Landing OpenMP on Cyclops-64: An Efficient Mapping of OpenMP to a Many-Core System-on-a-Chip

Juan del Cuvillo, Weirong Zhu, and Guang R. Gao
Department of Electrical and Computer Engineering, University of Delaware
Newark, Delaware 19716, U.S.A
{jcuvillo, weirong, ggao}@capsl.udel.edu

ABSTRACT

This paper presents our experience mapping OpenMP parallel programming model to the IBM Cyclops-64 (C64) architecture. The C64 employs a many-core-on-a-chip design that integrates processing logic (160 thread units), embedded memory (5MB) and communication hardware on the same die. Such a unique architecture presents new opportunities for optimization. Specifically, we consider the following three areas: (1) a memory aware runtime library that places frequently used data structures in scratchpad memory; (2) a unique spin lock algorithm for shared memory synchronization based on in-memory atomic instructions and native support for thread level execution; (3) a fast barrier that directly uses C64 hardware support for collective synchronization. All three optimizations together, result in an 80% overhead reduction for language constructs in OpenMP. We believe that such a drastic reduction in the cost of managing parallelism makes OpenMP more amenable for writing parallel programs on the C64 platform.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors — Run-time environments, Optimization

General Terms: Languages, Measurement, Performance

Keywords: Chip Multiprocessor, System-on-a-Chip, Runtime System, OpenMP, Performance Evaluation

1. INTRODUCTION

It is increasingly clear that the huge number of transistors that can be put on a chip (now reaching 1 billion and continuing to grow) can no longer be effectively utilized by traditional microprocessor technology that only integrates a single processor on a chip. A new generation of technology is emerging by integrating a large number of tightly-coupled simple processor cores on a chip empowered by parallel system software technology that will coordinate these processors toward a scalable solution.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CF'06, May 3–5, 2006, Ischia, Italy.
Copyright 2006 ACM 1-59593-302-6/06/0005 ...$5.00.

Cyclops-64 is a petaflop supercomputer project under development at IBM Research Laboratory. The C64 is intended to serve as a dedicated compute engine originally designed for running high performance applications such as molecular dynamics to study protein folding [2], or image processing to support real-time medical procedures. To the best of our knowledge, the C64 project is one of the most ambitious projects currently under development. Unlike other academia projects, a Cyclops-64 system is planned to be delivered in 2007.

Given a machine such as the C64 cellular architecture, the challenge is to use this massive intra-chip parallelism to obtain highly sustainable performance. For fast application prototypes on a C64 chip, we are looking into different high-level programming models, and OpenMP seems to be a reasonable first candidate.

OpenMP has emerged as the industry de facto standard for writing parallel programs on shared memory systems. OpenMP specification [24, 25] provides a collection of compiler directives, library functions and environment variables, suitable for incremental and portable development of parallel applications. However, the scalability of this programming interface on systems with a large number of processing elements is sometimes limited. In some instances overheads for language constructs in OpenMP account for up to 12% of the total execution time [8] and developers are often suggested to reduce the number of parallel regions to limit the impact of these overheads [17]. As a consequence, the number of applications amenable to effective parallelization decreases dramatically.

The contribution of this paper is as follows. We have developed a mapping strategy that explores the opportunities to optimize OpenMP programs on a state-of-the-art many-core chip architecture such as the Cyclops-64. This is realized by an implementation of our mapping strategy under an OpenMP runtime library, which results in a significant performance improvement (e.g. overhead reduction as high as up to 2 orders of magnitude, and at least 80% for OpenMP language constructs). More specifically, we highlight three areas of optimizations: (1) a memory aware runtime library that takes advantage of Cyclops-64 explicit memory hierarchy by placing frequently used runtime data structures in scratchpad memories that are closer to the processor/thread units where they are used; (2) a unique spin lock algorithm designed to make best use of the C64 architecture supported in-memory atomic instructions and thread sleep/wake-up mechanisms; (3) a fast barrier synchronization based on the

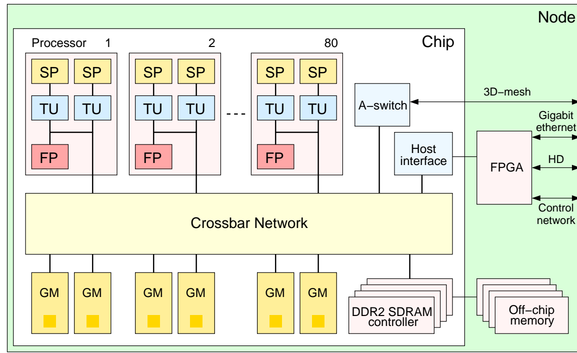


Figure 1: Cyclops-64 node

16-bit signal bus, to which all the threads on a chip are connected. All three together, result in the drastic reduction of overheads in the OpenMP runtime library and pave the way for a productive use of OpenMP as a high-level parallel programming model for the Cyclops-64 platform.

In this paper, we investigate the details that hinder parallel performance, which are hidden from the end user: overheads due to language constructs in OpenMP. With this goal, first we investigate how OpenMP constructs are implemented in the Omni OpenMP runtime library. Second, we port this compiler environment to the Cyclops-64 using the C64 toolchain and TNT microkernel 1.5 release. Third, we redesign key aspects of the library to make it suitable for the C64 computing environment. Finally, using EPCC microbenchmarks we compare the original and enhanced libraries and observe an 80% overhead reduction for OpenMP language constructs.

2. CYCLOPS-64 CHIP ARCHITECTURE

The Cyclops-64 (C64) is the latest version of the Cyclops cellular architecture designed to serve as a dedicated petaflop compute engine for running high performance applications. A C64 supercomputer is attached to a host system through a number of Gigabit Ethernet links. The host system provides a familiar computing environment to application software developers and end users.

A C64 is built out of tens of thousands of C64 processing nodes arranged in a 3D-mesh network. Each processing node consists of a C64 chip, external DRAM, and a small amount of external interface logic. A C64 chip employs a many-core-on-a-chip design with a large number of hardware thread units, half as many floating point units, embedded memory, an interface to the off-chip DDR2 SDRAM memory and bidirectional inter-chip routing ports, see Figure 1. A C64 chip has 80 processors, each with two thread units, a floating-point unit and two SRAM memory banks of 32KB each. A 32KB instruction cache, not shown in the figure, is shared among five processors. The C64 chip has no data cache. Instead a portion of each SRAM bank can be configured as scratchpad memory (SP). The remaining sections of SRAM together form the global memory (GM) that is uniformly addressable from all thread units. On-chip resources are connected to a 96-port crossbar network, which provides a 4GB/s bandwidth per port, in total 384GB/s on each direction. This huge bandwidth sustains all the intra-chip traffic communication and the six routing ports

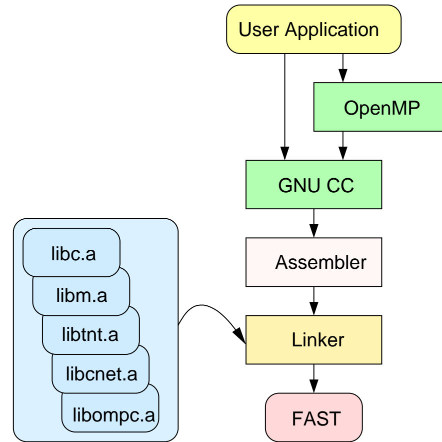


Figure 2: Cyclops-64 software toolchain

that connect each C64 chip to its nearest neighbors in the 3D-mesh network. The intra-chip network also facilitates access to special devices mapped under a generic host interface. These devices are a Gigabit Ethernet port, the control network and a serial ATA disk drive.

The C64 architecture represents a major departure from mainstream microprocessor design in several aspects. The C64 chip integrates processing logic, embedded memory and communication hardware in the same piece of silicon. However, it provides no resource virtualization mechanisms. For instance, execution is non-preemptive and there is no hardware virtual memory manager. The former means one single application can run at a given time on a set of C64 nodes and the C64 microkernel will not interrupt the user application unless an exception occurs. The latter means the three-level memory hierarchy of the C64 chip is visible to the programmer. From the processing core standpoint, a thread unit is a simple 64-bit, single issue, in-order RISC processor with a small instruction set architecture (60 instruction groups) operating at a moderate clock rate (500MHz). Nonetheless, it incorporates efficient support for thread level execution. For instance, a thread can stop executing instructions for a number of cycles or indefinitely; and when asleep it can be woken up by another thread through a hardware interrupt. As we have already seen, C64 is truly unique. In this paper we demonstrate how to take advantage of these architecture features to efficiently map OpenMP into this platform. In particular, we exploit relevant architecture features such as: (1) the capability to configure a section of every SRAM bank as scratchpad memory, which provides a fast temporary storage to exploit locality under software control; (2) a rich set of hardware supported in-memory atomic instructions. Unlike similar instructions on common off-the-shelf microprocessors, atomic instructions in the C64 only block the memory bank where they operate upon while the remaining banks continue servicing other requests. This functionality provides a higher memory bandwidth; (3) a 16-bit signal bus to which all threads within a chip are connected, that provides a means to efficiently implement barriers.

3. EXPERIMENTAL PLATFORM

Figure 2 illustrates the software toolchain available for application development on the C64 platform. This software

development environment has been in use by the architecture design team at IBM, system software developers and application scientists for the last years. More recently, we extended the toolchain with an OpenMP front-end, a porting from the Omni-1.6 OpenMP compiler [17]. The front-end serves as a translator that takes C and Fortran77 OpenMP programs as input and generates C programs with function calls to the Omni OpenMP runtime library. The resulting C programs are compiled with the Cyclops-64 cross-compiler and linked to the Omni runtime library. The toolchain also provides an assembler, linker and other binary utilities all based on binutils-2.11.2. The C standard and math libraries are derived from those in newlib-1.10.0. The experimental platform is completed by means of a functionally accurate simulator (FAST). FAST is an execution-driven, binary-compatible simulator of a multi-chip C64 system. It accurately reproduces the functional behavior and count of hardware components of a C64 system. In addition, it generates timing information that accounts for the main sources of pipeline delays and stalls such as contention in memory, the crossbar, and/or other functional units. Although not cycle accurate, this information has proven to be useful for performance estimation and application tuning as well [6].

4. BASE RUNTIME LIBRARY

For parallel execution, Omni relies on the POSIX thread library, which makes porting to other platforms easy. On C64 there is not a POSIX thread library. However, for the purpose of this work we extended the C64 native microkernel and multi-threaded runtime (TNT) with macros that provide the POSIX thread API. Hence, the OpenMP runtime library is built on top of TNT. A point worth noting is that instead of a Pthread-like high level abstraction model, TNT is a low level implementation of a thread virtual machine customized for the unique features of the C64 architecture [7]. Therefore, the OpenMP runtime library obtained from this straightforward porting is already efficient in the sense that it brings the runtime library closer to the underlying hardware, making use of the efficient thread management techniques provided by TNT, for instance.

The Omni OpenMP run time library described above, which is a direct porting of the original library to the C64 toolchain, is regarded as our base runtime library. In the following sections, first we describe how it can be further optimized. Then, we quantify the improvements due to these enhancements, by comparing the base and optimized runtime libraries by means of the EPCC microbenchmarks.

5. OPTIMIZED RUNTIME LIBRARY

In this section we present our mapping strategy that explores the opportunities to optimize the OpenMP runtime library for running programs on a C64 chip. During this process special care is taken to implement mechanisms that take into account the unique features of the C64 architecture, and hence introduce minimum overhead. Our approach is comprised of three steps:

1. A memory aware runtime library that takes advantage of the C64 explicit memory hierarchy by placing frequently used data structures in scratchpad memories that are closer to the processor/thread units.
2. A unique spin lock algorithm designed to make best use of the C64 architecture supported in-memory atomic instructions and thread sleep/wake-up mechanisms.
3. Use of the signal bus that provides a means for very fast communication of a small amount of information among thread units within a chip to implement a barrier synchronization.

5.1 Memory aware runtime library

The C64 chip has no data cache. Instead a portion of the embedded memory can be configured as scratchpad memory and accessed through a dedicated path. Such a memory provides a fast temporary storage to exploit locality under software control. Based on our previous work, we leverage the use of scratchpad memory to improve the OpenMP runtime library.

C64 toolchain provides segmented memory support. If the programmer precedes the declaration of a variable or function with a *pragma sram*, *pragma dram* or *pragma spm* the linker allocates the object or code section in global, off-chip or scratchpad (only data) memory, respectively. In addition, *pragma spm* makes a variable thread private, as the linker makes a local copy on each thread's scratchpad memory.

Using this mechanism we privatize the descriptors defined within the library to handle both the processors (hardware thread units) and threads (OpenMP threads). The benefit of this relocation is simple: faster access to the descriptor. Load from local, global and off-chip memory takes 2, 20 and 36 cycles, respectively. Therefore the improvement is clear and requires little explanation. Nonetheless, to help quantify the improvement we wrote a simple microbenchmark.

Figure 3 summarizes the (normalized) overhead due to several factors such as having the thread descriptor in global or local memory, reading one, two or three parameters from the descriptor, or even having an indirect access to the master thread, for instance. Reading from a thread descriptor, although a simple operation, may be repeated many times during the execution of a parallel loop, hence its relevance. Independently of the number of parameters read, the difference between having the descriptor in local or global memory is a single load latency. In the first case it is easy to hide a 2-cycle latency, no matter the number of memory references (LM1, LM2, LM3). For global memory, only the first load causes some delay. Successive memory instructions are issued before the pipeline stalls. The crossbar handles consecutive requests and after 20 cycles, starts delivering the results, one every cycle (GM1, GM2, GM3). Notice the overhead increases with the number of memory references. This has to do with the additional integer instructions required to prepare the memory loads, not with the memory delay itself. When there is an indirect access, the scenario changes significantly. With the descriptor in scratchpad an indirect reference is like two consecutive references (LMI). However, when the descriptor is in global memory the indirection stalls the pipeline before the second load is issued, hence the overhead effectively doubles (GMI). Furthermore, for a large number of active threads (64 and 128) microbenchmarks with the descriptor in global memory experience a performance degradation between 3-5% due to memory contention.

Besides faster access to its contents, the new location of the descriptor provides new opportunities for improvement: self-identification and less frequent access to the master thread descriptor.

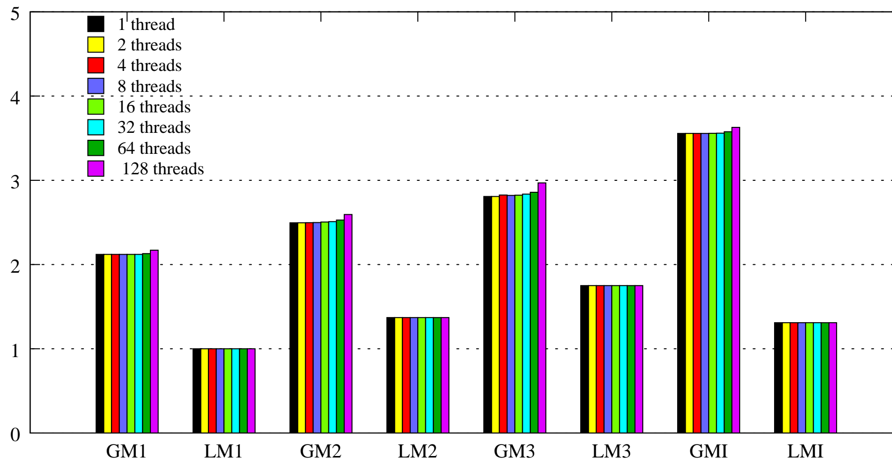


Figure 3: Normalized Overhead of Thread Descriptor Microbenchmarks. GM1: one load from a thread descriptor located in global memory; LM1: one load from a thread descriptor located in local memory; GM2: two loads from a thread descriptor located in global memory; LM2: two loads from a thread descriptor located in local memory; GM3: three loads from a thread descriptor located in global memory; LM3: three loads from a thread descriptor located in local memory; GMI: two loads from a thread descriptor located in global memory, the first being an indirect reference to the second; LMI: two loads from a thread descriptor located in local memory, the first being an indirect reference to the second.

In the original runtime library, self-identification requires a call to *pthread_self()*, which returns a unique identifier to the calling thread. This number is then used as input to a hash function that finally returns a pointer to the processor descriptor. Moving the descriptors to scratchpad memory simplifies this process. A reference to thread local data consists of the addition of a 16-bit offset, which is a constant known at compile time, to the higher 16 bits of a base register. The operation takes four cycles only!

Threads spawned during a parallel region form a team and share some information such as a function pointer, arguments to this function, etc. The parameters are defined by the OpenMP front-end based on the OpenMP directive and any additional clause, hence they are known to the master thread only. Threads within a team have to look for this information in the master descriptor. In the enhanced library, when the master thread polls workers to assign them new work, it copies this data into each worker's descriptor. This can be seen as the master thread prefetching data on behalf of the workers. A store instruction can be issued each cycle and the load-store data dependency is hidden by the instruction schedule (load from scratchpad). Thus, little overhead is added to the critical path. Once the workers start running in parallel they do not have to reach their parent thread. At that point, waiting for some data would have added considerable overhead to the critical path (load from global memory), which is what happens in the original library. Notice there would be no difference if the descriptor were in global memory, since the master thread would stall while prefetching from global memory.

Once the worker threads locally obtain many parameters, the number of references to the parent descriptor decreases significantly. Indeed, it is only to perform collective operations such as reduction, ordered execution, etc. that an access to the parent thread descriptor is required. At this point, it seems reasonable to split the definition of the worker and parent descriptors. The resulting size reduction translates into a more efficient use of the scratchpad memory, which is important given its limited size.

5.2 Mutual exclusion

On a shared memory system with 160 threads like C64, mechanisms for mutual exclusion are of vital importance. In the original OpenMP runtime and user libraries mutual exclusion is implemented by means of a two-level lock: mutex and spin lock, where the first one is provided by the POSIX thread library. Based on the Linux implementation, a mutex has two pieces of state: a "locked" bit and a queue. When the mutex is not locked, the queue is empty. Otherwise, the queue may contain identifiers representing threads waiting to acquire the lock. When the mutex is unlocked while the queue is not empty, the first queue entry is removed and the corresponding thread is implicitly given the lock's ownership. We use the test-and-set with exponential backoff for the spin lock algorithm, and while waiting on a queue, threads are put to sleep. However, the two-level lock results in the mutex and associated spin lock being allocated in DRAM, which is not efficient.

We propose to study existing spin lock and lock-free algorithms and improve them based on C64 special features. The objective of the study is to determine the most efficient algorithm and use it in the OpenMP runtime system and library. From our observations and contrary to common belief, we do not find any benefit from adopting a lock-free concurrent data structure, not to mention the complexity, maintainability, and debuggability of these. Instead, we choose our own adaptation of the MCS algorithm, we call MCS-SW, that uses little space in scratchpad memory and makes good use of the sleep and wake-up instructions to avoid spinning on any global or local shared flag.

5.2.1 Spin lock

We implement five spin lock algorithms on C64 as described below. Other well known algorithms are disregarded because we do not consider them appropriate. For example, array-based queuing locks [3, 9] are designed for multiprocessor systems with caches, which C64 does not have.

Test-and-set (TS) With TS lock, a processor repeatedly checks the lock to see if it is available and, if available, marks it as unavailable. A hardware *test-and-set* instruction is used to perform the check-and-mark-if-available actions atomically. In our implementation, the lock object is allocated in the on-chip global memory. We do not employ the well known *test-and-test-and-set* approach [28], because there is no data cache.

Test-and-set with exponential backoff (TS-exp) The same as TS lock, but instead of retrying immediately after a fail, an exponential increasing backoff is performed before the next try.

Ticket Before acquiring the lock, a processor increments a global counter to determine its position in a waiting list. All processors spin on a second global counter, which will be incremented when the lock is released. Both counters are allocated in on-chip global memory. Only the increment-and-get-ticket operation on the first global counter requires the use of a *fetch-and-inc* hardware atomic instruction, the spin uses a normal load instruction. A backoff mechanism is also used between two spins.

MCS An MCS lock [19] uses a distributed linked list to maintain the queue of waiting threads. Each thread spins on a separate node of the linked list. In our implementation, the node where a thread spins on is allocated in its own scratchpad memory. Therefore, there is no memory traffic generated to the crossbar network when a thread spins locally.

MCS with sleep/wake-up (MCS-SW) We modify the original MCS algorithm using the C64 sleep/wake-up support as introduced in Section 2. Instead of spinning, a thread waiting on a lock goes to sleep after it adds its node (allocated in its scratchpad memory) to the linked list. When a lock owner releases the lock, it wakes up its successor by sending a wake-up signal, which has the same cost of as a store instruction.

To evaluate the efficiency of different spin lock algorithms, we use two microbenchmarks proposed in [16]: *lock-delay*, and *lock-null*. In both benchmarks, each thread repeatedly performs 1,000 pairs of acquires and releases. The lock-delay microbenchmark uses fixed delays both inside and outside the critical section. The delay (D_i) inside the critical section is large enough (we use $3 \times D_0$; and D_o is the delay outside the critical section) such that the last thread that released the lock is already waiting to acquire the lock before the lock is released. For this microbenchmark, the overhead of a lock can be computed as follows:

$$\text{Overhead} = \begin{cases} \frac{\text{Execution Time}}{\text{No. Lock Acquires}} - D_i - D_o, & P = 1 \\ \frac{\text{Execution Time}}{\text{No. Lock Acquires}} - D_i & , P > 1 \end{cases}$$

$$\text{No. Lock Acquires} = \text{Iterations Per Thread} \times \text{No. Threads}$$

The second microbenchmark, lock-null, does not use any delay at all. Each thread continuously acquires and releases the lock 1,000 times. This is also the benchmark used in other studies [19]. Here,

$$\text{Overhead} = \frac{\text{Execution Time}}{\text{Iterations Per Thread} \times \text{No. Threads}}$$

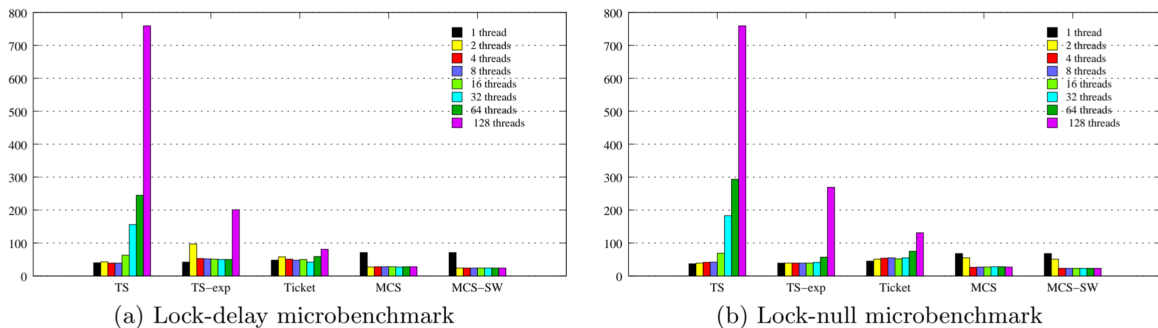
We measured the overhead and contention of our five spin lock algorithms using the lock-delay, and lock-null microbenchmarks. The overhead is calculated using the equations above. The amount of contention is reported by the simulator. When tow or more threads compete for the same resource at the same cycle, the simulator increments the contention counter by one. We normalize the number of contentions by the number of threads for the report in Figure 5. Low contention is important because it does not affect the overhead of the lock acquire/release only, but the normal execution of the user program as well.

Figures 4 and 5 demonstrate that MCS and MCS-SW always have lower overhead and contention than any other algorithm. In addition, MCS-based implementations show perfect scalability, as overhead and contention remains constant when the number of threads increases. It is worth noting that Ticket has lower overhead but higher contention than TS-exp lock. As we mentioned earlier, TS-exp always spins with the *test-and-set* instruction, which holds a memory module for three cycles. However, Ticket uses *fetch-and-increment* once to get its ticket and increment the counter. Then it spins using normal load instructions, which are served by memory in one cycle. As a result, with contention at the memory module TS-exp experiences a longer delay than Ticket.

MCS-SW shows several cycles lower overhead than MCS for more than one thread. And they both show the same level of contention. Additionally, MCS-SW executes less instructions per pair of lock/release than the original MCS lock. Actually with MCS-SW, each lock acquire/release operation takes a constant number of instructions, no matter how many threads contend for the lock. Unlike MCS, with which a thread keeps spinning on the local flag in scratchpad memory, with MCS-SW, a thread suspends after adding itself to the waiting queue. During the wait, the thread remains asleep and stops executing instructions until it is woken up by its predecessor. This is an important observation. Because when a thread is suspended, it consumes much less power. Therefore, MCS-SW is a time, memory, and power efficient spin lock algorithm for C64, and it is our algorithm of choice for the implementation of the OpenMP runtime library.

5.2.2 Lock-free

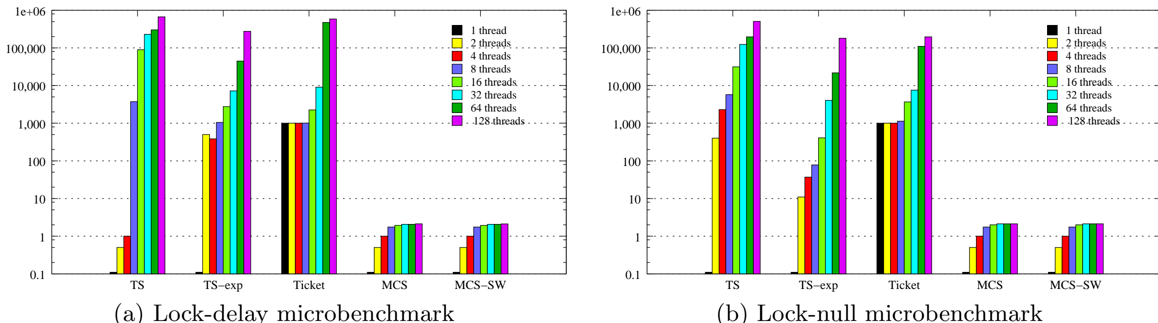
In the last decade, lock-free concurrent data structures and algorithms have emerged in literature. A lock-free concurrent data structure is "one that guarantees that if multiple threads concurrently access that data structure, then some thread will complete its operation in a finite number of steps, despite the delay or failure of other threads" [13]. It is argued that lock-free concurrent data structures do not



(a) Lock-delay microbenchmark

(b) Lock-null microbenchmark

Figure 4: Overhead of Spin Lock Algorithms [clock cycles]



(a) Lock-delay microbenchmark

(b) Lock-null microbenchmark

Figure 5: Normalized Contention of Spin Lock Algorithms

only avoid the inherent problem with locks, i.e. *priority inversion*, *convoying*, and *deadlock* [14], but also scale better and achieve higher performances than their lock-based counterparts. The only drawback seems to be difficulty and complexity of designing general lock-free concurrent data structures. For that reason, most of the work has focused on lock-free versions of commonly used basic data structures, such as stacks [15, 12], queues [29, 23, 10, 21], sets [18, 29, 10, 11, 20], etc. Since queues, stacks and hash tables are precisely the data structures implemented in the runtime library that are protected with locks to guarantee mutual exclusion, we compare lock-free with lock-based implementations on C64. Since there is no *priority inversion* and *convoying* problem in C64, performance and memory contention are the only factors we take into consideration.

For our study, we implement the lock-free version of the FIFO [23], LIFO [15], and hash table [20]. All these lock-free implementations adapt Michael's *Hazard Pointers* mechanism to guarantee safe memory reclamation of lock-free objects [22]. We implement the lock-based counterparts, which are straightforward, using our best spin lock algorithm: MCS-SW. We also implement a lock-free-backoff version for each data structure, which uses the same algorithms but an exponential increasing backoff is added before retry, when a fail is observed in the algorithm.

To evaluate the performance of these implementations we use microbenchmarks similar to those described in the spin lock study. For FIFO and LIFO, at each iteration, a thread performs either one enqueue/push or dequeue/pop operation randomly. A thread finishes after it completes 1,000 pairs of enqueue/push and dequeue/pop operations. After each operation, a small random delay is inserted before performing the next operation. The hash table has 25 buckets, and each bucket manages an ordered linked list. The hash table is initialized with a load factor of 10, which represents the average number of items per bucket. Each thread performs 10,000 operations, of which 20% are insertions, 20% deletions, and 60% searches. At each iteration, the operation to be performed is randomly determined, after which a small random delay is inserted. For the lock-based version, each bucket in the hash table is protected with a different lock to avoid unnecessary serialization. Finally, execution time and contention are normalized by the number of threads.

Figures 6(a) and 7(a) show that lock-based FIFO performs better and generates much less contention than both lock-free, and lock-free-backoff versions. Figure 7(b) shows that the lock-free-backoff LIFO generates slightly less contention than the lock-based version (although within the same range). However, the lock-based LIFO is much faster. Both the FIFO and LIFO implementations of the lock-free algorithm are worse than the lock-based version; especially regarding contention, which is increased by several magnitude orders. For the hash table, because there are 25 different buckets, conflicts do not happen frequently at runtime. All three implementations show the parallelism. In all instances, the lock-based version still performs faster and generates less contention than the other two.

Based on the above observations and contrary to common belief, we do not see any benefit from adopting lock-free concurrent data structures for the implementation of the OpenMP runtime library.

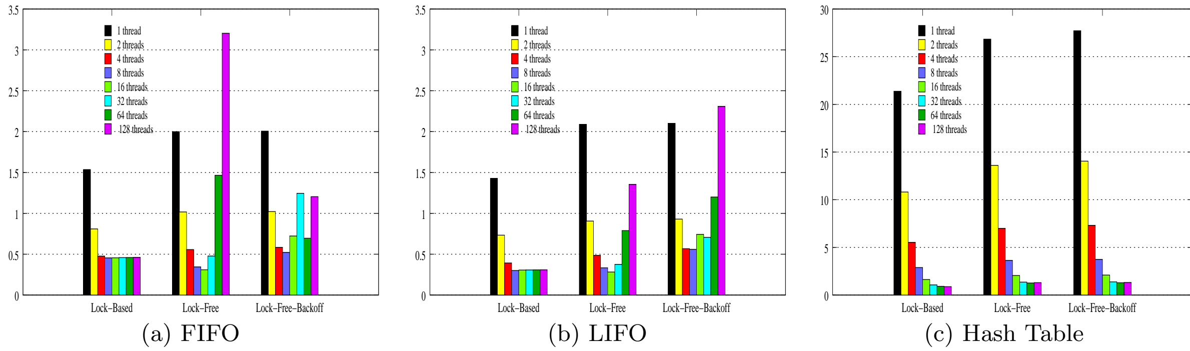


Figure 6: Normalized Execution Time of Lock and Lock-Free Algorithms [ms]

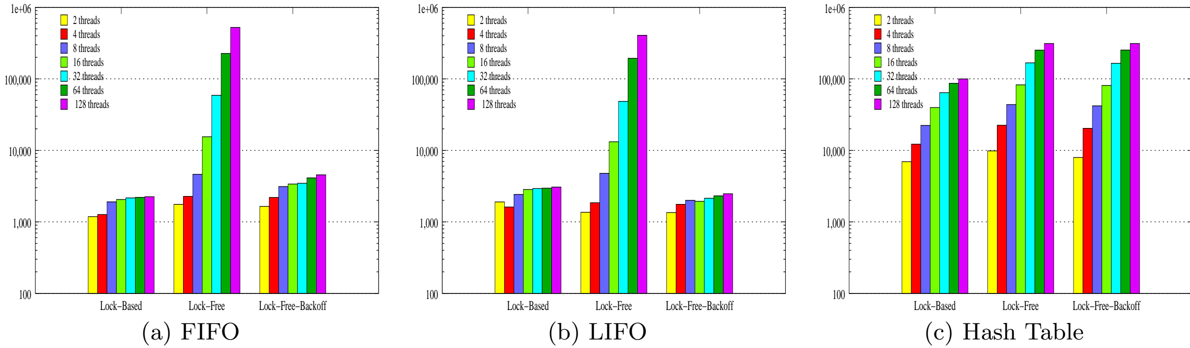


Figure 7: Normalized Contention of Lock and Lock-Free Algorithms

5.3 Barrier synchronization

Threads within a C64 chip are connected to a 16-bit signal bus that provides a means to efficiently implement barriers. A thread may write 16 bits onto the signal bus at any time. On reading, instead of the last value written, the logical OR of the last value written by all the thread units on the chip is returned.

Omni barrier function implements a 1-read/n-write busy-wait algorithm [19]. Obviously, the hardware mechanism for barrier synchronization available on C64 should outperform this or any other software implementation. Hence, the default barrier function has been replaced with calls to the TNT library that access the signal bus interface register. Besides significant improvements in execution time, the signal bus reduces memory traffic and power consumption, as spinning waiting for a signal bus line to drop does not interfere with other thread units or generate excessive heat.

6. RESULTS

In this section we demonstrate the efficiency of our newly redesigned OpenMP runtime library. The combination of the three optimizations described in the previous section: memory aware runtime library, time and memory efficient spin locks, and fast barrier synchronization, results in an 80% overhead reduction for all OpenMP language constructs. The improvement is relative to the Omni runtime library obtained from our earlier porting, which is regarded as the baseline. All the experiments are conducted with the C64 software tool set 1.5 release.

An important factor in determining the performance of a shared memory system is the overhead due to synchronization for language constructs in OpenMP. Furthermore, the costs of these operations are dependent on their implementation in the OpenMP runtime library. To find out what the cost for each construct is, we use the EPCC microbenchmarks [5], the main purpose of which is to measure the overhead of synchronization and scheduling in OpenMP. The overhead is basically determined by comparing the time taken for a section of code executed sequentially to the time taken for the same code executed in parallel enclosed in a given directive.

Figure 8 shows that OpenMP *parallel* and *parallel for* language constructs experiment an 80% overhead reduction for 128 threads.

Barrier synchronization improves by two orders of magnitude. This is not a surprise since the library migrated from a software implementation to a hardware-based synchronization mechanism. The *single* construct experiences a similar improvement. In part due to the new barrier synchronization, but also because of the other enhancements to the library. Enhancements such as self-identification, fast access to the thread descriptor, less frequent access to the parent thread descriptor, and efficient spin lock, not improvements to the barrier, are the only reason behind the drastic improvement in the *ordered* construct, see Figure 9.

The overhead for *critical*, *atomic* and *lock/unlock*, see Figure 10, are all basically the same, since the library functions, *ompc_lock/unlock* are used to implement the critical and atomic constructs. In all three cases there is an 80% re-

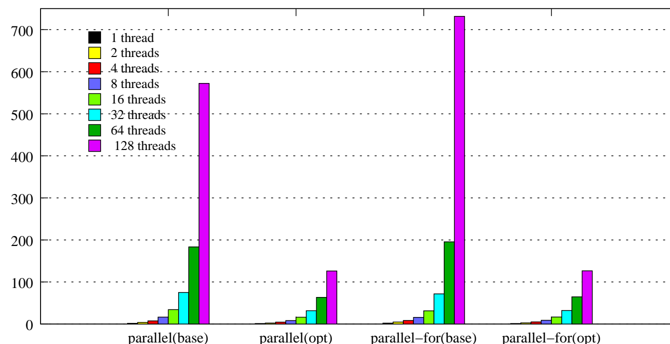


Figure 8: Overhead of OpenMP *parallel* and *parallel for* [μs]

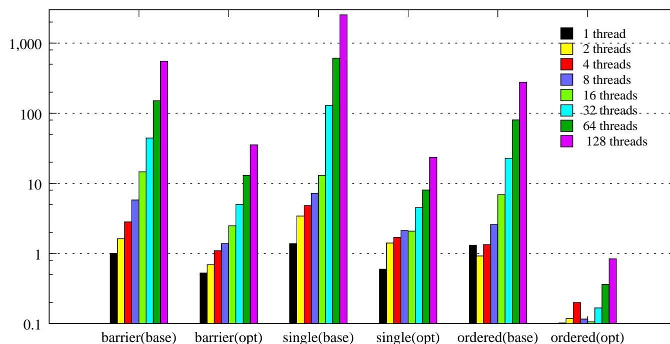


Figure 9: Overhead of OpenMP *barrier*, *single* and *ordered* [μs]

duction overhead that comes from using the MCS-SW spin lock algorithm directly instead of a Pthread mutex.

Finally, the *reduction* construct can be seen as a comprehensive example where all the optimizations previously described (self-identification, fast access to data in the descriptor, global barrier, etc.) add up. As a result, the optimized *reduction* drops to 15% of its original cost, see Figure 11.

7. FUTURE WORK

The scalability of the OpenMP programming interface on systems with a large number of processing elements is sometimes limited. In some instances the overhead due to the OpenMP runtime library accounts for a significant part of an application execution time. As a consequence, applications amenable to effective parallelization are often restricted to programs with a small number of parallel regions and/or regular data access patterns.

The goal of this paper was to optimize the Omni OpenMP runtime library, using the specific hardware features of the C64 architecture. In that sense, we believe that an 80% overhead reduction for language constructs in OpenMP should be regarded as an important milestone. As future work, we intend to enlarge the evaluation with some real applications and/or benchmarks. However, we are not interested in regular memory-intensive applications. We believe these programs will not be able to take advantage of the C64 architecture, since the off-chip memory (DRAM) bandwidth is quite limited. Instead, we are looking into irregular applications (with numerous parallel regions, and/or continuous synchronization) that usually are disregarded for not being suited to effective parallelization with OpenMP. For this type of code, the interaction between the runtime system and the application is significant, hence the results hereby provided become more relevant.

8. RELATED WORK

To the best of our knowledge, most of the previous work on performance characterization of OpenMP focuses on benchmarking general purpose shared memory systems and commercial compilers. Since the computing environment is given, the authors are limited to report their findings with little chance to make significant improvement if any at all. On the other hand, we target a specific machine and we optimize the library by efficiently mapping software constructs to the available hardware resources.

Initially, Bull [5] proposed a set of microbenchmarks that offer some insight on the implementation of an OpenMP runtime library for comparison and optimization purposes. He made use of the EPCC microbenchmarks to compare three different platforms: Sun HPC 35000, SGI Origin 200 and Compaq Alpha Server, and suggested that the differences observed would open the door to further improvements. Using a similar procedure Berrendorf [4] investigated the level of support for OpenMP in four commercial compilers. Prabhakar [26] proposed an enhanced set of microbenchmarks derived from EPCC and demonstrated its usefulness with a comparative study of OpenMP language constructs on the IBM SP3 and SUN SunFire systems. More recently,

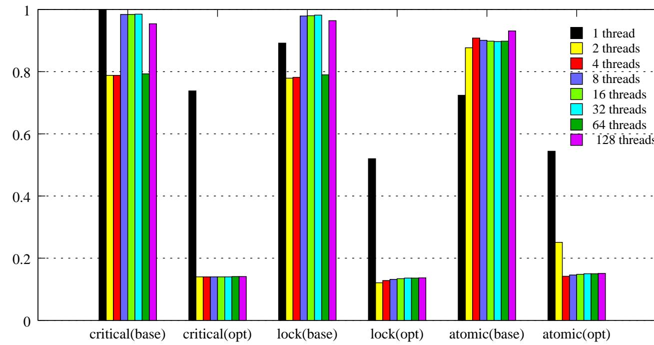


Figure 10: Overhead of OpenMP *critical*, *lock/unlock* and *atomic* [μs]

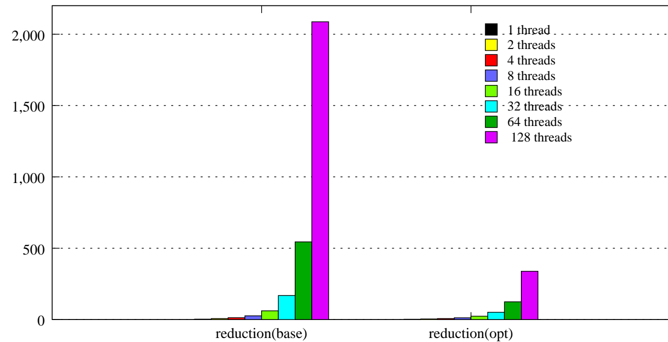


Figure 11: Overhead of OpenMP *reduction* [μs]

Fredrickson [8] presented an exhaustive report after running the EPCC microbenchmarks, NAS parallel benchmarks and the SPECOMPL2001 benchmark suite on a 72-processor Sun Fire 15K SMP server. However, the author did not offer any insight as to why specific applications spent a significant fraction of the total execution time due to overhead and how to reduce it. In the first evaluation of the Omni OpenMP compiler, the authors discussed implementation details and their initial attempt to optimize the library on conventional SMP systems [17]. Probably the closest related work is that done on an experimental Cyclops architecture (never to be built) that (compared to C64) included data caches, a larger amount of on-chip memory and no off-chip memory, among other differences. In [1] the authors ported the Nanos thread library (based on Quick Threads) to this special Cyclops platform. However no optimization to the runtime library was described, even though the Nanos library requires excessive memory to represent parallelism, which to our understanding, makes it inefficient on an environment with such a large number of threads and limited memory like the Cyclops platform. In a subsequent work [27], two memory optimizations were presented. First, they remapped the data caches, i.e. changed the order of some hardware address lines. Second, they padded the thread descriptors to distribute memory access across all the data caches. Memory efficiency issues within the library were not addressed in either case. Furthermore, both solutions focus on improving data cache locality, hence they are specific to the Cyclops architecture used in that study.

9. SUMMARY

We have reported our experience mapping OpenMP into the Cyclops-64 architecture. Although porting the Omni OpenMP compiler to C64 using the TiNy Threads™ runtime system makes the resulting runtime library already efficient, we demonstrate how further improvements can be achieved with the mindful usage of relevant architecture features. In particular, we show an 80% overhead reduction for all OpenMP language constructs. This represents a substantial decrease in the cost of managing parallelism, which makes OpenMP a parallel programming model amenable for writing parallel programs on the C64 platform.

Acknowledgments

We acknowledge the support from the Department of Defense, the Department of Energy (DE-FC02-01ER25503), the National Science Foundation (CNS-0509332), IBM, ETI and other government sponsors. The authors would like to acknowledge all the people at CAPSL and ETI who have been involved in the C64 project, in particular Brice Dobry, Geoffrey Gerfin, John Tully, Wesley Toland and Ziang Hu.

10. REFERENCES

[1] George S. Almási, Eduard Ayguadé, Călin Caşcaval, José Castaños, Jesús Labarta, Francisco Martínez, Xavier Martorell, and José Moreira. Evaluation of OpenMP for the Cyclops multithreaded architecture. In *OpenMP Shared Memory Parallel Programming: International Workshop on OpenMP Applications and Tools, WOMPAT 2003*, volume 2716 of *Lecture Notes*

in Computer Science, pages 69–83, Toronto, Canada, June 26–27, 2003.

[2] George S. Almási, Călin Caşcaval, José G. Castaños, Monty Denneau, Wilm Donath, Maria Eleftheriou, Mark Giampapa, Howard Ho, Derek Lieber, José E. Moreira, Dennis Newns, Marc Snir, and Henry S. Warren, Jr. Demonstrating the scalability of a molecular dynamics application on a petaflops computer. *International Journal of Parallel Programming*, 30(4):317–351, August 2002.

[3] Thomas E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[4] Rudolf Berrendorf and Guido Nieken. Performance characteristics for OpenMP constructs on different parallel computer architectures. *Concurrency - Practice and Experience*, 12(12):1261–1273, 2000.

[5] J. Mark Bull. Measuring synchronization and scheduling overheads in OpenMP. In *Proceedings of the First European Workshop on OpenMP*, Lund, Sweden, September 30 - October 1, 1999.

[6] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. FAST: A functionally accurate simulation toolset for the Cyclops64 cellular architecture. In *Proceedings of the Workshop on Modeling, Benchmarking and Simulation*, pages 11–20, Madison, Wisconsin, June 4, 2005. Held in conjunction with the 32nd Annual International Symposium on Computer Architecture.

[7] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Toward a software infrastructure for the Cyclops-64 cellular architecture. In *Proceedings of the 20th International Symposium on High Performance Computing Systems and Applications*, St. John's, Newfoundland and Labrador, Canada, May 14–17, 2006.

[8] Nathan R. Fredrickson, Ahmad Afsahi, and Ying Qian. Performance characteristics of OpenMP constructs, and application benchmarks on a large symmetric multiprocessor. In *Proceedings of the 2003 International Conference on Supercomputing*, pages 140–149, New York, June 23–26 2003.

[9] Gary Graunke and Shreekant Thakkar. Synchronization algorithms for shared-memory multiprocessors. *Computer*, 23:60–69, June 1990.

[10] Michael B. Greenwald. *Non-blocking synchronization and system design*. PhD thesis, Stanford University, 1999.

[11] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, number 2180 in Lecture Notes in Computer Science, pages 300–314, Lisbon, Portugal, October 3–5, 2001.

[12] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the 16th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 206–215, Barcelona, Spain, June 27–30, 2004.

[13] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems*, 23(2):146–196, May 2005.

[14] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, San Diego, California, May 17–19, 1993.

[15] IBM system/370 extended architecture, Principle of operation, publication no. SA22-7085, 1983.

[16] Sanjeev Kumar, Dongming Jiang, Rohit Chandra, and Jaswinder Pal Singh. Evaluating synchronization on shared address space multiprocessors: Methodology and performance. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):23–34, June 1999.

[17] Kazuhiro Kusano, Shigehisa Satoh, and Mitsuhisa Sato. Performance evaluation of the Omni OpenMP compiler. In *Proceedings of the 3rd International Symposium on High Performance Computing*, volume 1940 of *Lecture Notes in Computer Science*, pages 403–414, Tokyo, Japan, October 16–18, 2000.

[18] Vladimir Lanin and Dennis Shasha. Concurrent set manipulation without locking. In *the 7th ACM Symposium on Principles of Database Systems*, pages 211–220, March 1988.

[19] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[20] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, August 2002.

[21] Maged M. Michael. CAS-based lock-free algorithm for shared deques. In *the 9th Euro-Par Conference on Parallel Processing*, pages 651–660, August 2003.

[22] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst*, 15(6):491–504, 2004.

[23] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, New York, USA, May 1996.

[24] OpenMP Architecture Review Board. OpenMP FORTRAN application program interface. Technical Report 2.0, November 2000.

[25] OpenMP Architecture Review Board. OpenMP C and C++ application program interface. Technical Report 2.0, March 2002.

[26] Achal Prabhakar, Vladimir Getov, and Barbara Chapman. Performance comparisons of basic OpenMP constructs. In *Proceedings of the 4th International Symposium on High Performance Computing*, number 2327 in Lecture Notes in Computer Science, pages 413–424, Kansai Science City, Japan, May 15–17, 2002.

[27] David Ródenas, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, George Almási, Călin Caşcaval, José Castaños, and José Moreira. Optimizing NANOS OpenMP for the IBM Cyclops multithreaded architecture. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium*, page 110, Denver, Colorado, April 4–8, 2005.

[28] Larry Rudolph and Zary Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 340–347, Ann Arbor, Michigan, June 5–7, 1984.

[29] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th Annual ACM Symposium of Distributed Computing*, pages 214–222, Ottawa, Ontario, Canada, August 2–23, 1995.