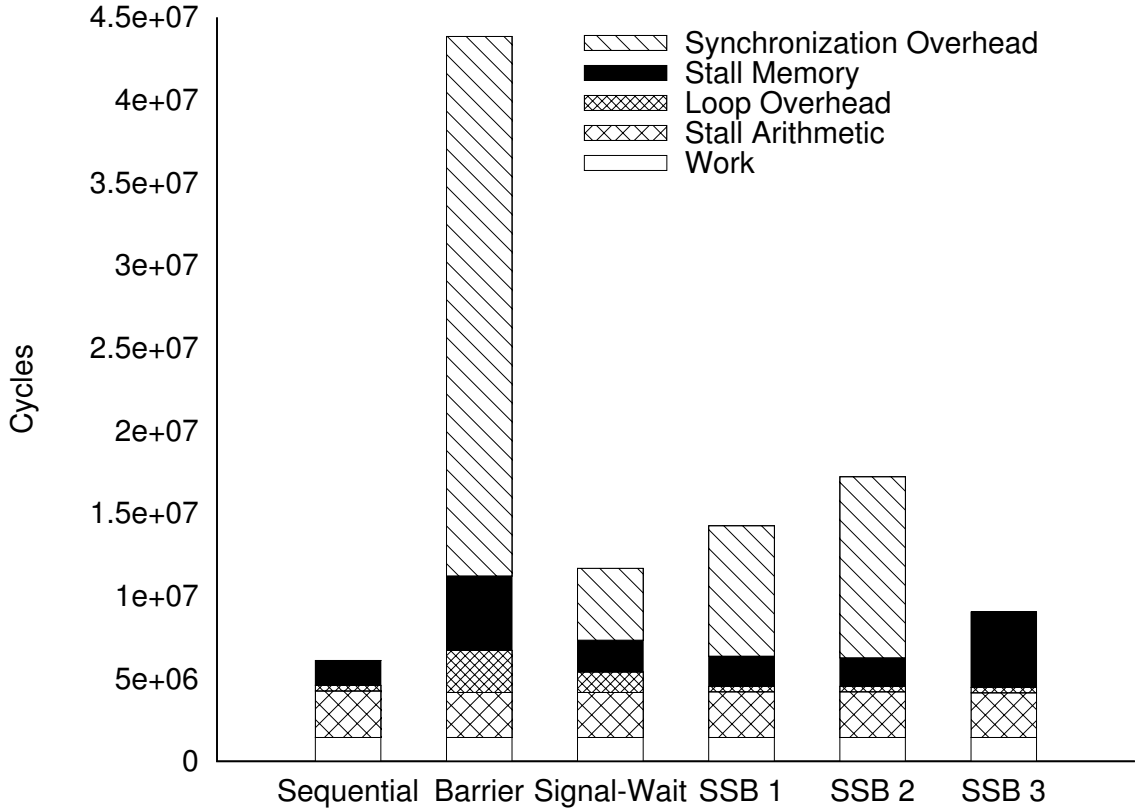


Figure 17: Wavefront Execution Runtime Breakdown: This histogram shows a breakdown of cycles spend on certain important aspects of the program. The Sequential version shows cycles spend by a single thread, whereas the other versions show the accumulated cycles spend by all 159 threads.

"Stall Memory" contains all the stall cycles inside the kernel due to instructions waiting on data to return from memory. "Synchronization Overhead" contains all the additional instructions, stalls and branch delays which were required to perform the actual synchronization. "Function Overhead" contains all the instructions, stall cycles, etc, which are not part of the kernel code, but the surrounding setup code of the function. "Stall Misc" contains stalls due to the fact that the crossbar port and the floating-point unit are shared between two thread units and other architecture related stall cycles. "Function Overhead" and "Stall Misc" are so small (less than 1%) that we omitted them in the figure.

The instruction mix break-down, overall, is predictable: "Work" and "Stall Arithmetic" are uniform across all benchmarks and indicate a compute-bound kernel. "Loop Overhead" should also be rather constant across all benchmarks, but signal-wait and barrier have a much higher loop overhead than the other versions. A detailed analysis showed that this increase is solely due to branch delays for the signal-wait version, because the loop does not fit completely in the instruction buffer. The barrier version suffers from the same problem, but it also has more complex loop exit checks on top of that. The barrier version also suffers a lot under a very large

amount of memory related stall cycles. This is due to the pathologic nature of programs that use barriers, which have normally three distinct phases. During phase one, all threads try to obtain the data at the same time, followed by the computation phase with almost no memory operations. Finally, in the last phase the results are written back to memory. This behavior makes the first phase a memory bound problem, which is responsible for the increase in memory related stall cycles.

Due to the factors described above, the main components that determine performance are the "Stall Memory" and the "Synchronization Overhead". The SSB 3 method shows no "Synchronization Overhead", because these cycles are hidden in the memory stall cycles. Thus, for a fair comparison, both of these components must be used together to evaluate our synchronization methods. Under these conditions, SSB 3 still clearly outperforms all other methods, even the highly optimized signal-wait version.

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 [23], the Synchronization State Buffer (SSB) [16], and the Tera MTA/Cray XMT [9, 15]. 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; it also 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, which means using fewer hardware resources, and the non-strict behavior of I-Structures. Another approach that gained momentum in recent years is Transactional Memory (TM) [24, 25], which also employs a non-blocking synchronization approach. The major difference with our approach is that in TM, if a transaction fails, all changes done inside a transaction must be rolled back and the transaction has to be restarted. This results in unnecessary computation every time a transaction has to be restarted. Our approach does not require this.

6 Conclusion and Future Work

In this paper, we presented a new design for a dataflow-like fine-grain synchronization, based on the Synchronization State Buffer (SSB) presented in [16], 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, such as 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 toward a new paradigm in code generation for many-core architectures and we intend to utilize this in future compiler research.

Acknowledgements

Our utmost respect goes to Monty Denneau for creating such a great architecture. We also would like to thank all the reviewers for their comments, suggestions, and help to improve this paper. This work would have not been possible without the support by NSF (CCF-0833122, CCF-0925863, CCF-0937907, CNS-0720531, and OCI-0904534), and other government sponsors.

References

[1] J. Dennis, J. Fosseen, and J. Linderman, "Data Flow Schemas," in *International Symposium on Theoretical Programming*, pp. 187–216, Springer, 1974.

[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. R. Traub, "A Compiler for the MIT Tagged-token Dataflow Architecture," 1986.

[5] K. S. Yee, "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media," *IEEE T. Antenn. Propag.*, vol. 14, 1966.

[6] A. Taflove, S. Hagness, *et al.*, *Computational electrodynamics: the finite-difference time-domain method*. Artech House Norwood, MA, 1995.

[7] A. Initiative, "The ASCI sweep3d benchmark code," 1995.

[8] B. J. Smith, "Architecture and applications of the HEP multiprocessor computer system," *Real-Time Signal Processing IV*, pp. 241–248, 1982.

[9] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera Computer System," in *Proceedings of the 4th International Conference on Supercomputing*, pp. 1–6, ACM, 1990.

[10] 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, 1987.

[11] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh, "CEDAR: A Large Scale Multiprocessor," *ACM SIGARCH Computer Architecture News*, vol. 11, no. 1, pp. 7–11, 1983.

[12] 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.

[13] M. Fillo, S. Keckler, W. Dally, N. Carter, A. Chang, Y. Gurevich, and W. Lee, "The M-Machine Multicomputer," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 146–156, IEEE Computer Society Press, 1995.

[14] M. Noakes, D. Wallach, and W. Dally, "The J-Machine Multicomputer: An Architectural Evaluation," in *ACM SIGARCH Computer Architecture News*, vol. 21, pp. 224–235, ACM, 1993.

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

[16] 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.

[17] 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 *20th International Parallel and Distributed Processing Symposium (IPDPS)*, p. 10, IEEE, 2006.

[18] 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.

[19] 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.

[20] 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.

[21] J. Ributzka, Y. Hayashi, F. Chen, and G. Gao, "DEEP: An Iterative FPGA-based Many-core Emulation System for Chip Verification and Architecture Research," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 115–118, ACM, 2011.

[22] A. Kejariwal, H. Saito, X. Tian, M. Girkar, W. Li, U. Banerjee, A. Nicolau, and C. Polychronopoulos, "Lightweight Lock-Free Synchronization Methods for Multithreading," in *Proceedings of the 20th Annual International Conference on Supercomputing*, pp. 361–371, ACM, 2006.

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

[24] 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.

[25] 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.