Improving Wait-Free Algorithms for Interprocess Communication in Embedded Real-Time Systems [*]

Hai Huang, Padmanabhan Pillai, and Kang G. Shin
Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122
{haih,pillai,kgshin}@eecs.umich.edu

Abstract

Concurrency management is a basic requirement for interprocess communication in any multitasking system. This usually takes the form of lock-based or other blocking algorithms. In real-time and/or time-sensitive systems, the less-predictable timing behavior of lock-based mechanisms and the additional task-execution dependency make synchronization undesirable. Recent research has provided non-blocking and wait-free algorithms for interprocess communication, particularly in the domain of single-writer, multiple-reader semantics, but these algorithms typically incur high costs in terms of computation or space complexity, or both. In this paper, we propose a general transformation mechanism that takes advantage of temporal characteristics of the system to reduce both time and space overheads of current single-writer, multiple-reader algorithms. We show a 17–66% execution time reduction along with a 14–70% memory space reduction when three wait-free algorithms are improved by applying our transformation. We present three new algorithms for wait-free, single-writer, multiple-reader communication along with detailed performance evaluation of nine algorithms under various experimental conditions.

1 Introduction

A key benefit provided by operating systems is a task or thread abstraction to manage the complexity that rapidly evolves even in very small embedded systems. A task/thread model mitigates the complexity growth of large monolithic programs, and simplifies the sharing of computing resources between the disparate functions of the system. However, the tasks of a system very rarely work independently of each other, hence needing interprocess communication (IPC) between tasks.

The simplest method of IPC is through global, shared variables. This is a very low-overhead method of communication, but has obvious flaws in concurrent accesses by multiple tasks. Even if we restrict the domain to single-writer semantics, which is common in embedded systems and sensor networks, data corruption can occur.

To avoid reading corrupted data from a concurrent object, critical sections are often used to coordinate accesses from different tasks. The simplest approach to implementing critical sections is to disallow task preemption inside of the critical section. This can be done by disabling and enabling interrupts in the CPU at the beginning and end of the critical sections, respectively. These are privileged operations and require kernel intervention. The read and write operations must be implemented in the kernel, or the application must be wholly trusted, since any task running with interrupts disabled cannot be preempted and may, either maliciously or inadvertently, disrupt the system. Moreover, disabling interrupts does not suffice to manage concurrency in multiprocessor systems.

The most common way to implement critical sections is to use software locks — typically through mutexes and semaphores. A task has to acquire the necessary locks before it can access shared objects. If the needed lock is already held by another task, the task blocks, and the operating system will resume it when the resource becomes available. Using locks serializes concurrent tasks that try to access the shared objects simultaneously, thus preventing corruption. In a multiprocessor environment, this reduces parallelism and decreases the utilization of available resources.

Locks can also cause more serious problems such as unpredictable blocking times and deadlocks. If a task is blocked while still holding the lock (e.g., a page fault occurred, or it is preempted by a higher-priority task), any other tasks waiting for the lock are unable to make progress until the lock is subsequently released. In the worst case, the task may fail while holding the lock, or block indefinitely due to circular lock dependencies, causing deadlock and blocking other tasks from ever making progress.

Even with safeguards to avoid deadlock, locks are particu-

[*]The work reported in this paper is supported in part by the U.S. Airforce Office of Scientific Research under Grant No. F49620-01-1-0120, and by DARPA administered under AFRL contract F30602-01-02-0527.

larly unattractive in real-time and embedded systems. Due to blocking and switching to other tasks, using locks can incur high and unpredictable execution time overheads, and cause many other problems, including priority inversion, convoying of tasks, more difficult schedulability analysis, and increased susceptibility to faults. In real-time systems, tasks are usually assigned fixed or deadline-based priorities, according to which they are scheduled. Priority inversion can occur when a high-priority task is blocked waiting for a lock, but the lock holder does not make progress due to its low priority. This is such a serious issue that many algorithms have been developed to limit the effects of priority inversion, including the priority inheritance protocol, the priority ceiling protocol, and the immediate priority ceiling protocol [3, 28, 29]. Furthermore, providing real-time execution guarantees becomes more difficult. The simple, classical real-time analysis techniques [21] assume independently-executing tasks, which is clearly violated when locks are used. More complex analysis [29] may be used to provide real-time guarantees by accounting for worst-case blocking times, but this may result in poorer utilization of system resources.

Due to the above problems associated with lock-based synchronization IPC approaches, several algorithms that perform non-blocking and wait-free[1] communication with single-writer, multiple-reader semantics have been proposed. These allow tasks to independently access the shared message area without locks and the problems introduced by blocking. These algorithms, however, are not perfect. Although blocking is avoided, the operations may become quite complex and can incur non-negligible computational overheads. More importantly, the algorithms all use multiple buffers to avoid corruption, so their space overhead is high, wasting memory resources that are severely limited in small, embedded systems.

In this paper, we present three new wait-free algorithms. We develop a generalized transformation mechanism that can improve existing wait-free algorithms by exploiting the temporal characteristics of communicating tasks, significantly reducing both space and execution time overheads. For some existing algorithms, we show up to 66% reduction in execution time and 70% reduction in memory requirements after applying our transformation. The transformed algorithms preserve all of the benefits of wait-free communication along with significant time and space savings.

In the following section, we present some background information and further motivate this work. We present our transformation mechanism in Section 3, and illustrate it using some actual IPC algorithms. Detailed evaluations are done in Section 4. We will put our work in the perspective of related work in Section 5, before concluding in Section 6.

[1]A concurrent object implementation is non-blocking if at least one process that is accessing the object can complete an operation within a finite number of steps regardless of failures. Furthermore, it is wait-free if every process that is accessing the object can complete an operation within a finite number of steps [13]. Wait-free is a stronger form of non-blocking as it ensures starvation-free access.

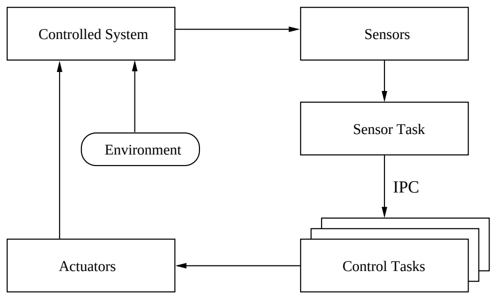


Figure 1: A schematic block diagram of a real-time system.

2 Motivation

In this paper, we are primarily concerned with communication between a single writer and multiple readers. This is a very common scenario in embedded systems — ranging from as complex as automotive and industrial control systems to as simple as the controllers in kitchen appliances. Figure 1 shows a typical real-time system. The sensors are used to acquire information from the controlled system. A sensor task reads the data, performs any preprocessing, and distributes the information to the various control tasks. The control tasks perform computations and set the actuators based on this information, so it is important that they obtain uncorrupted, most-recently produced data from the sensor task.

Traditionally, the writer (i.e., sensor task) must pass the data to the readers (i.e., control tasks) by means of mailboxes, one of which is associated with each reader. However, if there is a large disparity in the execution frequencies of the tasks, especially if the sensor read rate is higher than the actuator control output rates, as is common, data messages will queue up in the mailboxes. The reader will obtain outdated messages, and will either have to process these or discard them to acquire the most current information. Generating multiple copies of each message incurs overheads in processor cycles and memory space, both of which are scarce resources in an embedded system. Therefore, the mailbox approach is neither appropriate nor efficient for typical IPC needed in real-time and embedded systems.

State messages are used to alleviate such problems. They were proposed in the MARS project [16] and implemented in ERCOS [25]. The state messages approach associates mailboxes with the writer instead of the readers, so only the writer associated with a particular mailbox can write to it. Furthermore, each message is assumed to include all data that needs to be communicated, so that the single, most current message conveys all information. Since data are time-sensitive, a new message can simply overwrite the previous one, effectively presenting the readers with the most up-to-date information. However, since the writer and readers can access the writer's mailbox concurrently, the readers can potentially read corrupted data if

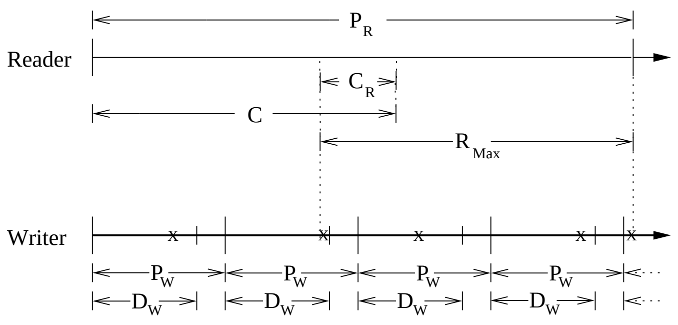


Figure 2: Reader and writer execution timelines, and each $\times$ denotes a write operation performed by the writer.

the writer simultaneously writes new data.

There are many synchronization-based algorithms [9, 10] designed to ensure that reader tasks will always access uncorrupted messages. As mentioned earlier, synchronization, particularly with locks, can cause many problems of its own. Therefore, in this paper, we focus on wait-free, single-writer, multiple-reader IPC algorithms [7, 8, 17, 24, 31]. However, these algorithms have higher space overheads than the synchronization-based algorithms. Even though the worst-case time overhead of these algorithms is significantly lower than that of the synchronization-based ones, the execution overheads can still be significant. Later in this paper, we present a transformation mechanism that takes advantage of the real-time properties of the communicating tasks to reduce both the time and space overheads of this class of algorithms. First, however, we present a brief overview of real-time systems and tasks in the next section.

2.1 Attributes of Real-Time Tasks

Tasks in a typical real-time system are periodically invoked/released and executed.[2] Each task T is associated with various attributes, including its period P, relative deadline D,[3] and worst-case execution time (WCET) C. The task must be run once each period, and needs to receive enough processing time to complete execution by its relative deadline. The real-time scheduler uses these attributes to decide when to run tasks, and can guarantee that all tasks will meet their deadlines as long as they require no more than their specified WCETs. From high-level program flow analysis and low-level timing information, a task's WCET can be determined statically. Figure 2 shows the relationship between these values for a typical scenario with one reader and one writer processes. The top timeline represents the reader's period. For simplicity, the reader's relative deadline is assumed to be equal to its period in our discussion and not shown here. In general, it is less than or equal to P_R, where P_R is the reader's period. C denotes the reader's WCET, and C_R is the time to perform a read operation. R_{Max} represents the maximum time the reader can take to perform a read operation. Note in Figure 2 that the read operation is placed at the end of the reader task's execution. It is only drawn there to show the relationship between R_{Max}, C_R, P_R and C more clearly, but, in general, the read operation can be anywhere within the reader's execution time C. The bottom timeline represents 4 writer periods. The writer's period and relative deadline are denoted by P_W and D_W, respectively.

[2]Aperiodic tasks can be handled by a periodic server [18], so the periodic task model is not a limiting assumption.

[3]This equals the deadline minus the release time of the task.

2.2 Temporal Concurrency Control

Since R_{Max} includes the time the reader is preempted by higher-priority tasks, it determines the maximum time the writer process may interfere with the reader within the reader's period without the reader missing its deadline. R_{Max} is calculated as follows:

$$R_{Max} = P_R - (C - C_R).$$

Assuming that all deadlines are met, Figure 2 illustrates the worst-case scenario in terms of the maximum number of preemptions of the reader by the writer task. This occurs when the first interfering-write happens as late as possible within the writer's period (first vertical dotted line — just before the writer's deadline) and the last interfering-write happens as early as possible within the writer's period (second vertical dotted line — just after the writer is released).

Let N_{Max} denote the maximum number of times the writer might interfere with the reader process during a read operation. N_{Max} can be calculated as:

$$N_{Max} = max\left(2,\ \left\lceil \frac{R_{Max} - (P_W - D_W)}{P_W} \right\rceil + 1\right).$$

Therefore, if we use an $(N_{Max} + 1)$-deep circular buffer instead of a single message buffer, the writer can post messages cyclically without ever interfering with the reader process, assuming that the real-time constraints are met. This allows the reader and writer to access the message area independently of each other without blocking, using only temporal characteristics guaranteed by the real-time scheduling and a sufficiently-deep circular buffer to manage concurrency. With multiple readers, we simply choose an N_{Max} value large enough to work for all readers, i.e., compute it using the task with largest R_{Max}. Finally, we keep a pointer to the most recently written message. This is updated by the writer, and subsequently used by the readers to retrieve the latest message. This concept was first introduced in [16] and later implemented in the Non-Blocking Write (NBW) protocol [17].

This algorithm is very efficient in terms of execution time, i.e., almost as fast as using global variables with no protection. The only overhead associated with this algorithm is the cost of maintaining the pointer for the most recently written message. Therefore, it is easy to see that it has optimal timing behavior among wait-free algorithms.

2.3 Restricting Memory Use

With a deep enough buffer, the above algorithm will always guarantee that the readers will not acquire corrupted data. However, when R_{Max} is large or P_W is small, N_{Max} can get quite large and would require a large buffer space. This is undesirable, especially in embedded systems where memory is usually a scarce resource.

EMERALDS's state message algorithm [35] improves upon the NBW protocol. To limit memory usage, EMERALDS simply sets a static maximum buffer threshold for the state message. The reader tasks are divided into two groups, *fast* and *slow* readers. Tasks that have N_{Max} values less than this maximum buffer threshold are classified as fast readers, while the others are classified as slow readers.

The fast readers execute according to the NBW protocol. Since these readers have small N_{Max} values, they are both time- and space- efficient. For slow readers, EMERALDS provides a system call mechanism that (i) disables interrupts, (ii) copies the message from the shared buffer to the slow reader's local space on behalf of the reader, and (iii) re-enables interrupts. The overhead of this system call is quite high; however, according to the definition of slow readers, this call is invoked relatively infrequently, so it was claimed not to greatly impact the overall average-case execution time overheads.

As we will see in Section 4, the amount of overhead due to this system call is significant enough to make its average-case execution time much higher than the non-blocking algorithms. We would like to have the low execution overheads of the NBW protocol and the low memory usage achieved by the EMERALDS implementation, but without resorting to locks, disabled interrupts, or other synchronization-based concurrency control mechanisms. The following section details how to achieve this by transforming existing wait-free IPC mechanisms.

3 Improving Wait-Free IPC

In order to gain the benefits of wait-free IPC along with low memory usage, and low average- and worst- case execution times, we first generalize the concept of fast and slow readers (to reduce the memory requirements) introduced in EMERALDS. We then devise a transformation mechanism that can be applied to existing wait-free algorithms, preserves all of their inherent benefits, and simultaneously improves their performance.

Here, fast readers are defined as those tasks for which temporal concurrency control suffices to ensure uncorrupted reads without excessive memory usage. Slow readers consist of all of the other reader tasks, which would require too much memory to employ temporal concurrency control alone. The actual division of tasks would depend on the requirements of the final system, as we will see later.

We can transform IPC algorithms to use this concept of fast and slow readers. The fast readers will basically employ the NBW read mechanism, and will require sufficient buffers to ensure temporal concurrency control. The slow readers will use the existing IPC mechanism, although slight changes may be required because of the parallel approach employed by the fast readers. The writer requires more significant changes in order to interact with both types of readers. The precise nature of these changes depends on the actual algorithm transformed.

In general, we can make some predictions about the resulting performance. First, the average-case execution time (ACET) will decrease, since the highest-frequency readers will use the very efficient NBW mechanism. Worst-case execution time (WCET) is also often reduced, since for most algorithms, execution time depends on the number of simultaneous readers using the mechanism, which is reduced to only the slow readers. With the proper division of tasks into fast and slow readers (Section 3.4), the transformed algorithm should require much less memory on average than the original algorithm, and in the worst case, require no more than the original.

Our transformation mechanism can be illustrated more concretely by showing how we apply it to some actual algorithms. We first apply our transformation to the algorithm proposed by Chen *et al.* in [7]. We then show how to transform the Double Buffer algorithm, which we have developed and present in Section 3.2. Chen's algorithm has a relatively high execution time overhead and low space overhead, so we expect our transformation to primarily improve execution time. In contrast, the Double Buffer algorithm has a high space overhead and low execution time overhead. We expect this algorithm to benefit primarily from memory usage reduction after transformation. The following subsections detail the improved algorithms, which are evaluated in Section 4.

3.1 Improving Chen's Algorithm

Chen *et al.* [7] proposed a single-writer, multiple-reader wait-free algorithm using the Compare-And-Swap (CAS) instruction. This instruction is used to atomically modify the states of control variables used to ensure that the writer never writes to a buffer currently in use by some readers. The CAS instruction is commonly used in non-blocking algorithms to coordinate accesses to shared buffers and is supported on most modern microprocessors. Even if an architecture does not support this instruction, it can be synthesized by using other system primitives or system support [5]. The instruction CAS(A,B,C) is defined to be equivalent to atomically executing "if A equals B, then set A to C and return true, else return false."

Chen's algorithm requires $(P + 2)$ message buffers, where P is the number of reader tasks. There is a global variable, `Latest`, that indexes to the most recently written message buffer. Additionally, each reader has an entry in a usage array indicating the buffer it is using. When the reader reads, it

```
int NSReader;              # Number of slow readers
int NBuffer;               # Number of buffers
int Latest;                # Index to the latest message
message Buff[NBuffer];     # Message buffer
char Reading[NSReader];    # Usage count

     SlowReader_i() {
1:       Reading[i] = NBuffer;
2:       ridx = Latest;
3:       CAS( Reading[i], NBuffer, ridx );
4:       ridx = Reading[i];
5:       read Buff[ridx];
     }
     int GetBuff() {
         InUse[NBuffer];
6:       for (i = 0; i < NBuffer; i++) InUse[i] = false;
7:       InUse[Latest] = true;
8:       for (i = 0; i < NSReader; i++) {
9:           j = Reading[i];
10:          if (j ≠ NBuffer) InUse[j] = true;
11:      }
12:      for (i = ((Latest + 1) mod NBuffer); ;
13:           i = ((i + 1) mod NBuffer)) {
14:          if (InUse[i] == false)
15:              return i;
         }
     }
     Writer() {
16:      widx = GetBuff();
17:      write Buff[widx];
18:      Latest = widx;
19:      for (i = 0; i < NSReader; i++)
20:          CAS(Reading[i], NBuffer, widx);
     }
```

Figure 3: Improved Chen's Algorithm.

first clears its entry, and then uses CAS to atomically set this to `Latest` if it is still cleared. It then reads back the value from its entry, and can then safely read from the indicated buffer. The writer has slightly more work to do. It first scans the usage array and selects a free buffer. It performs the write, updates `Latest`, and then must scan and set each reader entry that is cleared to `Latest` using CAS. This has been proven to ensure correct non-blocking IPC behavior in [7].

By taking into account the real-time properties of the communicating tasks, we can divide the reader set into two sets: fast and slow reader sets. By separating the reader set, we can reduce the space requirement from $P + 2$ to $M + max(2, N)$, where M is the *number* of slow readers and N is the *number of buffers* needed by the fast readers. Section 3.4 describes how to compute M and N in order to optimize for space. Because N is chosen to be less than, or equal to, the number of fast readers (i.e., $N \leq P - M$), the improved algorithm requires no more buffer space than the original algorithm. In the worst case (i.e., all readers are slow readers), the improved algorithm simply degenerates to the original algorithm. Furthermore, the execution time overheads will be greatly reduced, since fast readers use the very efficient NBW mechanism and the writer overhead is linear to the number of slow readers only, rather than all readers. Therefore, both space and time overheads can be reduced.

The Improved Chen's algorithm is shown in Figure 3. `NSReader` is the number of slow readers. `NBuffer` is the total number of message buffers. `Buff[]` is the array of message buffers shared between the writer and readers. `Latest` is a control variable that indexes this array, indicating the most recently written message buffer. `Reading[]` is the usage array associated with the slow readers such that `Reading[`i`]` indicates which buffer entry the i^{th} slow reader is currently reading.

The slow readers operate identically to the readers in Chen's algorithm. Just before the i^{th} slow reader reads from the message buffer, `Reading[`i`]` is set to a value between 0 and `NBuffer-1` to indicate the index of the buffer it will be reading. The writer will not overwrite this buffer slot as long as the slow reader is still using it. The slow reader first assigns `Reading[`i`]=NBuffer` to indicate that it is preparing to make a read operation. Then, it reads `Latest`, and attempts to set `Reading[`i`]` to this value atomically using CAS. If the writer has preempted the reader and completed a buffer write before this instruction, it would have already set `Reading[`i`]` to the new `Latest` value, and the reader's CAS would fail. In any case, by line 3, `Reading[`i`]` would have been atomically set to a buffer index that the writer will not use. So the slow reader simply reads the index and can now read from the indicated buffer safely.

The fast reader (not shown) is the same as in the NBW protocol. It relies only on temporal concurrency control, so it just reads `Latest` and uses the indicated buffer.

The `Writer()` process looks just like the one in Chen's algorithm. It calls `GetBuff()` to determine which buffer slot is safe to use next. After it writes the next message, it updates `Latest` and then modifies each `Reading[`i`]` using CAS if necessary.

The key difference lies in `GetBuff()` function, which is modified to allow temporal concurrency control for fast readers. First, to prevent the writer from interfering with slow readers, `GetBuff()` picks a buffer, m, such that no slow reader is using it (i.e., for all i, `Reading[`i`]`$\neq m$). To protect the fast readers, as with the NBW protocol, we must ensure that there are at least $(N - 1)$ writes between two consecutive writes to any particular buffer, where N is the buffer depth required for temporal concurrency control (Section 2.2). `GetBuff()` prevents the writer from interfering with the fast readers by cyclically choosing buffer entries starting from `Latest`. When `NBuffer` is chosen correctly (Section 3.4), even if each slow reader is using a unique buffer, there will be enough buffers (i.e., `NBuffer` $-$ `NSReader`) left so that the cyclic selection will ensure sufficient time between two consecutive writes to the same buffer, satisfying the requirements for temporal concurrency control. Thus, the writer will not interfere with either fast or slow readers.

Let us illustrate this using the example shown in Figure 4. Assume that there are 20 readers, of which 3 are identified as slow readers. Assume further that relative execution frequencies of the fast readers and the writer are such that they require a 4-deep buffer to ensure temporal concurrency control. In this

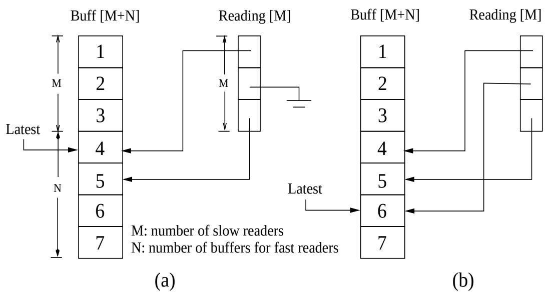

Buff [M+N]
Reading [M]
Buff [M+N]
Reading [M]
Latest
M
N
M: number of slow readers
N: number of buffers for fast readers
Latest
(a)
(b)

Figure 4: An example for Improved Chen's algorithm.

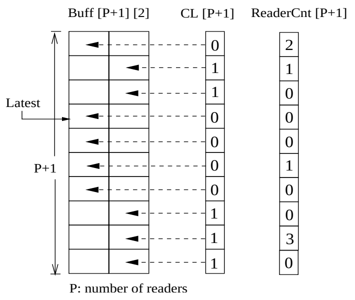


Figure 5: Constructs in the Double Buffer algorithm.

system, therefore, we need 7 message buffers (4 for the fast readers, and 1 for each of the slow readers), as compared to 22 buffers needed with the original Chen's algorithm. Figure 4(a) shows a particular execution state of the task set with `Latest` points to the 4^{th} buffer slot. Since `Reading[0]` and `Reading[2]` point to the 4^{th} and 5^{th} buffer slots, the writer knows these may be in use, and will not use these buffers. Instead, it will cyclically select and write to the next available slot after `Latest`, the 6^{th} buffer. The worst-case scenario occurs when the last slow reader now makes a read operation. It will now prevent the writer from using the 6^{th} buffer. Even if the three slow readers never relinquish their buffers, the writer can continue to write cyclically to the remaining 4 buffers, with the repeating access pattern $\{7, 1, 2, 3, 7, \ldots\}$. This ensures that no buffer is used more frequently than every fourth write, satisfying the conditions for the fast readers.

The biggest drawback of Chen's algorithm lies in the complexity of the `GetBuff()` function and the expensive CAS instruction itself. As shown in Figure 3, there are three loops inside of this function. The first one loops `NBuffer` times, and the second one loops `NSReader` times. Finally, the last one can potentially loop `NBuffer` times again. Furthermore, the writer has a loop that executes CAS `NSReader` times. As the number of slow readers decreases, we expect the performance enhancement from the Improved Chen's algorithm, as compared to the original Chen's algorithm.

3.2 Double Buffer Algorithm

We have devised a new wait-free IPC mechanism that is less computationally complex than Chen's algorithm. It, however, trades off time for space complexity, requiring approximately twice the buffer space. Hence, it is called the *Double Buffer* algorithm.

The basic constructs of the Double Buffer algorithm are shown in Figure 5, and the algorithm is summarized in Figure 6. A two-dimensional shared message buffer, `Buff[][]`, has $(P + 1)$ rows, where P is the number of reader tasks. Each row has two buffers. Associated with each row i is a usage count, `ReaderCnt[`i`]`, representing the number of readers currently using either buffer in the row, and a flag, `Cl[`i`]`, indicating which of the two buffers is more current. A variable, `Latest`, points to the row containing the most recently written data. A reader task first reads `Latest`, and indicates it is using the row by incrementing the usage count. It then reads the buffer indicated by the row's `Cl` flag, and decrements the row's usage count when it finishes reading. Note that the increment and decrement operate directly on memory variables and must be atomic. This is commonly available on modern processors, including the x86 architecture.

The writer is fairly straightforward. It first scans `ReaderCnt[]`, and selects a row that is not being used by the readers. It then writes to the buffer that was least recently written in the selected row (i.e., opposite to the one indicated by the row's `Cl` flag). We will see why this is necessary shortly. Finally, it updates the row's `Cl` flag to point to the newly-written buffer, and sets `Latest` to the row that contains this buffer. In case each reader is concurrently reading from a unique row, this algorithm requires $(P + 1)$ rows for the writer to work correctly, where P is the number of readers. As each row has 2 buffers, the space required for the message buffer array is $2(P + 1)$.

To see the correctness of the algorithm, let us consider the possible interference scenarios. The writer can only interfere with the reader when they both choose to use the same row. This can only occur in two cases. The first case can occur when a reader is interrupted after it has chosen a row (after line 1), but before it updates the use count (before line 2). The writer then executes, and can potentially choose the same row as the reader. The second case occurs when the writer is interrupted after it has chosen a row (after line 7). If this row happens to be `Latest`, then the reader can also choose to read from this same row. So, it is possible for the readers and the writer to select the same row i. However, the reader will read from the buffer indicated by `Cl[`i`]`, while the writer will use the opposite one. As the writer updates `Cl[`i`]` only after the complete message is written, and the reader always increments the use count before reading `Cl[`i`]`, we can guarantee that the writer and readers cannot interfere with each other in this algorithm, even if they happen to use the same row.

The Double Buffer algorithm is less computationally complex than Chen's algorithms, but has a space requirement twice that of the original Chen's algorithm. In the next section, we use our transformation technique to improve the Double Buffer

```
int NReader;                  # Number of readers
int NRows = NReader + 1;      # Number of rows in the message buffer
int Latest;                   # Index to the row with the latest message
message Buff[NRows][2];       # Message buffer
int ReaderCnt[NRows];         # Reader count for each row
boolean Cl[NRows];            # Column with more up-to-date message

    Reader_i() {
        ridx = Latest;
        inc ReaderCnt[ridx];
        cl = Cl[ridx];
        read Buff[ridx][cl];
        dec ReaderCnt[ridx];
    }
    Writer() {
        for (i = Latest; ; i++)
            if (ReaderCnt[i mod NRows] == 0) break;
        cl = not Cl[i];
        write Buff[i][cl];
        Cl[i] = cl;
        Latest = i;
    }
```

Figure 6: Double Buffer algorithm.

algorithm. As we will see in Section 4, the number of buffers required by the transformed Double Buffer algorithm is usually comparable to, if not less than, the original Chen's algorithm.

3.3 Improved Double Buffer Algorithm

Applying the same techniques used in devising the Improved Chen's algorithm, we now try to improve the Double Buffer algorithm. Again, we divide the reader tasks into fast and slow readers. The fast readers need a minimum of N buffers to ensure temporal concurrency control, while the M slow readers use the original Double Buffer scheme. The total message buffer requirements will now be $2(M+max(1,\lceil\frac{N}{2}\rceil))$ buffers, which is less than or equal to the original algorithm's $2(P+1)$ buffers, assuming correct partitioning of the readers (see Section 3.4). As before, the highest-frequency readers now use the very low overhead NBW read mechanism, so execution times should be improved as well.

The data structures and algorithm for Improved Double Buffer are shown in Figures 7 and 8, respectively. The slow readers are unmodified from the original readers. Fast readers simply read from the buffer indicated by `Latest` and the corresponding row's `Cl` entry. The writer, too, is mostly unmodified. To ensure temporal concurrency control for the fast readers, the writer should not reuse any particular buffer until at least $N-1$ subsequent writes have occurred. This is ensured by changing the buffer selection loop to search starting at row `(Latest+1) mod NRows`. The rows are used cyclically, and the buffers within a row alternate on subsequent writes, so $\lceil\frac{N}{2}\rceil$ rows suffice to ensure temporal concurrency control for the fast readers. Therefore, the improved algorithm needs $2(M+max(1,\lceil\frac{N}{2}\rceil))$ buffers.

To illustrate overhead improvements, let us consider a system with 20 reader tasks, of which 5 are classified as slow readers. Assume further that based on the N_{Max} calculations (Section 2.2), the fast readers need 7 buffers to ensure tempo-

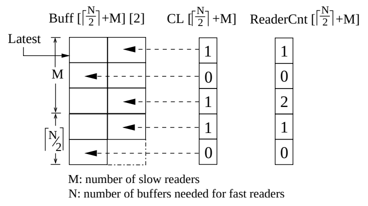

Buff [$\lceil\frac{N}{2}\rceil$+M] [2]
CL [$\lceil\frac{N}{2}\rceil$+M]
ReaderCnt [$\lceil\frac{N}{2}\rceil$+M]
Latest
M
$\lceil N/2\rceil$
1
0
1
1
0
1
0
2
1
0
M: number of slow readers
N: number of buffers needed for fast readers

Figure 7: Constructs in the Improved Double Buffer algorithm.

ral isolation from the writer. With Improved Double Buffer, we need 18 message buffer slots, while the original needs 42, a significant memory reduction. Moreover, the other control variables are proportional to the number of rows, so they, too, are reduced. With the new algorithm, the slow readers and the writer remain virtually unchanged, but the fast readers have less computation than the original readers, so the overall execution overheads will decrease as well. Generally, as the number of fast readers increases, the execution performance increases, but this is not necessarily the case for space requirements. In the following section, we will determine how to partition a reader set into fast and slow readers, optimizing for space.

3.4 Identification of Fast Readers

We now present a simple algorithm for partitioning the reader set into fast and slow readers, optimizing for minimum memory usage. The algorithm is shown in Figure 9, and can be used with any single-writer, multiple-reader IPC scheme improved with our transformation by simply changing a few constants to match the algorithm.

The algorithm initially sets all reader tasks to be slow readers. It keeps the tasks sorted by non-decreasing order of their N_{Max} values, computed as with the NBW protocol (Section 2.2). It tries to move one task at a time from the slow reader set to the fast reader set, and recomputes the number of buffers needed, $(S+F)$, where S is the requirement for the slow readers, and F for the fast readers. By keeping track of the setting with lowest memory use so far, after a single pass through all of the tasks, we obtain the `Splitpoint`, which indicates the last fast reader. All tasks with lower N_{Max} values are also part of the fast reader set.

This partitioning of the reader set is optimal with respect to the number of message buffers. This is easy to show: take a partitioning that is space-optimal, and let task i be the fast reader with the largest N_{Max} value. Now, all tasks with lower N_{Max} values than task i must also be part of the fast reader set (otherwise, we can move them to the fast reader set; they will not affect the number of buffers needed for the fast readers (i.e., largest N_{Max} value), but will reduce the slow reader set's buffer requirements, and the optimality assumption would be invalid). Since the above algorithm considers all partitions in which all tasks with less than a particular N_{Max} value are in the fast reader set, the optimal partition will be found by the algorithm.

```
int Latest;                     # Index to the row with the latest message
int NRows;                      # Number of rows in the message buffer
message Buff[NRows][2];         # Message buffer
int ReaderCnt[NRows];           # Reader count for each row
boolean Cl[NRows];              # Column with more up-to-date message

    SlowReader_i() {
1:      ridx = Latest;
2:      inc ReaderCnt[ridx];
3:      cl = Cl[ridx];
4:      read Buff[ridx][cl];
5:      dec ReaderCnt[ridx];
    }
    FastReader_i() {
6:      ridx = Latest;
7:      cl = Cl[ridx];
8:      read Buff[ridx][cl];
    }
    Writer() {
9:      i = (Latest + 1) mod NRows;
10:     for (; ; i = ((i + 1) mod NRows))
11:         if ( ReaderCnt[i] == 0 ) break;
12:     cl = not Cl[i];
13:     write Buff[i][cl];
14:     Cl[i] = cl;
15:     Latest = i;
    }
```

Figure 8: Improved Double Buffer algorithm.

The partitioning algorithm uses certain constants that depend on the specific IPC mechanism used. For the initialization, `Splitpoint` is always set to NULL and F always set to 0, but `MinNumBuff` and S are both set to the number of buffers needed assuming that all tasks are slow readers. For the Improved Chen's algorithm, this is $(P + 2)$, and for the Improved Double Buffer, it is $2(P + 1)$, where P is the number of tasks. Additionally, V is the number of buffers used for each additional slow reader, and is set to 1 and 2, for Chen's and the Double Buffer mechanisms, respectively.

We illustrate the partitioning algorithm using the sample task set in Figure 10, which indicates the writer's period and relative deadline, as well as the readers' periods (relative deadlines) and computation times. R_{Max} and N_{Max} values, assuming C_R is negligible and the readers' relative deadlines are equal to their periods, are also shown. Assuming Double Buffer algorithm, initially $S =$ `MinNumBuff` = 16, $F = 0$, and all readers are in the slow reader set. Tasks are moved one at a time according to their N_{Max} values, so first, Reader 0 is moved to the fast reader set. Now $S = 14$ and $F = 3$, so $(F + S)$ is not the lowest value seen, and `Splitpoint` is not changed. We continue with Reader 1, resulting in $S = 12$ and $F = 3$, so $S + F \leq$ `MinNumBuff` holds. `Splitpoint` is updated to Reader 1, and `MinNumBuff` is set to $S + F$. We repeat this with all of the readers, in order. By the end, `Splitpoint` points to Reader 4, and `MinNumBuff` = 10. So, with the first five readers as fast readers, we achieve the minimum number of buffers required for this example, a 37.5% reduction from the original algorithm.

```
Order reader tasks by $N_{Max}$ from smallest to largest;
# Note: $S_0$ is the no. of buffers needed if all tasks are slow readers
    $S = S_0$;                  # no. of buffers for slow readers
    $F = 0$;                    # no. of buffers for fast readers
    MinNumBuff = $S_0$;
    Splitpoint = NULL;

For each reader task $T_R$ (ordered by $N_{Max}$)
    Move $T_R$ from the slow reader set to the fast reader set;
    $S = V \times$ sizeof(slow reader set);
    $F = T_R$'s $(N_{Max} + 1)$;
    if ($S + F \leq$ MinNumBuff)
        Splitpoint = $T_R$;
        MinNumBuff = $S + F$;
```

Figure 9: Algorithm to find space-optimal division of fast and slow readers and the amount space required.

Process	P_W	D_W		
Writer	10	7		
	P_R	C	R_{Max}	N_{Max}
Reader 0	8	4	4	2
Reader 1	12	7	5	2
Reader 2	23	14	9	2
Reader 3	22	9	14	3
Reader 4	50	30	20	3
Reader 5	150	25	125	14
Reader 6	500	25	475	49

Figure 10: Task set with one writer and seven reader processes.

3.5 Transformation Mechanism

We have shown here how two different single-writer, multiple-reader wait-free IPC mechanisms can be modified to take into account real-time characteristics of tasks to reduce both memory and execution time overheads. In general, we can apply our transformation to other such IPC algorithms with the following steps.

Step 1. Identify fast and slow readers for a particular system: simply apply the algorithm in Section 3.4. This will minimize the number of message buffers needed, while still ensuring temporal isolation between the writer and the fast readers.

Step 2. Fine-tune reader sets: we may not always want to optimize for space, so we can adjust the partitioning obtained in Step 1 if needed.

Step 3. Convert reader code to slow reader code: Typically, there are no modifications needed for slow readers, so this is just a renaming step.

Step 4. Introduce fast reader code: The fast readers are trivially implemented — they just read the pointer indicating the most recently written message buffer, and then read from that buffer.

Step 5. Modify writer code to ensure temporal isolation with fast readers: this is the most significant change required.

Since most algorithms have some code for selecting a buffer to write, this step usually only requires modifying the selector to ensure that the same buffer is not reused within N consecutive writes. Sometimes, this can simply be done by using the available buffers in a cyclic fashion, and having enough total buffers.

Applying these steps, we can modify existing wait-free single-writer, multiple-reader algorithms to use real-time characteristics of the tasks and reduce processing and memory costs.

4 Performance Evaluation

The goal of our transformation mechanism is to reduce the time and space overheads when applied to single-writer, multiple-reader algorithms. We now evaluate how much improvement we can achieve with the proposed transformation. Specifically, we will compare a total of 9 different IPC mechanisms, including Chen's, Improved Chen's, Double Buffer, and Improved Double Buffer algorithms.

We also consider another wait-free, single-writer, multiple-reader IPC mechanism, Peterson's algorithm, as well as our transformed version of it. In Peterson's algorithm [24], the reader determines if its read is corrupted, and may have to perform the read up to 3 times. The writer may also have to write a message up to ($P+2$) times, where P is the number of readers. The mechanism has been revised [34] such that readers read a message at most 2 times, and the writer writes a message at most ($P+1$) times to avoid corruption. We only consider the revised version here. We derive the Improved Peterson's algorithm by applying our general transformation as described in Section 3.5.

For the purpose of comparison, we also evaluate the NBW protocol and the EMERALDS variant of this. As discussed earlier, NBW is the most efficient algorithm in terms execution time, but may induce high space overheads. The EMERALDS IPC mechanism tries to limit memory use at some cost to performance. Finally, we also include a very efficient implementation of synchronization-based IPC, using a lock algorithm that relies on the atomic Test-And-Set instruction, to show the tradeoffs between synchronization-based and synchronization-free mechanisms.

To make fair and comprehensive comparisons between these algorithms, we have considered various parameters trying to answer the following questions.

- How much does the transformation reduce the average-case and worst-case execution times?
- How much does the transformation reduce the buffer space requirement?
- Is the transformation applicable in both uniprocessor and multiprocessor environments? How do they differ?

Class	Subclass	Percentage within Class	Relative Frequency to the Writer Process
Fast	Fastest	15–25%	twice as frequent
	Fast	75–85%	1–15 times less frequent
Slow	Slow	75–85%	15–50 times less frequent
	Slowest	15–25%	50–100 times less frequent

Figure 11: Reader task set distribution.

- Will different message sizes affect the results?
- Will the size of the reader set affect the results?

We evaluate the algorithms for memory usage and execution time overheads, in both average and worst cases, and for both uniprocessor and symmetric multiprocessor (SMP) environments. The only exception is for the EMERALDS IPC mechanism, which is evaluated only for uniprocessors. Because it assumes that operations are atomic if interrupts are disabled, it will not work correctly with SMP architectures where this assumption does not hold.

4.1 Experiment Setup

The algorithms we evaluate in this section are implemented and executed under EMERALDS OS [35] running on a Pentium-III 500Mhz processor. The experiments use a synthetic reader task set, which is divided into two sets — fast readers and slow readers, where 'fast' and 'slow' are defined relative to the writer's period. In a real system, there are usually tasks that are executed very frequently, and tasks that run very infrequently. To model this behavior, we further divide the fast and slow reader sets into finer-grained categories, as shown in Figure 11. By making approximately 20% of fast and slow readers either very fast or very slow, the resulting task set represents realistic range of task periods that may occur in a real-time embedded system. A random reader task set is generated for each experiment according to the desired division of readers into the four categories.

4.2 Average vs. Worst-case Execution Time

The average-case (ACET) and worst-case execution times (WCET) to perform an IPC read/write operation are both important factors in the performance of an IPC algorithm. A low ACET would indicate that the algorithm generally incurs low computation overheads. However, to provide timeliness guarantees in embedded real-time systems, the scheduler must account for the WCET. An algorithm with low ACET but high WCET may result in poor system utilization.

The ACET and WCET of the SMP versions of the eight evaluated algorithms are shown on the top and bottom rows, respectively, in Figure 12. The SMP versions of the algorithms include bus-lock operations to ensure the atomic operation of the critical CAS and TAS instructions with multiple processors. The message size is 8 bytes, and the task set consists

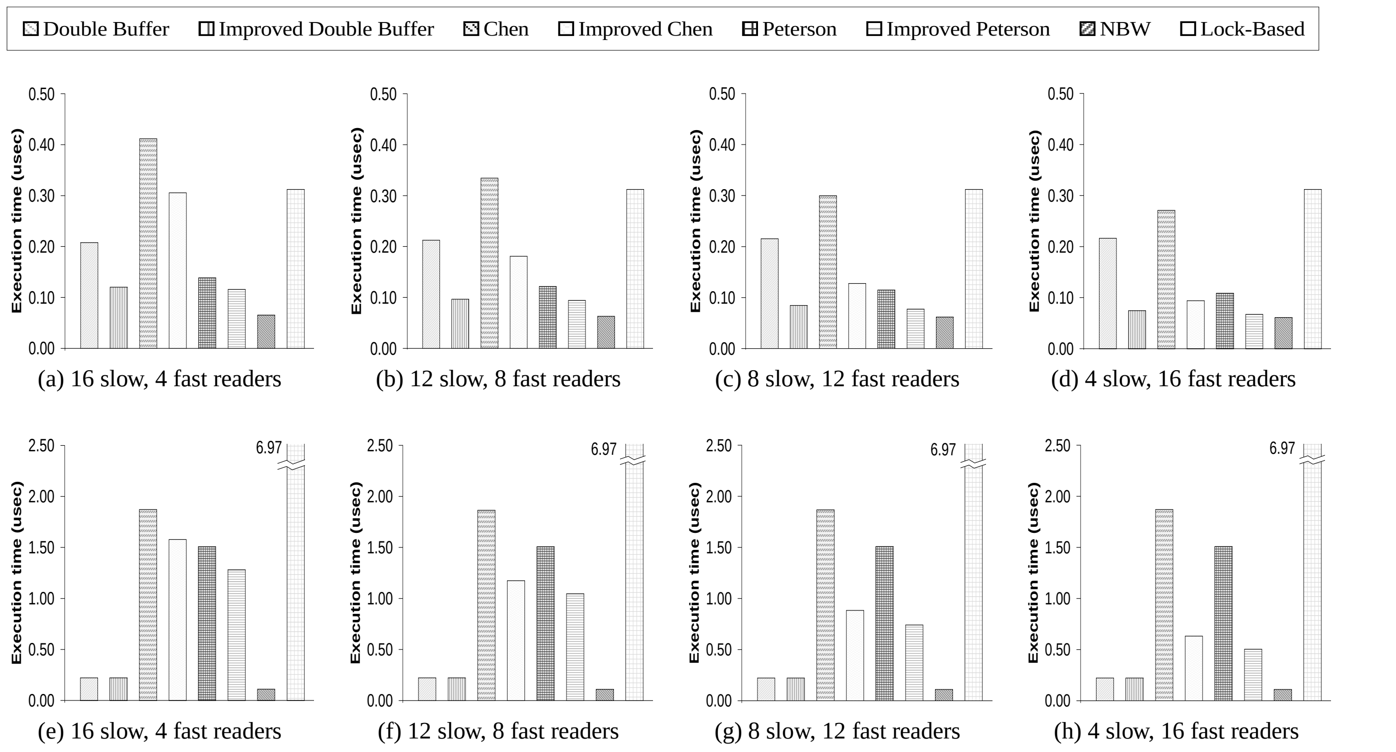


Figure 12: The top and bottom rows show the average-case and worst-case execution times, respectively, of the SMP version of the algorithms, to perform an IPC read / write operation with 8-byte message size.

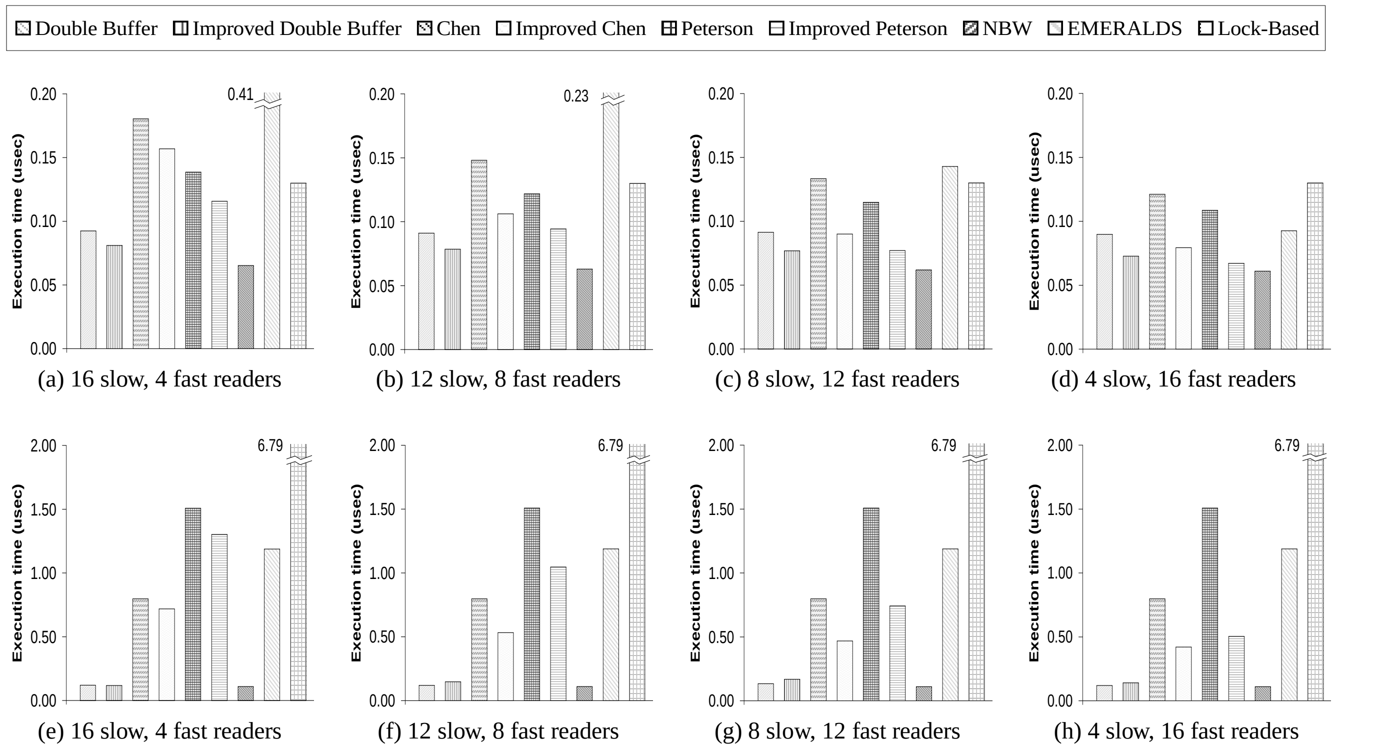
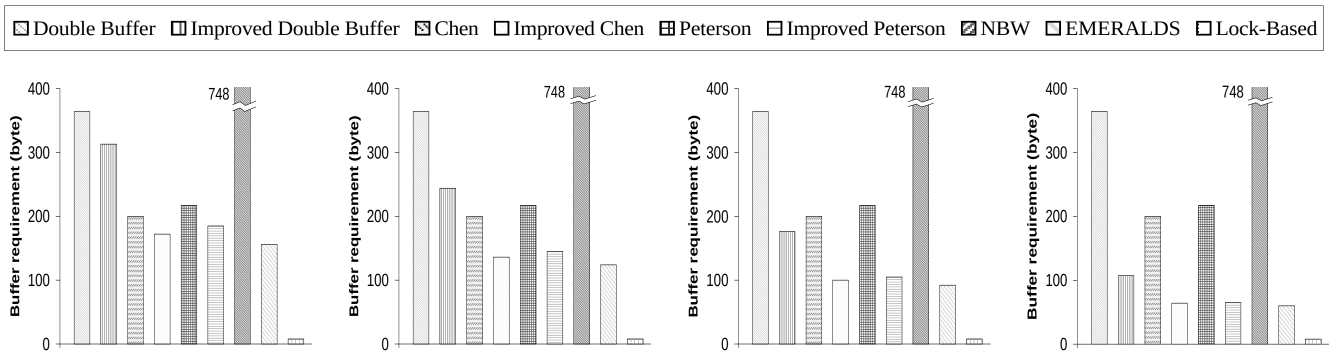


Figure 13: The top and bottom rows show the average-case and worst-case execution times, respectively, of the uniprocessor version of the algorithms, to perform an IPC read / write operation with 8-byte message size.



(a) 16 slow, 4 fast readers (b) 12 slow, 8 fast readers (c) 8 slow, 12 fast readers (d) 4 slow, 16 fast readers

Figure 14: These graphs show the space requirements for different algorithms with 8-byte messages (note that the space requirement is architecture-independent).

of 1 writer and 20 readers, of which a varying fraction are fast readers. Specifically, we evaluated these algorithms when the reader set contains 20%, 40%, 60% and 80% fast readers. The first three pairs of columns on the graphs are for the three single-writer, multiple-reader algorithms and their corresponding transformed algorithms. We can see significant reductions in both the ACET and WCET from comparison of the transformed algorithms with the original ones. As the number of fast readers in the reader set increases, the reduction in computation time for the transformed algorithms gets more pronounced. ACETs for Double Buffer and Chen's algorithms improve by as much as 66%, and for Peterson's algorithm by as much as 38%. This trend is shown in Figure 15(a).

Although the amount of improvement is a non-decreasing function of the percentage of fast readers in the reader set, the magnitude of this improvement depends on the particular algorithm. In these experiments, all of the transformed algorithms perform better than the original versions except for the WCET of the Double Buffer algorithm. This can be attributed to the fact that the WCET for the Double Buffer algorithm occurs in the slow readers. As this time is not affected by the number of slow readers in the system, the WCET does not improve. For the other algorithms, the WCETs occur in the writers, whose overheads are functions of the number of slow readers, and, therefore, improve greatly.

It is interesting to note that even though the ACET of the lock-based algorithm is only up to 4 times larger than those of the transformed wait-free algorithms, its WCET is much higher — 4 to 30 times higher. This is in fact an underestimate of the true overhead of the lock-based mechanism, since we assume no blocking time here. In actual systems, unless the system employs mechanisms to limit blocking times, the lock-based execution time may be unbounded.

4.3 Uniprocessor vs. SMP

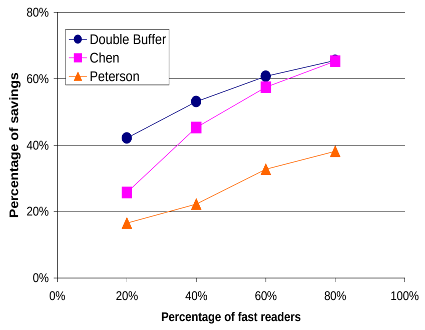
The correctness of some asynchronous algorithms rely on the fact that certain instructions will be executed atomically. For example, Chen's algorithm requires that the CAS instruction be performed atomically. For SMP architectures, this requires that expensive bus-locking (e.g., by using the *LOCK* prefix in the x86 architecture) be performed to ensure an atomic read-modify-write of memory. Under uniprocessor environments, however, such measures are generally not needed. In most architectures, including x86, these instructions are already guaranteed to be atomic with respect to uniprocessor systems without incurring any additional overheads. As a result, we can reduce the costs of CAS for Chen's algorithm, atomic *inc* and *dec* for Double Buffer, and TAS for the lock-based mechanism. We now repeat the above experiments, but using code restricted to uniprocessor machines. The results, including evaluations of the EMERALDS IPC mechanism, are shown in Figure 13.

As expected, the ACET and WCET of these algorithms are lower than their counterparts for SMP. Even in this case, we can still save a significant percentage of execution time overheads. It is worth noting how close the ACETs of the transformed algorithms are to the optimal NBW protocol execution time. WCET improvements after transformation are even more pronounced than for ACET, except with the Double Buffer algorithm. This anomaly is due to the complexity we have introduced in the writer to handle both kinds of readers. Nonetheless, the WCET of the Double Buffer algorithm is still very close to that of NBW.

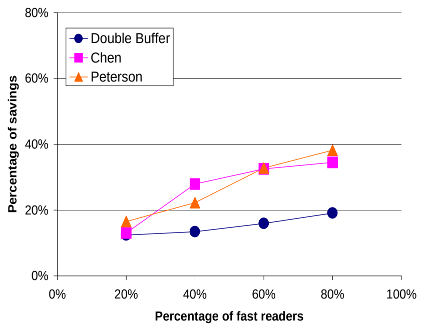
We summarize the reduction in ACET as the percentage of fast readers changes in Figure 15(b). Compared to the SMP results in Figure 15(a), the ACET reduction in uniprocessor environments is less pronounced. Nonetheless, our transformation still reduces a good amount in execution time.

4.4 Savings in Space

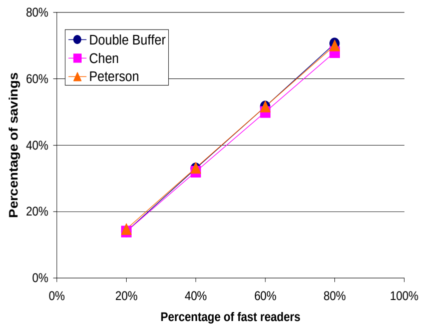
Thus far, we have shown that our transformation mechanism enhances the performance of algorithms in terms of the ACET and the WCET in both SMP and uniprocessor environments. Here, we present results to support our claim that the transformation mechanism not only reduces the time overheads but



(a) Savings in ACET (SMP)



(b) Savings in ACET (UP)



(c) Savings in space required

Figure 15: Varying the percentage of fast readers in the reader set, (a) and (b) show the percentage of savings in ACET for SMP and uniprocessor versions of the algorithms, respectively. (c) shows the percentage of savings in space.

also the space overheads of the algorithms. This is shown in Figure 14. Again, we varied the percentage of fast readers in the reader set. As expected, the amount of improvement increases as the number of fast readers increases. Moreover, the synchronization-based algorithm requires the least space, since it only needs a single shared message buffer. The NBW protocol and lock-based IPC, therefore, represent the extreme cases for the tradeoff between space requirements and WCET. The non-blocking IPC mechanisms, especially with our transformation, provide a good compromise, balancing WCET and memory usage.

Interestingly, the percentage of space reduction for all three transformed algorithms is the same, as shown in Figure 15(c). This does make sense since the memory requirements of the three original algorithms are all proportional to the number of readers in the reader set. So, the memory used by the transformed algorithms decreases proportionally to the number of slow readers. The slight variations in Figure 15(c) are due to some of the control variables that do not scale with the number of reader tasks. Overall, we achieve a reduction in memory usage that ranges from 14 to 70%.

4.5 Effects of Message and Reader Set Size

The experiments in the previous sections all use 8-byte messages. To see how varying the message size affects the savings in time and space, we have performed the same set of experiments with larger messages (64 bytes). The measurements follow a similar trend, but the percentage reduction in execution time is less than when using 8-byte messages. This is because the execution overhead of the actual message buffer read/write operation, which cannot be reduced, becomes a more dominant part of the total execution overheads. The percentage reductions in space overheads are the same, or slightly better than for the 8-byte message case, since the constant overheads of some of the control variables are less apparent. Due to the substantially similar results, the 64-byte message measurements are not presented here.

We have also conducted experiments while varying the total size of the reader set. Running the previous experiments with 10 reader tasks resulted in nearly identical relative performance improvements with our transformation mechanism. Of course, with fewer readers, any complexity increase in the writer task has greater weight in the average execution time, but this is offset by the performance gains in the fast readers. Space reduction, as before, is basically linear to the percentage reduction in the number of slow readers. Again, due to their substantially similar results, the data for the 10 readers case are omitted here.

5 Related Work

Some earlier work [17, 20] on lock-free objects was done using read-and-check loops. The reader is required to check if its reading was interfered with by the writer, in which case it performs the read operation again until it succeeds. Optimization techniques to reduce the number of loops were proposed in [15], using an exponential backoff policy. Kopetz *et al.* [17] and Anderson *et al.* [2] later demonstrated how to bound the number of retries by either increasing the buffer size or through judicious scheduling.

To reduce the time overheads associated with read-and-check loops, algorithms that make space and time tradeoffs were later proposed [6–8, 17, 24, 31, 35]. These algorithms provide a good middle-ground between the purely lock-based approach (high WCET) and the purely buffer-based approach (large buffer requirement). The benefit of these algorithms is that less time is wasted in read-and-check loops and the timing behavior is more predictable, improving schedulability of task sets as well as system utilization. Although the timing behavior is more predictable, the computational complexity of these algorithms is still high. Moreover, they may still incur a large buffer space requirement, and may be difficult to use in small-memory embedded systems. This difficulty can be overcome by our transformation mechanism, which makes significant reductions in both time and space overheads.

Most non-blocking algorithms rely on the availability of some form of atomic memory update instructions, such as Compare-And-Swap or Load-Linked/Store-Conditional in hardware. A few modern hardware platforms, however, do not implement some of these instructions. The author of [23] demonstrated how to emulate these instructions by synthesizing more commonly-implemented instructions to close the gap between the primitives that the algorithm designers rely upon, and the primitives provided by the hardware. Bershad [5] proposed how to implement CAS instruction in software by using operating system support, and Greenwald *et al.* [12] generalized this technique to implement Double-Word CAS and Multi-Word CAS instructions. Similar work was done in Synthesis [22] and Cache kernel [12]. Our transformation mechanism does not use such operations, so it is not directly affected by whether the atomic operations used by the original IPC algorithms are supported by the hardware or are emulated. However, the degree of performance improvement will be different. All of the algorithms we evaluated use atomic update instructions supported natively by the x86 architecture. We expect an even greater improvement with our transformation if these instructions are emulated since the overheads for emulation will most likely be higher.

Herlihy [13] proposed the first general methodology to transform sequential data objects to the equivalent non-blocking structures. Alemany *et al.* [1] and LaMarca [19] proposed techniques to reduce the inefficiencies in applying this methodology to large objects at the cost of more communication between the application process and the operating system. Other methods to improve this were proposed in [4, 32]. Prakash *et al.* [26] and Turek *et al.* [32] presented techniques to transform multiple-lock concurrent objects into lock-free objects. However, it was shown that their transformed algorithms are less efficient than the corresponding lock-based algorithms [15, 19, 30]. These authors are concerned with transforming sequential objects to non-blocking objects, and the related performance issues. We take the next logical step by transforming non-blocking objects, in particular, those with single-writer, multiple-reader semantics, to better-performing and less space-consuming non-blocking objects.

Some interesting work [14, 15, 27] has also been done in the construction of more complex concurrent objects. Concurrent non-blocking array-based stacks, FIFO queues and multiple lists were implemented using Double-Compare-And-Swap in [12]. Valois introduced non-blocking algorithms for queues, linked-lists, and arrays in [33]. Eliot *et al.* [11] proposed non-blocking algorithms for garbage collection. We do not look at these complex structures, but focus instead on the more common, single-writer, multiple-reader state message construct, used for IPC in embedded systems.

6 Conclusions

In this paper, we have argued for efficient IPC mechanisms, particularly for memory- and processing-power- constrained embedded real-time systems. Traditional and synchronization-based IPC methods incur too much time overhead and follow incorrect semantics for most of such systems. Instead, we considered wait-free, single-writer, multiple-reader IPC algorithms, which are more appropriate for these systems, but still can incur substantial overheads.

By taking advantage of the temporal characteristics of the tasks in these systems, we have proposed a general transformation mechanism that can significantly reduce both space and time overheads of the wait-free IPC algorithms. This allows the most frequently-executing reader tasks to use very low-overhead operations, while reducing the total number of buffers needed to ensure corruption-free message passing. We have demonstrated our transformation on the existing Chen's algorithm and the new Double Buffer algorithm that we have introduced here.

Our extensive experiments show a 17–66% reduction in ACET, and a 14–70% reduction in memory requirements for the IPC algorithms improved with our transformation. For algorithms with relatively high WCETs, these are shown to be improved greatly as well. The experiments also demonstrate the tradeoff between time and space in IPC mechanisms: the NBW protocol is time-optimal, but requires large buffers, while a lock-based approach requires just a single message buffer, but suffers from very high worst-case execution overheads. Overall, the single-writer, multiple-reader non-blocking algorithms are good intermediate solutions, balancing WCET and space requirements. With our transformation, we can do even better, reducing both time and space requirements of these algorithms.

This transformation mechanism can be applied to other non-blocking IPC algorithms that are not considered here, and make them better optimized for systems with real-time characteristics. In the future, we would like to extend our methodology to reduce synchronization overheads in more general IPC algorithms with multiple-writer semantics and to extend this to more general communication channels as well.

7 Acknowledgments

We would like to thank our shepherd, Carla Ellis, and the anonymous reviewers for their excellent feedback. We also like to thank John Reumann for some helpful discussions earlier in this work.

References

[1] J. Alemany and W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pages 125–134, 1992.

[2] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 28–37, Dec 1995.

[3] T. P. Baker. Stack-based scheduling of realtime processes. *RT-SYSTS: Real-Time Systems*, 3, 1991.

[4] Greg Barnes. A method for implementing lock-free data structures. In *5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 93)*, 1993.

[5] B. Bershad. Practical considerations for non-blocking concurrent objects. In *IEEE International Conference on Distributed Computing Systems*, pages 264–273, 1993.

[6] James E. Burns and Gary L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *ACM Symposium on Principles of Distributed Computing*, pages 222–231, 1987.

[7] J. Chen and A. Burns. A fully asynchronous reader/writer mechanism for multiprocessor real-time systems. Technical Report YCS-288, Department of Computer Science, University of York, 1997.

[8] J. Chen and A. Burns. A three-slot asynchronous reader/writer mechanism for multiprocessor real-time systems. Technical Report YCS-186, Department of Computer Science, University of York, 1997.

[9] P. Courtois, F. Heymans, and D. Parnas. Concurrent control with readers and writers. *Communications of the Association of Computing Machinery*, 14(10):667–668, 1971.

[10] E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the Association of Computing Machinery*, 8(9):569, 1965.

[11] J. Eliot and B. Moss. Lock-free garbage collection for multiprocessors. In *Parallel Algorithms and Architectures*, 1991.

[12] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Operating Systems Design and Implementation*, pages 123–136, 1996.

[13] M. Herlihy. A methodology for implementing highly concurrent data structure. *Proceeding of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, 1989.

[14] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[15] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[16] H. Kopetz and *et al*. Distributed fault-tolerant real-time systems: the Mars approach. *IEEE Micro*, 9(1):25–40, 1989.

[17] H. Kopetz and J. Reisinger. The non-blocking write protocol NBW: A solution to a real-time synchronisation problem. In *IEEE Real-Time Systems Symposium*, pages 131–137, 1993.

[18] C. M. Krishna and K. G. Shin. *Real-Time Systems*. McGraw-Hill, 1997.

[19] Anthony LaMarca. A performance evaluation of lock-free synchronization protocols. In *ACM Symposium on Principles of Distributed Computing*, pages 130–140, 1994.

[20] L. Lamport. Concurrent reading and writing. *Communications of ACM*, 20(11):806–811, Nov 1977.

[21] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[22] H. Massalin and C. Pu. A lock-free multiprocessor os kernel. Technical Report CUCS-005-91, Department of Computer Science, Columbia University, 1991.

[23] M. Moir. Practical implementations of non-blocking synchronization primitives. In *ACM Symposium on Principles of Distributed Computing*, pages 219–228, 1997.

[24] G. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, 1983.

[25] S. Poledna, T. Mocken, and J. Schiemann. Ercos: an operating system for automotive applications. Technical Report 960623, Society of Automotive Engineers Technical Paper Series, 1996.

[26] S. Prakash, Y.-H. Lee, and T. Johnson. Non-blocking algorithms for concurrent data structures. Technical Report 91–002, Department of Computer Science, University of California, Los Angeles, 1991.

[27] S. Prakash, Y.-H. Lee, and T. Johnson. A non-blocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, 43(5):548–559, 1994.

[28] R. Rajkumar. Synchronization in real-time systems - a priority inheritance approach. *Kluwer Academic Publishers*, 1991.

[29] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[30] Nir Shavit and Dan Touitou. Software transactional memory. In *ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[31] H. R. Simpson. Four-slot fully asynchronous communication mechanism. In *IEEE Proceedings*, 1990.

[32] John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *ACM Proceedings of the Principles of Database Systems*, pages 212–222, 1992.

[33] J. D. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages 64–69, Las Vegas, NV, 1994.

[34] K. Vidyasankar. Concurrent reading while writing revisited. *Distributed Computing*, 4(2):81–86, 1990.

[35] K. Zuberi, P. Pillai, and K. G. Shin. EMERALDS: a small-memory real-time microkernel. In *ACM Symposium on Operating Systems Principles*, volume 34, pages 277–299, 1999.