Waiting Algorithms for Synchronization in Large-Scale Multiprocessors

Beng-Hong Lim and Anant Agarwal
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Through analysis and experiments, this paper investigates two-phase waiting algorithms to minimize the cost of waiting for synchronization in large-scale multiprocessors. In a two-phase algorithm, a thread first waits by polling a synchronization variable. If the cost of polling reaches a limit L_{poll} and further waiting is necessary, the thread is blocked, incurring an additional fixed cost, B. The choice of L_{poll} is a critical determinant of the performance of two-phase algorithms. We focus on methods for statically determining L_{poll} because the run-time overhead of dynamically determining L_{poll} can be comparable to the cost of blocking in large-scale multiprocessor systems with lightweight threads.

Our experiments show that *always-block* ($L_{poll} = 0$) is a good waiting algorithm with performance that is usually close to the best of the algorithms compared. We show that even better performance can be achieved with a static choice of L_{poll} based on knowledge of likely wait-time distributions. Motivated by the observation that different synchronization types exhibit different wait-time distributions, we prove that a static choice of L_{poll} can yield close to optimal on-line performance against an adversary that is restricted to choosing wait times from a fixed family of probability distributions. This result allows us to make an optimal static choice of L_{poll} based on synchronization type. For exponentially distributed wait times, we prove that setting $L_{poll} = \ln(e-1)B$ results in a waiting cost that is no more than $e/(e-1)$ times the cost of an optimal off-line algorithm. For uniformly distributed wait times, we prove that setting $L_{poll} = \frac{1}{2}(\sqrt{5}-1)B$ results in a waiting cost that is no more than $(\sqrt{5}+1)/2$ (the golden ratio) times the cost of an optimal off-line algorithm. Experimental measurements of several parallel applications on the Alewife multiprocessor simulator corroborate our theoretical findings.

1 Introduction

Threads executing on a multiprocessor synchronize to ensure program correctness. As multiprocessors scale in size, the grain size of threads will decrease to satisfy higher parallelism requirements [10], causing a corresponding increase in synchronization rates and in the frequency of waits due to synchronization. Waiting threads waste processor cycles and incur a cost that is related not only to the wait times encountered but also to the efficiency of the waiting algorithm.

This paper studies two-phase waiting algorithms for synchronization in large-scale multiprocessors, and compares them with traditional always-block and always-poll algorithms. We use *expected*

This research was supported in part by the National Science Foundation under grant MIP-9012773, and in part by the Defense Advanced Research Projects Agency under contract N00014-87-K-0825. This paper appears in ACM Transactions on Computer Systems, Vol. 11, No. 3, August 1993.

waiting cost as a metric for comparing the algorithms. Waiting cost measures the efficiency of a waiting algorithm and refers to the processor cycles wasted by a thread while waiting. This cost depends on several factors including the wait-time distribution and the waiting mechanisms used. It however does not include the run-time overhead of the waiting algorithm itself.

Wait time is the interval from the instant a thread begins waiting on a synchronization condition to the instant that synchronization condition is satisfied. (The second instant is the earliest time at which the waiting thread is allowed to proceed.) Wait times depend on the program and also on the grain size of the threads and the size of the machine. In general, wait times are hard to predict statically, justifying the need for on-line waiting algorithms.

A *waiting mechanism* is an action taken by a thread while waiting on a synchronization condition. Two fundamental types of waiting mechanisms are *polling* and *signalling*. With polling, the waiting thread repeatedly probes a synchronization variable and proceeds when the variable attains a desired value. With signalling, the waiting thread suspends execution and relinquishes control of the processor until the synchronization condition is satisfied. Traditionally, multiprocessors provide *spinning* and *blocking* as mechanisms for polling and signalling respectively. Multithreaded multiprocessors, such as Alewife [3], additionally provide more efficient polling and signalling mechanisms called *switch-spinning* and *switch-blocking*, which will be described in Section 2.

A *waiting algorithm* chooses among available waiting mechanisms during synchronization faults. A common waiting algorithm used in shared-memory multiprocessors is to always-spin. If the synchronization condition is satisfied in a short time, a spinning thread can proceed quickly. However, retaining control of the processor while spinning creates a potential for deadlock. Another common waiting algorithm is to always-block, which forces the waiting thread to surrender the processor – usually an expensive operation – to another thread.

1.1 Spinning versus Blocking

Existing multiprocessors provide spinning and blocking as the only waiting mechanisms and rely on the programmer to make the correct choice. This choice is critical to performance due to potentially significant differences between the waiting costs of spinning and blocking. The waiting cost of spinning is equal to the wait time, t, measured in processor cycles. The waiting cost of blocking, B, is the number of processor cycles wasted in *unloading* and suspending the waiting thread, and then rescheduling and *reloading* it at a later time. Unloading a thread involves storing its processor-resident state into memory and reloading a thread involves restoring the saved state onto the processor.

Although it is apparent that spinning is appropriate when wait times are small relative to the cost of blocking, it is hard to make a correct choice in the face of uncertain program behavior. Long wait times hurt the performance of spinning. Furthermore, the programmer has to be responsible for avoiding deadlocks. On the other hand, blocking can incur a significant fixed cost for each synchronization fault because of the need to save and restore processor state.[1]

Without *a priori* knowledge of wait times, a more sophisticated algorithm is needed to select appropriate waiting mechanisms at run-time. An algorithm that combines the advantages of polling and signalling is the two-phase waiting algorithm, first suggested by Ousterhout [25]. In a two-

[1] As anecdotal evidence, blocking was tried as a performance enhancer for a system routine in the DYNIX operating system for the Sequent multiprocessor, unexpectedly causing bad performance under certain conditions. This fact was subsequently used in an advertising campaign by a competitor. See [32].

phase waiting algorithm, a waiting thread first polls a synchronization variable until the cost of polling reaches a limit L_{poll}. If further waiting is necessary at the end of the polling phase, the thread resorts to a signalling mechanism for waiting, incurring a fixed cost B. Thus, for short wait times, the waiting thread can proceed without incurring the overhead of a signalling mechanism.

The choice of L_{poll} is key to the performance of two-phase waiting algorithms. In a sense, two-phase waiting is a generalization of the always-spin and always-block algorithms: it introduces a continuum of choices between always-block ($L_{poll} = 0$) and always-spin ($L_{poll} = \infty$). Thus, the problem of *deciding the value of* L_{poll} *in a two-phase algorithm* replaces the the problem of deciding whether to poll or to signal.

The choice of L_{poll} can be made statically at compile time, or dynamically at run-time. In this paper, we focus on *static methods* for determining L_{poll} to minimize run-time overhead. (This run-time overhead is not to be confused with waiting cost.) Minimizing the run-time overhead of the waiting algorithm is crucial in large-scale multiprocessors that necessarily support lightweight threads. In such systems, the run-time overhead of dynamic methods can be comparable to blocking overheads. Noteworthy static choices for L_{poll} explored in this paper are 0, $0.54B$, $0.62B$, B, and ∞. Section 4 will analyze the relative performance of these different static choices and show that we can use readily available knowledge of synchronization types and their expected wait times to guide our choice.

1.2 Contributions of this Work

While two-phase waiting itself is not a new idea, and has been previously studied both analytically and empirically, this work provides new results by considering practical aspects of a large-scale parallel machine environment. It extends earlier work in significant ways by combining analysis and experimentation.

Previous empirical work [18] focused on waiting for spin-locks on small-scale bus-based machines. The empirical work here considers multiple types of synchronization in the context of a large-scale distributed-memory machine with the following characteristics:

- *Longer communication latencies* in large multiprocessor systems, which impact the effectiveness of polling as a waiting mechanism.
- Run-time systems that are optimized for lightweight thread management resulting in *low blocking overheads*. This makes blocking an effective waiting mechanism. When blocking overhead is low, two-phase algorithms are useful only if their run-time overhead is negligible.
- Processor architectures providing *efficient support for synchronization* in the form of streamlined trap interfaces for detecting synchronization faults, and multiple hardware contexts to support switch-spinning as an alternative to spinning.
- Use of a variety of *scalable synchronization types* typically found in large-scale multiprocessor application codes, including distributed barriers, fine-grain producer-consumer data-structures, and distributed locks. These allow the expression and efficient execution of fine-grain parallelism and lead to more frequent synchronization operations and shorter wait times compared to programs on small-scale machines that use less scalable synchronization methods.

In previous analytical work, Karlin *et al.* [19] present an efficient randomized waiting algorithm that achieves an optimal on-line competitive factor of $e/(e-1)$ against a *weak adversary*. (See

Section 4 for a definition of competitive factors, adversaries and optimality.) We prove in this paper that if we restrict the adversary by fixing the wait time distribution and allowing it to control only the parameters of the distribution – a reasonable practical assumption – a static choice of L_{poll} can also approach optimal on-line competitive factors against this *restricted adversary*.

Based on analysis and practical considerations, we explore the idea of predicting wait time distributions based on synchronization type and then linking the choice of L_{poll} to the synchronization type, a decision that can be easily made by a compiler. In other words, we expect each synchronization type to have a characteristic wait-time distribution, knowledge of which allows informed static choices for L_{poll}.

Under the assumption of Poisson arrivals of synchronizing threads, we show that the exponential and uniform distributions are reasonable models of wait times for producer-consumer and barrier synchronization respectively. These distributions are also applicable to wait times for mutual exclusion locks under low contention. Empirical measurements of wait times for several parallel applications using these synchronization types corroborate the models.

We prove that for exponentially distributed wait times, statically setting L_{poll} to $\ln(e-1)B \approx 0.54B$ yields an optimal on-line algorithm against a restricted adversary, and that no dynamic algorithm can achieve better performance against this restricted adversary. Empirically, we observed that setting L_{poll} to $0.5B$ resulted in better performance for producer-consumer synchronization than when $L_{poll} = B$. We also prove that for uniformly distributed wait times, setting L_{poll} to $\frac{1}{2}(\sqrt{5}-1)B \approx 0.62B$ yields an algorithm that is close to optimal. It is important to note that because the optimality of these settings of L_{poll} are *independent* of the actual parameters of the distributions, they can be chosen statically.

When the wait-time distributions are not known *a priori*, this paper demonstrates through experimental measurements of parallel applications that two-phase waiting with L_{poll} set to B is a robust algorithm under most circumstances. That is, the performance of two-phase waiting is either the best, or close to the best, waiting algorithm compared to always-poll or always-block. These measurements support our theoretically derived result that two-phase waiting (under exponentially-distributed wait times) is never worse than both spinning or blocking used exclusively.

Empirical results confirm the intuition that always-poll is an unacceptable waiting algorithm when there are more threads than hardware contexts due to the potential for deadlock and the use of non-preemptive scheduling in the experiments. On the other hand, always-block is found to be generally efficient, except in one case where wait times were mostly shorter than B.

This observation is contrary to results recently reported in [18], and is due to several factors. The study in [18] is focused on waiting for mutual-exclusion locks on a small bus-based multiprocessor. On such a machine, communication latencies are shorter than in network-based multiprocessors. Moreover, the blocking overhead on their system is higher than on Alewife, and wait times for mutual-exclusion locks are typically shorter than those for other synchronization types.

The rest of this paper is organized as follows. Sections 2 and 3 provide an in-depth discussion of waiting mechanisms and algorithms. Section 4 presents a theoretical framework for making an informed choice of L_{poll} for two-phase waiting algorithms. Section 5 describes experiments carried out to measure the performance of two-phase algorithms and to corroborate the theoretical results. Section 6 presents and discusses the experimental results. Section 7 discusses related work and Section 8 concludes the paper.

2 Waiting Mechanisms

We describe in this section the implementation and the waiting costs of the following waiting mechanisms: *spinning, blocking, switch-spinning, and switch-blocking.* The first two mechanisms are commonly found in traditional multiprocessors, while the last two are additional mechanisms provided by multithreaded multiprocessors. To provide firmer grounding, we describe these mechanisms in the context of the MIT Alewife multiprocessor [3], a distributed-memory multiprocessor supporting the shared-memory programming model. We begin with a brief overview of multithreading.

Multithreading is commonly prescribed as a method for tolerating latencies and increasing processor utilization in a large-scale multiprocessor. It accomplishes this by rapidly switching control of the processor to a different thread whenever a high latency operation is encountered. While previous multithreaded designs switch contexts at every cycle [27, 12], Alewife's multithreaded processor [2] switches contexts only on synchronization faults and remote cache misses. This style is called *block multithreading* [20] and has the advantage of high single thread performance.

Since multithreading introduces novel methods for manipulating threads, we define the following terms to avoid ambiguity.

Hardware context – A set of registers that implements the processor-resident state of a thread. A multithreaded processor has multiple hardware contexts to hold multiple threads at once.

Loading/Unloading – Loading a thread refers to the action of installing the state of a thread into a hardware context on a processor, and unloading a thread refers to the complementary action of saving the processor-resident state of a thread into memory.

Context switch – A transfer of processor control from a processor-resident thread to another processor-resident thread in a multithreaded processor. This should not be confused with traditional context switching, where the threads are unloaded and reloaded.

We now describe the waiting mechanisms and their associated waiting costs as a function of t, the wait time.

Spinning A thread spin-waits by continuously reading the value of a memory location. In cache-coherent multiprocessors such as Alewife, the spin location is cached locally to avoid network traffic while spinning. A change to the state of the memory location due to a write is communicated to the waiting threads through the ensuing cache invalidations. Because cycles spent spinning are wasted, the waiting cost of spinning for t cycles is equal to t.

Blocking Blocking a thread involves unloading it, and at a later time, reenabling and reloading it. Thus, a blocked thread allows other threads to use the processor. On Alewife, the blocked thread is placed on a software queue associated with the failed synchronization. When signalled to proceed, the thread is reenabled and eventually rescheduled and reloaded. The waiting cost of blocking, B, is the number of cycles needed to unload, reenable and reload a thread. The blocking routines used in this study were empirically observed to cost approximately 500 cycles. See Section 6 for a detailed breakdown of this cost and suggestions on how it can be further reduced. The experiments assume $B \approx 500$ cycles.

Switch-Spinning On a multithreaded processor, a waiting thread can switch rapidly to another processor-resident thread in a round-robin fashion, allowing the wait to be overlapped with useful

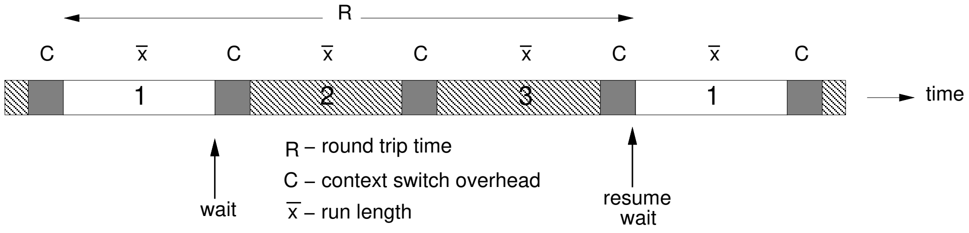

Figure 1: Switch-Spinning – time line of three active contexts sharing a processor. A switch-spinning thread occupies context 1 and its wait time is interleaved with executions of threads in context 2 and context 3.

computation by other threads. Control eventually returns to the waiting thread and the synchronization variable is re-polled. Switch-spinning is therefore a polling mechanism. Since other threads are allowed to utilize the processor, this is a more efficient polling mechanism than spinning.

We model the cost of switch-spinning for t cycles as t/β, where β is defined as the relative efficiency of switch-spinning over spinning. In other words, a switch-spinning thread that waits for t cycles wastes only t/β processor cycles. The following analysis models the value of β in a block-multithreaded processor.

Figure 1 illustrates a switch-spinning scenario with three hardware contexts. β depends on the number of hardware contexts, N, the context switch overhead, C, and the run length. Let $\bar{x}$ be the mean run length. Run length is the time between the instant a thread starts executing on the processor to the instant it encounters a context switch. Let R be the round-trip time, defined as the time between successive context switches to the same switch-spinning thread. R can be approximated by $N(\bar{x} + C)$. On Alewife, $N = 4$ and $C = 12$.

Suppose that a thread has to wait for t cycles. Control will return to the waiting thread $\lceil \frac{t}{R} \rceil$ times before it can proceed. To simplify the analysis, assume that a switch-spinning thread also has a mean run-length of $\bar{x}$ so that the cost of waiting is increased by $\bar{x} + C$ cycles each time control returns to the waiting thread. (This overestimates the cost since polling the variable should require fewer than $\bar{x}$ cycles. However, we use this approximation for simplicity.) Therefore, the waiting cost of switch-spinning for t cycles is approximately $\lceil \frac{t}{R} \rceil (\bar{x} + C)$ cycles.

We can now approximate β. If t is shorter than R, then $\beta = t/C$. Hence, in this case, switch-spinning is more efficient than spinning if $t > C$. If t is long compared to the R, we can ignore the ceiling operator and obtain $\beta = N$. This is commonly the case in our simulations. Thus switch-spinning amortizes the cost of polling among the N contexts, an intuitively appealing result. Note that if $\beta = 1$, the waiting costs of switch-spinning and spinning are identical.

Switch-Blocking Switch-blocking is a mechanism in which a waiting thread disables the hardware context in which it is resident, in addition to switching to another processor-resident thread. As in blocking, it then places itself on a queue associated with the failed synchronization. Further context switches skip over the disabled context until the context is reenabled. Since there is no need to load and unload threads, switch-blocking is a signalling mechanism with a very low fixed cost. In Alewife, the cost of switch-blocking should be less than 100 cycles. The performance implications of switch-blocking as a signalling mechanism is not studied in this paper.

3 Waiting Algorithms

Waiting algorithms have at their disposal the waiting mechanisms provided by the multiprocessor. It is the responsibility of the waiting algorithm to reduce the cost of waiting by judiciously selecting among these waiting mechanisms. In this study, we will consider spinning, switch-spinning, and blocking as available waiting mechanisms.

A choice of waiting mechanisms has to be made only if there are runnable threads to replace a blocked thread. To facilitate discussion, let us say that a program is *matched* if the number of concurrently runnable threads assigned to any processor never exceeds the number of hardware contexts on that processor; otherwise the program is *unmatched*. Thus, an always-poll algorithm should be used for matched programs since there are no other runnable threads to replace a blocked thread. For multithreaded processors, switch-spinning should be used as the polling mechanism since it has a lower cost than spinning.

However, parallel programs are commonly unmatched or consist of matched and unmatched phases so that a choice between polling and signalling has to be made by the waiting algorithm. This occurs in programs that spawn threads in a data dependent fashion, or when the run-time system dynamically partitions the program, or when the machine is multiprogrammed.

3.1 Single-Phase Algorithms

The simplest algorithms use any one of the waiting mechanisms in isolation, leading to the following algorithms: *always-spin*, *always-switch-spin* and *always-block*.

These single-phase algorithms rely on the correct choice to be made at program creation time. As mentioned in Section 1, this choice depends on the waiting times that will be encountered. Short wait times call for an always-poll algorithm and long wait times call for an always-block algorithm. Deadlock is another factor that affects always-poll algorithms. If the program is unmatched, polling admits the possibility of deadlock if non-preemptive scheduling is used. Although timeouts and preemptive scheduling can be used to avoid deadlock, polling could still suffer from poor performance.

What techniques might be used for making the right choice of single-phase algorithms? The analytical results in Section 4 and the measurements in Section 6 suggest that program wait-time profiles and knowledge about the behavior of different synchronization types are possible candidates. Wait-time profiles are useful if they are good indicators of wait times of future executions even with possibly different run-time conditions. Different synchronization types have different expected wait times, *e.g.*, barrier wait times are usually longer than mutual-exclusion lock wait times. Lastly, compiler analysis could be used to estimate waiting times, *e.g.*, in software combining tree barriers, waits near or at the root of the tree are likely to be shorter.

3.2 Two-Phase Algorithms

The problem with single-phase algorithms is that an improper choice of these algorithms can lead to poor performance and even to deadlock. In a study on the effect of data dependence and multiprogramming on expected wait times, Zahorjan *et al.* [32] showed that wait times can be highly dependent on run-time factors. They concluded that both sources of run-time uncertainty can lead to sharply increased wait times in the case of barrier synchronization. Because uncertainty is even more prevalent in large-scale machines, it is imperative that a waiting algorithm perform well in the face of uncertain wait times.

When the choice between polling and signalling cannot be made reliably at program creation time or compile time, a two-phase waiting algorithm can be used to make a run-time choice by splitting the wait between a polling phase and a subsequent signalling phase. A waiting thread polls until the cost of polling exceeds L_{poll}, after which a signalling mechanism is invoked. This limit, defined as the *polling cost limit*, is most naturally expressed as some multiple, α, of the cost of blocking, B. In other words, we represent

$$L_{poll} = \alpha B$$

If switch-spinning is used as the polling mechanism, the maximum *length* (in processor cycles) of the polling phase is $\beta L_{poll} = \beta\alpha B$. The additional factor of β cycles that a switch-spinning thread can afford to spend polling, given L_{poll}, is due to the higher efficiency of switch-spinning over spinning.

L_{poll} is an adjustable parameter of the two-phase algorithm and is an important determinant of its performance. There are several methods for choosing it. *Static* two-phase waiting algorithms fix L_{poll} at program creation time. *Randomized* two-phase waiting algorithms randomly pick L_{poll} from a predetermined probability distribution for each wait at run-time. *Adaptive* two-phase waiting algorithms dynamically maintain histories of wait times to help decide L_{poll}.

3.2.1 Static Two-Phase Algorithms

This paper focuses on static two-phase algorithms because they have the lowest run-time overheads, and because our theoretical results demonstrate that they can approach optimal on-line performance with an informed static choice of L_{poll}. Randomized and adaptive algorithms also approach optimal on-line performance, but incur higher run-time overheads. Adaptive algorithms incur substantial run-time overheads in large-scale machines with fine-grained synchronization because they need to maintain histories of wait times at each of the many synchronization locations. Furthermore, they are not suitable for single-assignment synchronization types like I-structures [5] because of the absence of wait time histories for the synchronization locations.

In using static two-phase algorithms, we have transformed the problem of choosing between spinning and blocking to the problem of deciding the appropriate value for L_{poll}. The advantage gained in using two-phase algorithms, however, is that the choice of L_{poll} is not as critical to performance as the choice between always-spin and always-block: the worst-case performance of two-phase algorithms can be bounded by a constant.

The same techniques described in Section 3.1 for choosing the correct single-phase algorithm can also be used for deciding L_{poll} in two-phase algorithms. If wait-time profiles of previous program runs are indicative of future wait times, then the profiles can be used off-line to determine the best setting for L_{poll} for future program runs.

The method explored in this paper is to choose L_{poll} based on the knowledge of likely wait-time distributions for different synchronization types. In practice, we expect each synchronization type to follow a characteristic wait-time distribution. Wait-time distributions for various synchonization types are derived in Appendix A. Briefly, if we assume Poisson arrivals of synchronizing threads, then wait times for producer-consumer synchronization are *exponentially distributed.* Also, modeling a mutual-exclusion lock (mutex) as an $M/M/1//M$ queue, wait times for mutexes can also be approximated by an exponential distribution when contention for the locks is low. In contrast, the wait-time distribution for barrier synchronization approaches a *uniform distribution* under Poisson arrivals of barrier participants. Because synchronization types are known at compile-time, we can easily implement this method in a compiler to guide its static choice of L_{poll}.

Section 4 will analyze the behavior of two-phase algorithms under exponentially distributed and uniformly distributed wait times and prescribe static values for L_{poll} that allow the algorithm to approach optimal performance.

3.3 A Nomenclature for Two-Phase Algorithms

We propose the following nomenclature for two-phase waiting algorithms. Each name consists of three components: `<phase1>/<phase2>/<phase1 cost limit>`, where

`<phase1>` is the waiting mechanism used during the first phase, and is either Ss for switch-spinning or s for spinning.

`<phase2>` is the waiting mechanism used during the second phase, and is b for blocking.

`<phase1 cost limit>` specifies the cost limit of the first phase as a multiple α of the waiting cost of the second phase. In other words, if this component is α, and blocking is used for the second phase, then L_{poll} is αB. For the optimal off-line algorithm, *Opt* is used instead.

When the value of a component is irrelevant to the algorithm, it is omitted. Under this nomenclature, the following abbreviations name the waiting algorithms considered in this paper.

$s//\infty$	– always-spin.
$Ss//\infty$	– always-switch-spin.
$/b/0$	– always-block.
$s/b/\alpha$	– two-phase spin/block with $L_{poll} = \alpha B$.
$Ss/b/\alpha$	– two-phase switch-spin/block with $L_{poll} = \alpha B$.
$Ss/b/Opt$	– optimal off-line using switch-spinning and blocking.

4 Analysis of Static Two-Phase Waiting Algorithms

In this section, we will model wait times and compare the waiting costs of various waiting algorithms under those models. After briefly reviewing competitive analysis, we derive waiting costs as a function of wait-time distributions and the cost of the constituent waiting mechanisms. We then analyze the expected performance of the waiting algorithms and derive optimal values for L_{poll} under exponential and uniform wait-time distributions. We show that two-phase waiting performs robustly under exponentially distributed wait times. That is, its performance is never worse than either always-block or always-spin used exclusively. Our analysis assumes that we can always find a runnable thread to replace a blocked thread.

4.1 Competitive Analysis

We use competitive analysis to characterize the performance of the algorithms. A *c-competitive* algorithm has a cost that is *at most c* times the cost of an *optimal off-line* algorithm plus a fixed constant term, given a fixed input sequence. c is termed the *competitive factor*. An optimal off-line algorithm has complete *a priori* knowledge of wait times and is thus able to choose the correct waiting mechanism at all times. An on-line algorithm is *strongly competitive*, and thus optimal, if it possesses the smallest possible competitive factor.

The cost of a waiting algorithm depends on the sequence of wait times presented to it by an adversary. Using terminology in [19], a *strong adversary* is one that chooses wait times in response to previous choices of L_{poll} by the waiting algorithm. A *weak adversary* is one that chooses wait times without considering previous choices of L_{poll} by the waiting algorithm.

We can easily achieve a 2-competitive waiting algorithm against a strong adversary by setting $L_{poll} = B$. When $L_{poll} = B$, the worst possible scenario is to block after polling, incurring a cost of $2B$, when the optimal off-line algorithm would have blocked immediately, incurring a cost B. If we weaken the adversary and consider expected costs we can achieve lower competitive factors. In [19], Karlin *et al.* present a dynamic, randomized two-phase waiting algorithm with an expected competitive factor of $e/(e-1) \approx 1.58$ and prove this factor to be optimal for on-line algorithms against a weak adversary.

We show in this section that if we further weaken the adversary by fixing the wait time distribution and allowing it to control only the parameters of the distribution, *static* two-phase waiting can attain or approach Karlin *et al.*'s optimal competitive factor of $e/(e-1)$. Let us call such an adversary a *restricted adversary*. We show that with exponentially or uniformly distributed wait times, a static choice of L_{poll} yields an efficient algorithm against a restricted adversary. We further show that with exponentially distributed wait times, the static algorithm with $L_{poll} = \ln(e-1)$ performs as well as any dynamic algorithm against a restricted adversary.

4.2 Expected Waiting Costs

In the following analyses, we let $f(t)$ be the probability density function (PDF) of wait times. $f(t)$ is nonzero only for $t \geq 0$. As previously defined, L_{poll} is expressed as a multiple α of the cost of blocking B. The cost of algorithm a is denoted C_a, and its expected cost is denoted $E[C_a]$.

The following equation is the expected waiting cost for static two-phase waiting algorithms, where switch-spinning is used for the first phase and blocking for the second. For spinning, simply set $\beta = 1$.

$$E[C_{Ss/b/\alpha}] = \int_0^{\alpha\beta B} \frac{t}{\beta} f(t)dt + \int_{\alpha\beta B}^{\infty} (1+\alpha)Bf(t)dt \tag{1}$$

The first integral is the contribution to the expected waiting cost due to the probability that wait times are less than $\alpha\beta B$ cycles. In this case, the waiting cost is simply the cost of switch-spinning, t/β. The second integral corresponds to the probability that the wait time is more than $\alpha\beta B$ cycles, where the waiting cost is L_{poll} plus B. $E[C_{s/b/\alpha}]$ is easily derived by setting β to 1 in equation 1. $E[C_{Ss//\infty}]$ is derived by setting α to ∞, and $E[C_{/b/0}]$ is derived by setting α to 0.

The following equation is the expected cost of an optimal off-line algorithm that uses switch-spinning and blocking, and is derived by observing that the optimal algorithm polls if $t \leq \beta B$, and blocks otherwise.

$$E[C_{Ss/b/Opt}] = \int_0^{\beta B} \frac{t}{\beta} f(t)dt + \int_{\beta B}^{\infty} Bf(t)dt \tag{2}$$

4.3 Wait Time Distributions

The expected cost of a two-phase algorithm depends on the wait-time distribution, $f(t)$, as described in Equation 1. We will analyze the cost of static two-phase waiting, using the exponential and uniform distributions as simple models of wait time distributions. Appendix A presents empirical measurements of wait-time distributions that show that exponential and uniform distributions are reasonable models. As future work, other wait-time distributions can be proposed and waiting costs analyzed using the same framework.

Under common patterns of usage, we expect each synchronization type to have its own characteristic wait-time distribution. We consider three types of synchronization: producer-consumer, barrier, and mutual exclusion.

Producer-consumer synchronization is performed between *one* producer and *one or more* consumers of the data produced.[2] Examples of this type of synchronization include `futures` [13] and I-structures [5].

Barrier synchronization ensures that all threads participating in a barrier have reached a point in a program before proceeding.

Mutual-exclusion synchronization is used to provide exclusive access to data structures and critical sections of code.

As motivation for the use of exponential and uniform wait-time distributions for these synchronization types, we model the arrival of threads at synchronization points as generated by a Poisson process and derive wait time models that are approximately exponential and uniform. The Poisson assumption has been a useful approximation of the behavior of many complex systems, and is usually necessary to make analysis tractable.

If we assume Poisson arrivals of producer threads, it follows that wait times for producer-consumer synchronization are exponentially distributed. Appendix A derives wait-time models for barrier and mutual-exclusion synchronization. The models indicate that wait times for barrier synchronization can be approximated by a uniform distribution. Wait times at mutexes can be modeled by either an exponential or uniform distribution, depending on the distribution of lock-holding times.

4.4 Theoretical Analysis of Waiting Costs

We now have the equations necessary to compute the expected waiting costs of static two-phase waiting algorithms. We will consider each synchronization type separately. For exponentially distributed wait times, we will prove that

1. The performance of two-phase waiting always lies in between those of always-block and always-spin.

[2]This is different from another common form of producer-consumer synchronization where multiple producers and consumers insert and remove items from a buffer, and each producer produces an item for a single consumer. The model for mutual-exclusion synchronization can be used to model waiting times for this form of producer-consumer synchronization.

2. When a restricted adversary chooses λ (the arrival rate), the competitive factor of static two-phase waiting has a lower bound of $e/(e-1)$. Furthermore, no dynamic algorithm can attain a lower competitive factor. Recall that this competitive factor is also optimal for on-line algorithms against weak adversaries.

3. A static value of $\ln(e-1)B$ for L_{poll} results in an algorithm that attains this lower bound of $e/(e-1)$ against a restricted adversary.

We also state here and prove in Appendix B that for uniformly distributed wait times,

1. When a restricted adversary gets chooses the parameter of the uniform distribution, the competitive factor of static two-phase waiting has a lower bound of $(\sqrt{5}+1)/2$ against this adversary.

2. A static value of $\frac{1}{2}(\sqrt{5}-1)B$ for L_{poll} results in an algorithm that attains this lower bound of $(\sqrt{5}+1)/2$ against a restricted adversary. Furthermore, no other static choice of L_{poll} attains this competitive factor.

4.4.1 Producer-Consumer Synchronization

We model producer-consumer wait times distributions as exponential, *i.e.*,

$$f(t) = \lambda e^{-\lambda t} \tag{3}$$

where λ is the arrival rate of the producer.

From Equations 1–3, we derive the following expressions for the expected costs of always-switch-spin ($Ss//\infty$), always-block ($/b/0$), static two-phase ($Ss/b/\alpha$), and optimal off-line ($Ss/b/Opt$). If spinning is used instead of switch-spinning, simply set $\beta = 1$.

$$E[C_{Ss//\infty}] = \int_0^\infty \frac{t}{\beta}\lambda e^{-\lambda t}dt = \frac{1}{\lambda\beta} \tag{4}$$

$$E[C_{/b/0}] = B \tag{5}$$

$$\begin{aligned} E[C_{Ss/b/\alpha}] &= \int_0^{\alpha\beta B} \frac{t}{\beta}\lambda e^{-\lambda t}dt + \int_{\alpha\beta B}^\infty (1+\alpha)B\lambda e^{-\lambda t}dt \\ &= \frac{1}{\lambda\beta}(1-e^{-\lambda\alpha\beta B}) + Be^{-\lambda\alpha\beta B} \end{aligned} \tag{6}$$

$$\begin{aligned} E[C_{Ss/b/Opt}] &= \int_0^{\beta B} \frac{t}{\beta}\lambda e^{-\lambda t}dt + \int_{\beta B}^\infty B\lambda e^{-\lambda t}dt \\ &= \frac{1}{\lambda\beta}(1-e^{-\lambda\beta B}) \end{aligned} \tag{7}$$

Comparing the expected performance of $Ss//\infty$, $/b/0$ and $Ss/b/\alpha$ yields an interesting result. We expect that when arrival rates are high, $Ss//\infty$ will perform better than $/b/0$, conversely, when arrival rates are low, $/b/0$ will perform better than $Ss//\infty$. The equations show that regardless of the arrival rate and L_{poll} the expected performance of static two-phase algorithms always falls in between the performance of $Ss//\infty$ and $/b/0$. More formally,

Theorem 1 *Under exponentially distributed wait times, the expected costs of the algorithms $Ss//\infty$, $/b/0$, and $Ss/b/\alpha$ are ordered as*

$$E[C_{/b/0}] \leq E[C_{Ss/b/\alpha}] \leq E[C_{Ss//\infty}] \quad \text{if} \quad \lambda\beta B \leq 1$$
$$E[C_{/b/0}] \geq E[C_{Ss/b/\alpha}] \geq E[C_{Ss//\infty}] \quad \text{if} \quad \lambda\beta B \geq 1$$

Proof: By inspection, $E[C_{/b/0}] \geq E[C_{Ss//\infty}]$ when $\lambda\beta B \geq 1$ and $E[C_{/b/0}] \leq E[C_{Ss//\infty}]$ when $\lambda\beta B \leq 1$. Comparing $E[C_{/b/0}]$ with $E[C_{Ss/b/\alpha}]$ yields

$$\begin{aligned} E[C_{Ss/b/\alpha}] \leq E[C_{/b/0}] &\Leftrightarrow \frac{1}{\lambda\beta}(1 - e^{-\lambda\alpha\beta B}) + Be^{-\lambda\alpha\beta B} \leq B \\ &\Leftrightarrow \lambda\beta B \geq 1. \end{aligned}$$

Comparing $E[C_{Ss//\infty}]$ with $E[C_{Ss/b/\alpha}]$ yields

$$\begin{aligned} E[C_{Ss/b/\alpha}] \leq E[C_{Ss//\infty}] &\Leftrightarrow \frac{1}{\lambda\beta}(1 - e^{-\lambda\alpha\beta B}) + Be^{-\lambda\alpha\beta B} \leq \frac{1}{\lambda\beta} \\ &\Leftrightarrow \lambda\beta B \leq 1 \end{aligned}$$

□

Empirical measurements (see Section 6) further indicate that two-phase algorithms are remarkably robust, and their performance is usually close to the better of $Ss//\infty$ and $/b/0$.

Next we observe that when $\lambda\beta B = 1$, the costs of all three algorithms are equal to B. That is, at this breakeven point where the arrival rate $\lambda = 1/\beta B$, the choice of L_{poll} does not affect the expected cost of the two-phase algorithm. More formally,

Theorem 2 *Under exponentially distributed wait times with $\lambda\beta B = 1$, the competitive factor of $E[C_{Ss/b/\alpha}]$ is $e/(e-1)$, regardless of the value of α.*

Proof: When $\lambda\beta B = 1$, we know from Theorem 1 that

$$E[C_{/b/0}] = E[C_{Ss/b/\alpha}] = E[C_{Ss//\infty}] = B$$

Therefore

$$\frac{E[C_{Ss/b/\alpha}]}{E[C_{Ss/b/Opt}]} = \frac{\lambda\beta B}{(1 - e^{-\lambda\beta B})} = \frac{e}{(e-1)}$$

□

This leads to the following corollary:

Corollary 1 *There exists a lower bound of $e/(e-1)$ on the competitive factor of any two-phase algorithm against strong, weak and restricted adversaries.*

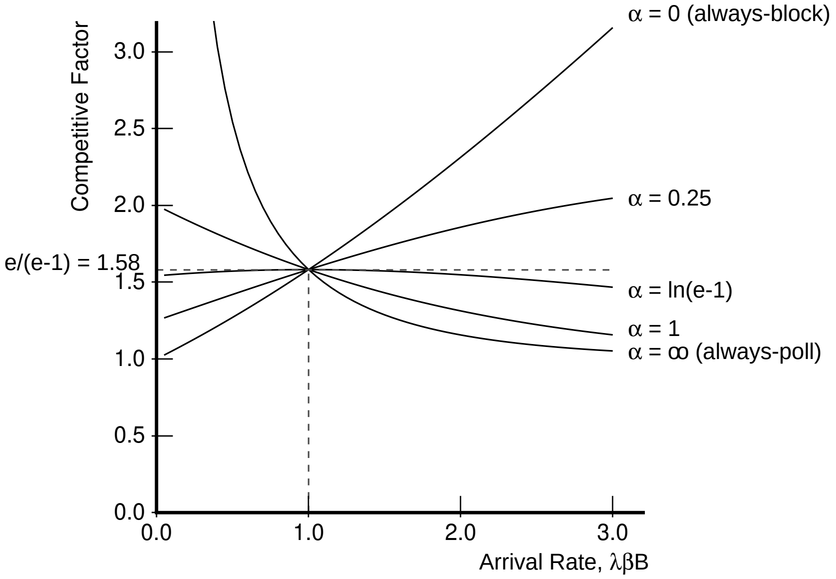

Figure 2: Expected competitive factors under exponentially distributed wait times.

Proof: When the adversary picks exponentially distributed wait times with $\lambda = 1/(\beta B)$, regardless of the choice of α and regardless of whether the choice is made statically or dynamically, Theorem 2 implies the waiting cost will be $e/(e-1)$ times that of an optimal off-line algorithm. It follows that one cannot construct a two-phase algorithm with a competitive factor lower than $e/(e-1)$. This competitive factor matches the lower bound obtained in [19] against a weak adversary. □

In light of this lower bound, the natural question to ask is whether a single static value for α can attain this lower bound under exponentially distributed wait times. Surprisingly, the answer is yes, and the following theorem prescribes a value of α that yields optimal performance for exponentially distributed wait times.

Theorem 3 *Under exponentially distributed wait times with $\alpha = \ln(e-1)$, the competitive factor of two-phase waiting, $E[C_{Ss/b/\alpha}]/E[C_{Ss/b/Opt}]$, is at most $e/(e-1)$, regardless of the arrival rate, λ, of the distribution.*

Proof: Set $\alpha = \ln(e-1)$ in the equation for $E[C_{Ss/b/\alpha}]/E[C_{Ss/b/Opt}]$. This yields an equation for the competitive factor for two-phase waiting as a function of λ. Differentiate this equation with respect to λ to find the maximum. The resulting maximum competitive factor is $e/(e-1)$ at an arrival rate of $\lambda = 1/\beta B$. □

These theorems are best illustrated by Figure 2. The figure plots the competitive factor of static two-phase waiting over an entire range of α and λ. We see that the curves for finite non-zero values of α lie in between those of always switch-spin ($\alpha = \infty$) and always-block ($\alpha = 0$), as indicated by Theorem 1. We can also see that all the curves intersect at a competitive factor of $e/(e-1)$ when $\lambda = 1/\beta B$ as indicated by Theorem 2. Lastly, we can see that the competitive factor is at most $e/(e-1)$ when $\alpha = \ln(e-1)$, as indicated by Theorem 3. Since actual values of λ are not relied upon, this limit holds in the face of run-time uncertainty and feedback effects of the waiting algorithm on the wait time as long as the wait-time distributions remain exponential.

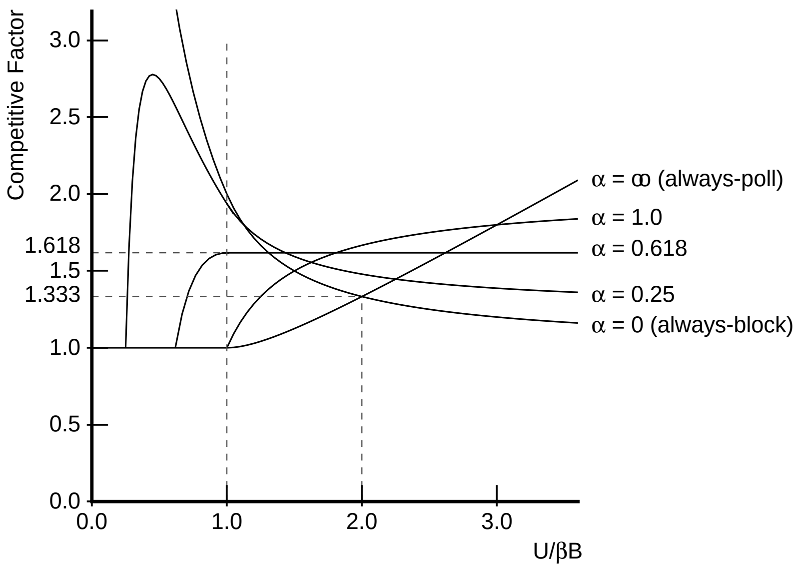

Figure 3: Expected competitive factors under wait times uniformly distributed between 0 and U.

These theorems imply that when wait times are exponential, we should choose our waiting algorithm depending on knowledge of λ. If we know that $\lambda < 1/\beta B$, we should choose $/b/0$, otherwise we should choose $Ss//\infty$. However, if we cannot reliably predict λ, we should choose $Ss/b/0.54$ to obtain the best competitive factor of 1.58.

4.4.2 Barrier Synchronization

A thread arriving at a barrier has to wait for the rest of the participants to arrive. To simplify the analysis, we assume that barrier wait times are uniformly distributed between 0 and U. In Appendix A, we argue that this is a reasonable model for barrier wait times. Repeating the analysis leading to Theorems 1–3 for the case of uniform wait times, the following theorem is proved in Appendix B and illustrated in Figure 3.

Theorem 4 *Under uniformly distributed wait times from $t = 0$ to U, with $\alpha = (\sqrt{5}-1)/2 \approx 0.618$, the competitive factor of two-phase waiting, $E[C_{Ss/b/\alpha}]/E[C_{Ss/b/Opt}]$, is at most $(\sqrt{5}+1)/2 \approx 1.618$, regardless of the parameter, U, of the distribution. Furthermore, if $\alpha \neq (\sqrt{5}-1)/2$, then the competitive factor under uniformly distributed wait times is larger than $(\sqrt{5}+1)/2$.*

The results show that we should choose our waiting algorithm using our knowledge of U. If we know that $U > 2\beta B$, we should choose $/b/0$, otherwise we should choose $Ss//\infty$. Therefore, with accurate information about U the competitive factor is at most 4/3 as shown in Figure 3. However, as observed by Zahorjan *et al.* [32], barrier wait times are highly dependent on run-time factors making it hard to predict U. If we cannot reliably predict U, we should choose $Ss/b/0.62$ to obtain the best competitive factor of 1.62 (the golden ratio), as prescribed by Theorem 4, and as illustrated in Figure 3. This is very close to the optimal on-line competitive factor of 1.58 against weak adversaries.

4.4.3 Mutual-Exclusion Locks

The wait time of a lock requester depends on the distribution of lock-holding times, and on whether the mutex enforces some queueing discipline. We first consider the case where there is no queueing discipline and waiters contend for the lock as soon as it is released. If lock-holding times are exponential, then the wait time for a lock is also exponential and the analysis for exponential wait times would apply.

If lock-holding times are fixed and deterministic, we have to differentiate between *new* waiters and *repeat* waiters. New waiters are requesters that are fresh arrivals, and repeat waiters are requesters that have recontended unsucessfully for the lock. Assuming reasonable lock contention, in a large-scale system with a large number of potential lock requesters, we can assume that the arrival of new waiters is Poisson. Thus the wait time for new waiters will be uniformly distributed between 0 and the fixed lock-holding time, and the previous analysis for uniformly distributed wait times would apply. The wait time for repeat waiters is simply the fixed lock-holding time and it should be straightforward to decide whether to block repeat waiters.

Next, we consider the case where the mutex enforces some queueing discipline. In addition to the lock holder, a waiter also has to wait for other waiters ahead of it in the queue. In Appendix A, we propose a wait-time model based on $M/M/1//M$ queues to predict the steady-state queue lengths and derive the wait-time distributions. Unfortunately the model is sufficiently complex that it does not lend itself to straightforward analysis, and we have not succeeded in arriving at a simple approximation of the wait-time model as we did for barriers. We therefore leave a rigorous analysis of this model for future research.

With some programmer/compiler effort, we could also use wait-time profiles to determine the optimal choice of L_{poll}. Zahorjan *et al.* show that lock wait times are not significantly affected by run-time factors if lock holders are never descheduled. This allows wait-time profiles to be accurate predictors of future wait times. In [18], Karlin *et al.* found that a static two-phase algorithm, with L_{poll} based on wait-time profiles of mutual-exclusion synchronization, had the best performance among the algorithms they considered.

5 Experimental Framework

To show that static two-phase waiting algorithms work well in practice and to corroborate the analysis of the previous section, we profiled the executions of several benchmark programs using various synchronization types on a simulator of the Alewife machine. This section overviews relevant features of the Alewife system and gives a brief description of the programs.

Alewife is a distributed-memory multiprocessor that supports the shared-memory programming abstraction. An Alewife node consists of a processor, a cache, a portion of distributed, globally-shared memory, a memory controller, a floating-point coprocessor, and a network router. Nodes are connected via a two-dimensional mesh network. The memory controller synthesizes a globally-shared address space and maintains cache coherence using the LimitLESS directory protocol [9].

A description of Alewife's processor, Sparcle, can be found in [2]. Sparcle is designed to meet several requirements significant to multiprocessing: it tolerates latencies through block multithreading, and it handles traps efficiently through a rapid-trap-dispatch mechanism.

Sparcle has four hardware contexts so that multiple threads co-reside on a single processor. Consequently, switching processor control among the processor-resident threads is accomplished rapidly. The experiments assume that context switches can be achieved in 12 cycles. With a dedicated synchronization trap line, the trap mechanism can pass control to the appropriate trap handler in 5 cycles. By default, context switching occurs in a round-robin fashion so that control eventually returns to each resident thread.

5.1 Hardware Support for Synchronization

Alewife's primitive hardware mechanisms for synchronization are *full/empty bits* and efficient traps. As in the HEP[27], a full/empty bit is associated with each memory word and atomic read/modify/write operations can be performed on the full/empty bit. `read-and-empty` atomically reads a memory word and simultaneously resets the associated full/empty bit. A *full/empty trap* is generated by the memory controller and communicated to Sparcle if the word was previously reset. Conversely, `write-and-fill` atomically writes a memory word and sets the associated full/empty bit. A full/empty trap is also generated if the word was already set.

A synchronization attempt *succeeds* if it does not generate a trap, and *fails* otherwise. Full/empty bits allow efficient implementation of many high level and fine-grain synchronization constructs. Because failures are signalled via traps, successful synchronizations are overhead-free.

Failed synchronizations rely on a trap handler to implement the waiting algorithm. The full/empty trap handler must determine the synchronization type that caused the trap in order to take appropriate action. In Alewife, the compiler communicates compile-time information about the synchronization operation to the trap handler by using otherwise unused bits in the machine instruction. The trap handler in Alewife examines these bits in the trapping instruction and thus implements efficient multiplexing of multiple traps on a single hardware trap signal.

5.2 Software Synchronization Constructs

The following synchronization constructs used in our benchmarks are supported by Alewife's software system and rely on hardware full/empty bits and traps for efficiency.

J-structures (Reusable I-structures) A J-structure is a data structure for producer-consumer-style synchronization on vector elements which enables efficient fine-grain data-parallel computation. It is implemented as a vector with full/empty bits associated with each vector slot. A slot is considered full if its full/empty bit is `1` and empty otherwise. A reader of a J-structure slot waits until the slot is full before returning the value. A writer of a J-structure slot writes a value to the slot, sets it to full, and releases all waiters for the slot. An empty vector slot doubles as the queue pointer for waiting readers. An error is signalled if a write is attempted on a full slot. J-structures can be used to implement I-structure [5] semantics.

We allow a J-structure slots to be *reset*. A reset empties the slot, permitting multiple assignments. Reusing J-structure slots in this way allows efficient cache performance. However, depending on the application, the programmer is responsible for proper synchronization between readers and resetters of a slot.

L-structures (Lock-able structures) Like J-structures, an L-structure is implemented as a vector with full/empty bits associated with each vector slot. L-structures support 3 operations: a locking read, an unlocking write, and a non-locking read. A locking read waits until a slot is full before emptying the slot and returning the value. An unlocking write writes a value to an empty slot, and sets it to full, releasing any waiters. It is an error to perform an unlocking write to a full slot. A non-locking read returns the value found in a slot if full; otherwise it returns an invalid value.

An L-structure therefore allows mutually exclusive access to each of its slots. The locking and unlocking L-structure reads and writes are sufficient to implement M-structures [7]. L-structures are different from M-structures in that they allow multiple non-locking readers.

Semaphores A semaphore is implemented as a one-element L-structure. `semaphore-p` and `semaphore-v` are easily implemented using L-structure reads and writes. Semaphores are used to implement mutual-exclusion.

Futures Futures specify parallelism in Multilisp [13] and synchronization on the return values of the threads. It is a form of producer-consumer synchronization. The future object is simply a memory word that initially holds the queue of waiting consumers and eventually holds the value of the future when resolved.

Barriers Barriers ensure that all participating threads have reached a point in a program before proceeding. To avoid excessive traffic to a single location, and to distribute the enqueuing and release operations, we use software combining trees [31] to implement barriers.

5.3 Simulation Environment

While the implementation of the Alewife machine is in progress, a cycle-by-cycle simulator called ASIM is being used for software and applications development. ASIM faithfully simulates the complete machine. A compiler and run-time system are also operational and allow us to execute and profile parallel programs. Table 1 lists the default parameters used in the simulations.

Since we are simulating a 64-processor machine on a uniprocessor, we are naturally constrained on the length of simulations we can support. However, to ensure that we are not measuring transient effects, we simulate the programs for sufficiently long periods of time so that a significant fraction of this time is spent in steady state execution.

We collected several statistics from the simulations. *Wait-time profiles* are a record of wait times encountered for each failed synchronization. For mutex waiters, the wait time measured from the first failed request to the time the mutex is *successfully acquired*.[3]

We measured the total number of cycles consumed by the blocking routines. For an always-block waiting algorithm these cycles constitute the waiting cost incurred in the program run. We provide this statistic as *wait overhead* for $/b/0$ in the results section. This figure is useful in estimating the potential effect of a waiting algorithm on the running time of a benchmark, and allows us to speculate on the performance of waiting algorithms on larger machines where wait overheads are expected to be more significant.

We also keep count of the total *number of threads blocked* during the execution of the program. We expect a two-phase algorithm to reduce the number of blocked threads, giving us some insight on how well the two-phase algorithm is performing relative to an always-block algorithm. *Program execution time* measurements are also compared because it is the ultimate performance metric.

[3]This is a different measure from that used in [18], where mutex wait time was measured from the first failed request to the time the mutex is released by the holder.

Parameter	Default setting
Number of Processors	64
Cache Coherence Protocol	LimitLESS[1]
Cache Size	64KB (4096 blocks)
Cache Block Size	16 bytes
Network Topology	2-D Mesh
Network Channel Width	1 byte

[1]using 4 hardware pointers

Table 1: Simulation Parameters.

5.4 Benchmarks

Our simulations used the following benchmarks representative of producer-consumer, barrier, and mutual-exclusion synchronization. A more detailed description of these benchmarks can be found in [21].

Producer-Consumer

MGrid applies the multigrid algorithm to solving Poisson's equation on a 2-D grid. Communication is nearest-neighbor except during shrink and expand phases. The 2-D grid is partitioned into subgrids, and a thread is assigned to each subgrid. Borders of each subgrid are implemented as J-structures to allow fine grain synchronization with neighbors. The J-structures are reset between iterations.

Jacobi performs Jacobi relaxation for solving Poisson's equation on a 2-D grid. Each thread is responsible for one grid point, and neighboring grid points are mapped onto neighboring processors. The grid is allocated uniformly so that only nearest-neighbor communication is necessary. J-structures are used to synchronize neighboring threads. The grain size of each thread is purposely made very small so that we can see the effects of synchronization as they become significant.

Factor Given a range of integers, **Factor** computes the largest prime factors of each integer and accumulates them. The synchronization structure of the program can be most easily viewed as a recursive function call tree with synchronizations occurring at each node of the tree. The program was dynamically partitioned with lazy task creation [24].

Queens solves the n-queens problem: given an $n \times n$ chess board, place n queens such that no two queens are on the same row, column, or diagonal. A search of all possible solutions is made and this particular benchmark was run with $n = 9$ and with lazy task creation. **Queens** has similar synchronization characteristics to **Factor**.

Barrier

CGrad is the conjugate gradient numerical algorithm for solving systems of linear equations. In this benchmark, the algorithm is used to solve Poisson's equation on a 2-D grid. Each iteration of **CGrad** involves global accumulates and broadcasts which are implemented using a software combining tree. These accumulates and broadcasts also serve as barriers between phases.

Jacobi-Bar solves exactly the same problem as **Jacobi**, but with a global barrier synchronization between iterations. Like in **Jacobi**, only nearest neighbor communication is necessary within an iteration.

Mutual Exclusion

CountNet tests an implementation of a counting network [6]. Threads repeatedly try to increment the value of a counter through a bitonic counting network so as to reduce contention and allow parallelism. Threads acquire and release mutexes at each network node as they traverse the network.

FibHeap tests an implementation of a scalable priority queue based on a Fibonacci heap [15]. Mutexes are used to ensure atomic updates to the heap and scalability is achieved by distributing mutexes throughout the data structure to avoid points of high lock contention and allow parallelism. The test involves repeatedly executing `insert` and `extract-min` operations on the priority queue.

Mutex is a synthetic benchmark to monitor the performance of mutexes under varying loads. Worker threads are distributed evenly throughout the machine and each thread runs a loop that with some fixed probability acquires a mutex, executes a critical section, then releases the mutex.

6 Results and Analysis

Let us begin by summarizing some of our major theoretical results and premises that will be validated in this section. The theoretical analysis of Section 4 predicts that different synchronization types should have different wait-time characteristics. It also predicts that the performance of two-phase waiting should be robust since its cost can be bounded. Moreover, for exponential wait time distributions, a static setting of $L_{poll} = 0.54B$ is optimal, while for uniform distributions, a static setting of $L_{poll} = 0.62B$ is optimal. Finally, our motivation for advocating *static* two-phase algorithms is based on the premise that the cost of blocking can be reduced to a point comparable to the overhead of making dynamic choices for L_{poll}.

To see if these results bear out in practice, we turn to empirical measurements of programs executing on ASIM. The wait-time profiles in Appendix A show that each of the three synchronization types considered have different distributions, suggesting different waiting strategies for each. The program execution statistics presented in this section show that two-phase waiting is extremely robust and performs close to the best across all the benchmarks and never results in pathologically bad performance.

While we would like to empirically confirm that the prescribed settings for L_{poll} of $0.54B$ for exponentially distributed wait times and $0.62B$ for uniformly distributed times lead to optimal competitive factors, doing so would require an infeasible amount of simulation. We would have to run a large set of benchmarks exhibiting a wide range of values of the wait-time distribution parameters because the optimality results apply to expected performance over the entire parameter range of a probability distribution. However, we attempt to lend some credence to the theoretical results by taking some point measurements for several benchmarks with $L_{poll} = 0.5B$.

We start by comparing the performance of two-phase waiting with $L_{poll} = B$ ($Ss/b/1$), always-switch-spin ($Ss//\infty$), and always-block ($/b/0$). (For $Ss//\infty$, L_{poll} is actually limited to 50000 cycles to implement a timeout mechanism for deadlock avoidance.) Tables 2–4 present the performance statistics of $Ss/b/1$, $Ss//\infty$ and $/b/0$. In the tables, *normalized runtime* is the running time normalized to the running time of $/b/0$. Section 6.4 explores the effect of changing L_{poll} and Section 6.5 investigates ways of reducing the cost of blocking.

From the tables, we see that the choice of waiting algorithm can make a substantial difference in the running times of the benchmarks. Because of the need to implement timeouts for deadlock avoidance, using $Ss//\infty$ can make the program run arbitrarily slowly, depending on the deadlock timeout interval. Even ignoring the cases with deadlock, the choice of waiting algorithm can result in nearly 240% difference between the best and worst running times.

Despite the wide variance in running times, $Ss/b/1$ never results in running times more than 53% over the best algorithm. If we ignore the matched program runs, where blocking is not beneficial, the worst running time under $Ss/b/1$ is actually a mere 6.6% over the best algorithm. This is strong evidence of the robustness of two-phase waiting as predicted by theoretical analysis.

Let us now consider the results for each of the synchronization types separately.

6.1 Producer-Consumer Synchronization

The simulation results for producer-consumer synchronization are summarized in Table 2. Since wait-time profiles for producer-consumer synchronization approximate an exponential distribution (see Appendix A), we expect the performance of $Ss/b/1$ to lie in between $/b/0$ and $Ss//\infty$ (see Theorem 1). This is indeed the case[4], but more importantly, the measured performance of $Ss/b/1$ is not far from the best algorithm in each case. $Ss/b/1$ has the best overall performance among the three waiting algorithms.

$Ss//\infty$ encounters deadlock and times out in unmatched **MGrid** and **Jacobi** and thus performs poorly. This problem with deadlock is not present for unmatched **Queens** and **Factor** because they are dynamically partitioned with lazy task creation [24]. $/b/0$ performs reasonably well except for matched **Jacobi** which has very short wait times.

6.2 Barrier Synchronization

Because of their nature, wait times at barriers are likely to be long: a waiting thread is likely to be held up for a large number of other threads, especially in large-scale machines. For our benchmarks, the wait-time profiles in Appendix A indicate that most of the wait times were longer than the blocking overhead. We see the effect of this in the performance figures in Table 3, where $/b/0$ performs best in the unmatched programs. The number of blocked tasks also confirm that most of the wait times are longer than B. This suggests that we should use $/b/0$ at barriers unless we know that the program is matched. $Ss//\infty$ runs into deadlock for the unmatched programs.

Nevertheless, $Ss/b/1$ performs quite well and is not more than 6.6% off from $/b/0$. We can do even better if we have some indication of the number of arrivals at the barrier. We can't rely on the availability of a global count of arrivals in large-scale machines because that would limit the scalability of the barrier algorithm. However, for tournament-style tree barriers, we know that waits near the root of the tree should be shorter than waits near the leaves. Accordingly, we should use an always-block algorithm for the lower sections of the tree and a two-phase algorithm for the upper sections.

[4] $Ss/b/1$ performs best in **Queens** because of an interaction with the scheduler and lazy task creation which resulted in a better partitioning of the program.

Benchmark	Contexts	Waiting Algorithm	Runtime (Kcycles)	Normalized Runtime	Wait Ovh.[1]	Blocked Threads
MGrid matched	4	$/b/0$	2,251	1.0	5%	6,365
	4	$Ss/b/1$	1,918	0.85		1,769
	4	$Ss//\infty$	1,731	0.77		4
MGrid unmatched	2	$/b/0$	1,865	1.0	7%	6,953
	2	$Ss/b/1$	1,885	1.01		5,150
	2	$Ss//\infty$	7,273	3.90		4,613
Jacobi matched	4	$/b/0$	930	1.0	21%	12,818
	4	$Ss/b/1$	524	0.56		1,144
	4	$Ss//\infty$	390	0.42		440
Jacobi unmatched	4	$/b/0$	719	1.0	21%	9,931
	4	$Ss/b/1$	757	1.05		7,756
	4	$Ss//\infty$	6,075	8.45		4,399
Queens unmatched	4	$/b/0$	458	1.0	3%	655
	4	$Ss/b/1$	434	0.95		300
	4	$Ss//\infty$	467	1.02		0
Factor unmatched	4	$/b/0$	769	1.0	5%	1,561
	4	$Ss/b/1$	790	1.03		913
	4	$Ss//\infty$	841	1.09		19

[1]as percentage of runtime

Table 2: Performance figures for producer-consumer synchronization.

Benchmark	Contexts	Waiting Algorithm	Runtime (Kcycles)	Normalized Runtime	Wait Ovh.[1]	Blocked Threads
CGrad matched	4	$/b/0$	1,052	1.0	11%	7,478
	4	$Ss/b/1$	999	0.95		7,161
	4	$Ss//\infty$	654	0.62		2
CGrad unmatched	2	$/b/0$	1,048	1.0	11%	7,625
	2	$Ss/b/1$	1,118	1.07		7,309
	2	$Ss//\infty$	3,905	3.73		3,714
Jacobi-Bar unmatched	4	$/b/0$	1,592	1.0	23%	26,880
	4	$Ss/b/1$	1,617	1.02		25,820
	4	$Ss//\infty$	3,497	2.20		14,395

[1]as percentage of runtime

Table 3: Performance figures for barrier synchronization.

Benchmark	Contexts	Waiting Algorithm	Runtime (Kcycles)	Normalized Runtime	Wait Ovh.[1]	Blocked Threads
CountNet matched	4	$/b/0$	1,378	1.0	9%	10,502
	4	$Ss/b/1$	1,293	0.94		1,913
	4	$Ss//\infty$	1,242	0.90		276
CountNet unmatched	2	$/b/0$	1,298	1.0	8%	7,646
	2	$Ss/b/1$	1,241	0.95		1,202
	2	$Ss//\infty$	1,224	0.94		43
FibHeap matched	4	$/b/0$	2,430	1.0	23%	7,882
	4	$Ss/b/1$	2,117	0.87		7,332
	4	$Ss//\infty$	2,617	1.08		581
Mutex unmatched	4	$/b/0$	612	1.0	19%	4.429
	4	$Ss/b/1$	583	0.95		1,652
	4	$Ss//\infty$	678	1.11		0

[1]as percentage of runtime

Table 4: Performance figures for mutual-exclusion synchronization.

Although our theoretical analysis suggests that barrier wait times are uniformly distributed, the wait-time profiles in Appendix A do not support this hypothesis. This deviation was due to the significant overhead of the software combining tree barrier implementation in Alewife. An experiment to filter out the combining tree overhead was performed and the resulting profile does indeed suggest uniformly distributed wait times. This profile is presented in Figure 8 in Appendix A.

6.3 Mutual Exclusion

In the mutual-exclusion benchmarks, deadlock is not an issue, even in unmatched conditions, because lock holders are never descheduled. $Ss/b/1$ performs well in both matched and unmatched **CountNet** and performs best in **FibHeap** and **Mutex**. This again demonstrates the robustness of two-phase waiting. $Ss//\infty$ unexpectedly performs worst even in matched conditions in **FibHeap**. We will explain these observations here.

Lock contention was low in **CountNet**, and we know that a large number of waits were short from looking at the profiles in Appendix A and by comparing the number of blocked threads for $/b/0$ and $Ss/b/1$. Under such conditions, $Ss//\infty$ performs best and $/b/0$ worst, with $Ss/b/1$ close to $Ss//\infty$. However, since the wait times are not exponential nor uniform, we cannot match these performance results with our theoretical analysis.

Lock contention was high in **FibHeap** and **Mutex**. The bad performance of $Ss//\infty$ in these benchmarks is due to the poor behavior of polling under conditions of high lock contention. Because lock waiters were not queued in the benchmarks, simultaneous release of all polling waiters when a highly contended lock is released causes detrimental hot-spot contention. All the released waiters try to acquire the lock at once, exacerbating the wait times at that lock. Recently published techniques for more efficient spin-waiting on locks can be used to improve the performance of polling for highly contended locks [23]. These include exponential backoff and software queueing of spin waiters.

Benchmark	Con-texts	Waiting Algorithm	Runtime (Kcycles)	Normalized Runtime	Wait Ovh.[1]	Blocked Threads
MGrid unmatched	2	$/b/0$	1,865	1.0	7%	6,953
	2	$Ss/b/0.5$	1,817	0.97		5,488
	2	$Ss/b/1$	1,885	1.01		5,150
	2	$Ss//\infty$	7,273	3.90		4,613
Jacobi unmatched	4	$/b/0$	719	1.0	21%	9,931
	4	$Ss/b/0.5$	699	0.97		8,437
	4	$Ss/b/1$	757	1.05		7,756
	4	$Ss//\infty$	6,075	8.45		4,399

[1]as percentage of runtime

Table 5: Performance figures for $L_{poll} = 0.5B$

Because blocked waiters take longer to be reactivated, $/b/0$ avoids the detrimental effect of bursty lock requests. This allows $/b/0$ to actually perform better than $Ss//\infty$, even in matched **FibHeap**. $Ss/b/1$ works best because it naturally polls on lightly contended locks and blocks on highly contended locks, combining the best of both worlds, an advantage not predicted by the theoretical models.

6.4 Changing L_{poll}

In the results presented above, L_{poll} was set to be equal to the cost of blocking. Theorem 3 indicates that setting L_{poll} to $0.54B$ will yield a more robust algorithm when waiting times are exponential.

We experimented with two of the producer-consumer benchmarks (**MGrid** and **Jacobi**) under unmatched conditions. Table 5 reproduces the results presented earlier, and includes results for $Ss/b/0.5$. We observe that a shorter polling phase results in better performance than $Ss/b/1$ in **MGrid** and **Jacobi** because producer arrival rates were low. Under such conditions, *i.e.*, when $\lambda < 1/\beta B$, our theoretical analysis predicts that $Ss/b/0.5$ will perform better than $Ss/b/1$. In [18], Karlin *et al.* also observed by analyzing measured wait-time profiles that setting L_{poll} to $0.5B$ can result in lower waiting costs.

Surprisingly, $Ss/b/0.5$ also performed better than $/b/0$. We think that this effect is due to the possibility that sometimes there are no runnable threads to execute after a thread is blocked, which violates the assumption made in the theoretical analysis. This would cause $/b/0$ to unnecessarily block more threads compared to two-phase waiting. Another possibility is that the waiting algorithm itself affects the wait times.

We did not consider it useful to measure the effects of changing L_{poll} for the barrier benchmarks because the wait times were dominated by the overhead of the software combining tree such that $/b/0$ would predictably be the best algorithm.

	Action	Instructions	Base Cycles
Unloading	Unload registers	21 stores	63
	Enqueue thread	2 stores 2 loads 7 other	17
	Book-keeping	6 stores 1 load 6 other	26
Reenabling	Lock queue of blocked threads	2 loads 1 store 6 other	13
	Queue on processor ready queue	6 loads 5 stores 12 other	39
Reloading	Reload registers	21 loads	42
	Restore misc. state	1 load 6 other	8
	Book-keeping	1 store 8 other	11
	Total	114	219

Table 6: Breakdown of the cost of blocking in Alewife.

6.5 Reducing the Cost of Blocking

Our focus on static two-phase algorithms is based on the premise that in large-scale multiprocessors with support for fine-grained threads, the cost of blocking will be very small and comparable to the overhead of dynamic two-phase algorithms. In this section, we will analyze blocking costs in Alewife and suggest ways of optimizing it to less than 100 cycles.

Reducing the cost of blocking is also motivated by another factor. We observed that $/b/0$ performs quite well in the benchmarks. Therefore, further reducing the cost of blocking would make $/b/0$ significantly more attractive. The shorter the cost of blocking, the shorter the polling phase of a two-phase algorithm should be. The performance of the $/b/0$ and two-phase algorithms will therefore become more similar.

Table 6 gives a breakdown of the costs of unloading, reenabling, and reloading a thread in terms of instructions and base-cycle times in Alewife (base cycles assume cache hits). In terms of base cycles, the cost of blocking is 219 cycles. However, the measured cost of blocking is experimentally observed to be about 500 cycles because of cache misses. Of the measured cycles, about 300 cycles are spent unloading the task, 100 cycles reenabling it and 65 cycles reloading it. Loads and stores are observed to take 3 times longer than the base-cycle time when unloading a thread due to cache misses. However, since an unloaded thread resides in the cache, reloading a thread takes close to the base-cycle time.

We have not yet fully optimized the cost of blocking in Alewife, but we believe it can be reduced significantly without additional hardware support. Cache misses can be largely avoided by reusing old thread frames for storage of unloaded threads. The number of instructions can also be reduced through careful handcrafting of the relevant portions of the scheduler.

Most of the cycles spent in blocking are due to saving and restoring registers. Since Sparcle loads take two cycles and stores three [28], the cost of blocking is higher than it would be on a processor with single-cycle loads and stores. With single-cycle load/stores, the cost of blocking can be reduced to 114 base cycles. Alternatively, we could pipeline the loads and stores of the registers. It is also possible to avoid saving most of the registers if a thread is unloaded at a procedure call boundary, where the caller has already saved live registers into the stack. We recently implemented this special case in Alewife, reducing the cost of blocking for this special case to 144 base cycles. With single-cycle load/stores, this reduces to 83 base cycles.

Processors with a smaller amount of resident thread state would allow faster blocking but will have worse single-thread performance. We believe stealing unused processor-to-cache cycles to unload blocked threads is a promising direction for future work. Special register paths to the cache or memory can also speed up register unloads, but specialized use of valuable processor-cache bandwidth may not be the right tradeoff when overall performance is considered.

In summary, the cost of blocking can be reduced to less than 100 cycles with hardware support for single-cycle load/stores, making a strong case for minimizing the run-time overhead of choosing L_{poll}.

7 Related Work

Two-phase waiting was first proposed by Ousterhout [25] who observed that blocking should be avoided if wait times are short, and suggested "pausing" a waiting process for some fixed time before blocking. His Medusa system implemented two-phase waiting with a user-settable L_{poll}. In a later study of multiprocessor scheduling algorithms, Lo and Gligor [22] found that use of two-phase waiting (with L_{poll} in between B and $2B$) improved the performance of group scheduling.

In a theoretical study of competitive algorithms, Karlin *et al.* presented a dynamic, randomized algorithm that achieves a competitive factor of $e/(e-1)$. They also proved that an algorithm that uses wait-time statistics to select an optimal static value of L_{poll} has a competitive factor of at most $e/(e-1)$. Indeed, we confirmed that this bound holds for exponential and uniform distributions: given the value of λ for an exponential distribution, we can choose L_{poll} such that the competitive factor is at most $e/(e-1)$. Given the value of U for a uniform distribution, we can choose L_{poll} such that the competitive factor is at most 4/3. (See Figures 2 – 3.) However, it is difficult to achieve this bound in practice because it requires precise knowledge of the distributions, knowledge that is dependent on run-time factors.

In recent work, Karlin *et al.* [18] empirically studied the performance of two-phase waiting algorithms for mutual-exclusion locks on a small bus-based machine. They investigated both static and dynamic methods for choosing L_{poll}. The static algorithms explored were $s/b/0.5$, $s/b/1$, and one in which wait-time profiles were used to compute an optimal value for L_{poll}. The dynamic methods used wait-time histories to dynamically adjust L_{poll}. They made two kinds of performance measurements: 1) a direct measurement of elapsed times under the different algorithms, and 2) an indirect measurement of waiting costs from post-processing the wait-time profiles of program executions. In the indirect measurements, the dynamic algorithms outperformed $s/b/0.5$ or $s/b/1$. However, in the elapsed-time measurements, which include the run-time overhead of the waiting algorithm, the static methods performed as well as the adaptive methods in one measurement while the adaptive methods performed better in the other.

Their study found always-block to perform poorly compared to two-phase waiting. This is contrary to our conclusions, which find always-block to be an acceptable waiting algorithm. As pointed out in Section 1, this is because we investigate producer-consumer and barrier synchronization in addition to mutual-exclusion synchronization, and because of the difference in the machine architectures and blocking costs.

Other studies [1, 4, 11, 23] have focused on reducing bus (or network) interference caused when spinning is used as a waiting mechanism. These studies explored methods to reduce the overhead of memory contention while spin waiting for locks and barriers. In cases of high lock contention, simple `test&test&set` [26] leads to contention at the memory module during lock releases due to sudden bursts of waiting threads vying for the lock. Exponential backoff and software queueing were shown to be effective methods for reducing the contention at locks and barriers.

These studies are concerned with making the effect of polling less intrusive on the rest of the machine by reducing contention for hardware resources. Unlike two-phase algorithms, these contention reducing mechanisms do not attempt to directly reduce the *waiting cost* and do not consider blocking as an option. However, these methods can be useful in the polling phase of two-phase algorithms to reduce the detrimental effects of contention.

Several researchers have recently been advocating the use of lock-free methods for synchronization [8, 14, 30], using `load-linked/store-conditional` [17] or `compare-and-swap` [16] as primitives. In the context of this paper, lock-free synchronization is a form of optimistic polling, and the results in this paper apply.

A typical lock-free synchronization protocol consists of a loop that executes a set of temporary updates and then attempts to commit those updates in an atomic operation at the end of the loop. The thread remains in the loop until the commit is successful. Thus, the thread is actually polling until a successful commit is executed, and the difference from lock-based synchronization is that potentially useful operations are executed while polling. Instead of interminably executing the loop in the hope of a successful commit, one could extend the lock-free protocol to a two-phase algorithm where the thread blocks after some number of unsuccessful commit attempts.

8 Conclusions

As the higher parallelism requirements of large-scale multiprocessors cause corresponding increases in synchronization rates, the overhead of waiting for synchronization becomes a significant determinant of multiprocessor performance. A poor choice of waiting algorithms can significantly increase the overhead and degrade performance. In our experiments, we observed a difference of a factor of 8.45 between the best and worst running times. This large factor was due to the potential for deadlock in always-poll algorithms. Ignoring the cases where deadlock was experienced, we still observed a significant difference of a factor of 2.38 between the best and worst running times (*cf.* matched **Jacobi**).

This paper analyzed static two-phase waiting algorithms and compared them with traditional methods of waiting in multiprocessors. Two-phase waiting combines the advantages of polling and signalling. When there are a sufficient number of short waits, two-phase waiting avoids the cost of blocking. Increased arrival rates resulting from fine-grained threads in highly parallel machines favor two-phase waiting algorithms that are geared to take advantage of short wait times, but without catastrophic behavior on long wait times.

Static two-phase algorithms are attractive because they are simple and suffer very little run-time overhead. This is especially important for fine-grained synchronization on large-scale machines with low blocking overheads comparable to the overheads of dynamic methods. For example in Alewife, we can reduce the blocking overhead to less than 100 cycles with a processor with single-cycle load-stores. This restricts the amount of run-time overhead a two-phase waiting algorithm can incur before it loses its advantage over an always-block algorithm, thus reducing the effectiveness of dynamic methods.

We suggest using knowledge about likely wait-time characteristics of different synchronization types to guide our choice of L_{poll} for static two-phase waiting algorithms. The wait-time profiles gathered show that each of the synchronization types considered have different wait-time distributions. We considered two simple but important models for wait times, and derived static settings for L_{poll} that attain or approach optimal performance. We proved that setting L_{poll} to $0.54B$ achieves a competitive factor of $e/(e-1)$ against a restricted adversary with exponentially distributed wait times, and that setting L_{poll} to $\frac{1}{2}(\sqrt{5}-1)B$ achieves a competitive factor of $(\sqrt{5}+1)/2$ for uniformly distributed wait times.

The experiments show that static two-phase waiting algorithms that rely on efficient waiting mechanisms provided in Alewife are robust under most operating circumstances. These measurements support our theoretical findings that under exponential wait times two-phase algorithms are never worse than both spinning and blocking used exclusively. The robustness of static two-phase waiting will relieve the programmer from worrying about the critical choice between polling and signalling. Furthermore, our theory shows that with some readily available knowledge on synchronization types a static setting of L_{poll} will achieve close to optimal competitive factors.

Short wait times benefit polling algorithms, such as $s//\infty$ or $Ss//\infty$, but their performance is highly sensitive to the presence of long wait times. Even a few extremely long waits can significantly hurt the performance of polling. Long wait times often result when the program is unmatched. Thus $Ss//\infty$ is an appropriate choice only when the program is guaranteed to be matched.

Always-block performs well in most of the benchmarks. In our simulations, $/b/0$ and $Ss/b/1$ generally perform similarly, except when the program is matched and wait times short. We believe that the cost of blocking can be further reduced, making always-block an even more attractive alternative. This conclusion differs from previously reported performance observations of always-block for mutual exclusion waits on bus-based multiprocessors. The acceptable performance of $/b/0$ as demonstrated by our measurements results from longer wait times encountered in our benchmarks, and an efficient blocking mechanism in Alewife. In fact, as multiprocessors scale, longer communication and synchronization latencies would tend to favor $/b/0$ even more.

Two-phase waiting and always-block algorithms will be even more attractive if the cost of blocking can be reduced further. We described several methods for reducing the cost of blocking. Switch-blocking is another attractive area for further research. Like blocking, it is a signalling mechanism. However its cost is much smaller, as it avoids unloading a thread unless all the other hardware contexts also contain switch-blocked threads.

9 Acknowledgments

We would like to thank the referees for many useful and insightful comments on this paper. One referee pointed out that barrier wait times are more accurately composed of hypoexponential distributions instead of Erlang distributions. Our gratitude to Eric Brewer, David Chaiken, Kirk Johnson, and Kai Li, who provided feedback on an earlier version of this paper. Eric suggested the idea of unloading threads at function call boundaries. Thanks also to Anna Karlin for helpful discussions on competitive algorithms. The members of the Alewife team made possible the simulation system. Machines used for simulations were donated by SUN Microsystems and Digital Equipment Corporation.

References

[1] A. Agarwal and M. Cherian. Adaptive Backoff Synchronization Techniques. In *Proceedings 16th Annual International Symposium on Computer Architecture*, pages 396–406, June 1989.

[2] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings 17th Annual International Symposium on Computer Architecture*, pages 104–114, June 1990.

[3] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication, and appears as MIT/LCS Memo TM-454, 1991.

[4] Thomas E. Anderson. The Performance Implications of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[5] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. In *Proceedings of the Workshop on Graph Reduction, (Springer-Verlag Lecture Notes in Computer Science 279)*, pages 336–369, September/October 1986.

[6] J. Aspnes, M.P. Herlihy, and N. Shavit. Counting Networks and Multi-Processor Coordination. In *Proceedings of the 23rd Annual Symposium on Theory of Computing*, pages 348–358, May 1991.

[7] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 538–568, August 1991.

[8] B. Bershad. Practical considerations for lock-free concurrent objects. Technical Report CMU-CS-91-183, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Sept 1991.

[9] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV).*, pages 224–234. ACM, April 1991.

[10] H. Davis and J. Hennessy. Characterizing the Synchronization Behavior of Parallel Programs. *ACM SIGPLAN Notices*, 23(9):198–211, September 1988.

[11] Gary Graunke and Shreekant Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *IEEE Computer*, pages 60–70, June 1990.

[12] R.H. Halstead and T. Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, New York, June 1988. IEEE.

[13] Robert H. Halstead. Multilisp: A Language for Parallel Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, October 1985.

[14] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Second ACM SIGPLAN Synposium on Principles and Practice of Parallel Programming*, pages 197–206, Mar. 1990.

[15] Qin Huang. An Analysis of Concurrent Priority Queue Algorithms. Master's thesis, EECS Department, Massachusetts Institute of Technology, Cambridge, MA, August 1990.

[16] IBM System/370 Principles of Operation. IBM, Order Number GA22-7000.

[17] E. Jensen, G. Hagensen, and J. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, Nov. 1987.

[18] A. Karlin, K. Li, M. Manasse, and S. Owicki. Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 41–55, October 1991.

[19] A. Karlin, M. Manasse, L. McGeoch, and S. Owicki. Competitive Randomized Algorithms for Non-Uniform Problems. In *Proceedings 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 301–309, January 1990.

[20] Kiyoshi Kurihara, David Chaiken, and Anant Agarwal. Latency Tolerance through Multithreading in Large-Scale Multiprocessors. In *Proceedings International Symposium on Shared Memory Multiprocessing*, Japan, April 1991. IPS Press.

[21] Beng-Hong Lim. Waiting Algorithms for Synchronization in Large-Scale Multiprocessors. MIT/LCS TR-498, Massachusetts Institute of Technology, Cambridge, MA, February 1991. S.M. Thesis.

[22] S. Lo and V. Gligor. A Comparative Analysis of Multiprocessor Scheduling Algorithms. In *7th International Conference on Distributed Computing Systems*, pages 356–363. IEEE, Sept. 1987.

[23] 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.

[24] Eric Mohr, David A. Kranz, and Robert H. Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, Jul 1991.

[25] John K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *3rd International Conference on Distributed Computing Systems*, pages 22–30. IEEE, 1982.

[26] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 340–347. IEEE, June 1984.

[27] B.J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *SPIE*, 298:241–248, 1981.

[28] SPARC Architecture Manual, 1988. SUN Microsystems, Mountain View, California.

[29] K.S. Trivedi. *Probability and Statistics with Reliability, Queueing and Computer Science Applications*. Prentice-Hall, 1982.

[30] J.M. Wing and C. Gong. A Library of Concurrent Objects and Their Proofs of Correctness. Technical Report CMU-CS-90-151, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Jul 1990.

[31] P.-C. Yew, N.-F. Tzeng, and D.H. Lawrie. Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.

[32] J. Zahorjan and E. Lazowska. Spinning Versus Blocking in Parallel Systems with Uncertainty. Technical Report TR-88-03-01, Dept. of Computer Science, University of Washington, Seattle, WA, Mar 1988.

A Wait-Time Models and Distributions

This section proposes wait-time models for producer-consumer, barrier and mutual-exclusion synchronization under the assumption of Poisson arrivals of synchronizing threads. These models are used to help guide our choice of L_{poll} based on synchronization type.

We also present wait-time profiles gathered from the simulations and compare them with the proposed models. These profiles also help explain the performance results presented in Section 6. A number of the wait-time profiles approximate an exponential distribution. Whenever this is so, a semi-log plot is used so the exponential distribution is more easily recognizable as a linear set of points. Linear regressions on the log values of the wait time frequencies are also plotted, and correspond to fitting exponential curves through the original set of points. Outliers with frequencies less than 10 were pruned in the regressions.

A.1 Producer-Consumer

As mentioned in Section 4, producer-consumer synchronization experiences exponentially distributed wait times under Poisson arrivals of synchronizing threads. This simple model of wait times led to useful theorems about the performance of two-phase waiting algorithms, as presented in Section 4.

Figures 4 and 5 present semi-log plots of wait-time profiles obtained from benchmarks with producer-consumer synchronization. These profiles lend some support to our exponential wait time assumption that formed the basis of our mathematical analysis, and also help in obtaining insights into the differences in the performance of waiting algorithms.

We can see from the plots that the wait times are indeed largely exponentially distributed. However, there is some deviation for short wait times in unmatched versions of **MGrid** and **Jacobi**.

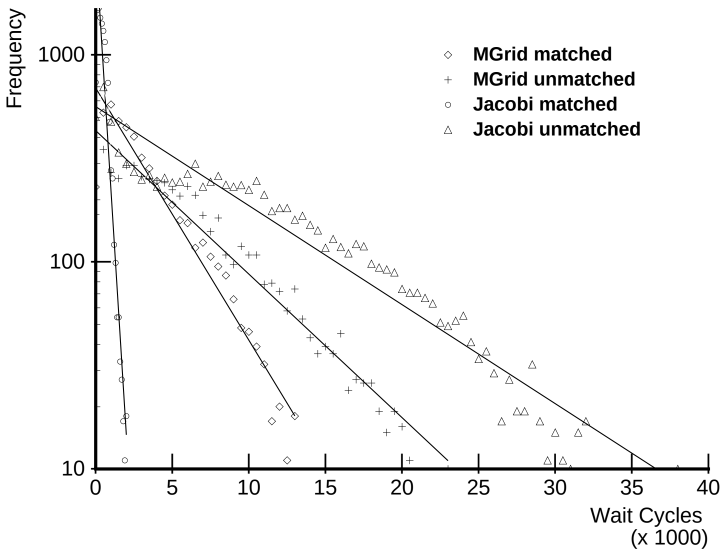

Figure 4: Measured wait times for J-structure readers.

We believe this is due to the effect of blocking on wait times. (Recall that an unmatched program requires blocking to avoid deadlock.) Blocked threads experience some delay before resuming execution. Since a blocked thread might itself be a producer, whose output is awaited by some other threads, this delay can cause a fraction of wait times to be skewed upward.

Although it would be premature to conclude from these few sample benchmarks that producer-consumer wait times are always exponentially distributed, the measurements show the existence of parallel programs that approximate such wait times, and the theoretical analysis of Section 4 applies.

A.2 Barrier Synchronization

The uniform distribution is a reasonable model for barrier wait times. Such wait times would arise if inter-barrier thread execution lengths are uniformly distributed within some time interval. Moreover, the following analysis shows that if arrivals at a barrier is Poisson, then the uniform distribution is a useful approximation to the resulting PDF of barrier wait times.

The wait time encountered at a barrier depends on two factors: 1) the number of threads that have yet to arrive at the barrier, and 2) the arrival rate of the threads. Let M be the number of participants in a barrier and r be the number of threads yet to arrive at a given point in time. Clearly $r = (M - i)$ for the i^{th} arrival.

Assuming Poisson arrivals of participating threads, the wait time for the i^{th} arrival is the maximum of r exponentially distributed random variables. This PDF is the hypoexponential distribution with parameters $\lambda, 2\lambda, .., r\lambda$ ([29] p. 166), where λ is the arrival rate of *each* of the threads. To make the analysis tractable, we make a further simplifying assumption that the arrival rate of participants at a barrier is independent of the remaining number of arrivals. Consequently, the wait-time distribution for r arrivals becomes the r^{th} order Erlang PDF, or,

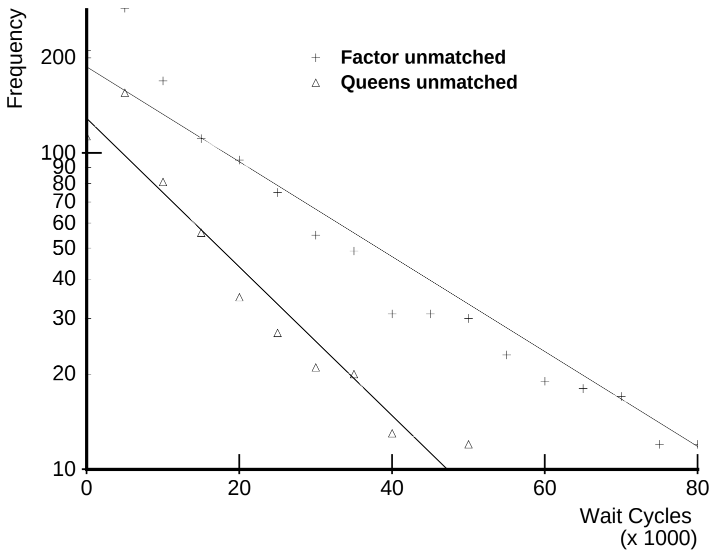

Figure 5: Measured wait times for futures.

$$f_r(t) = \frac{\lambda(\lambda t)^{r-1}e^{-\lambda t}}{(r-1)!} \tag{8}$$

For a fixed λ, the Erlang shifts the probability from shorter to longer wait-times over the hypoexponential distribution.

We can now derive the PDF for barrier wait-times as follows. Since a scalable implementation of a barrier must use distributed data structures, a waiting thread cannot easily determine the remaining number of threads. Let $N = M - 1$. Since each arriving thread that is forced to wait has an equal probability of $1/N$ of being the i^{th} arrival, where $1 \leq i \leq N$, the PDF for barrier wait times is the sum of the PDFs of the first N Erlang PDFs scaled by $1/N$. Thus,

$$f(t) = \frac{1}{N}\sum_{r=1}^{N} f_r(t) \tag{9}$$

We can simplify this sum using LaPlace transforms.

$$\begin{aligned} \mathcal{L}(f(t)) &= \frac{1}{N}\sum_{r=1}^{N} f_r^T(s) = \frac{1}{N}\sum_{r=1}^{N}\left(\frac{\lambda}{s+\lambda}\right)^r \\ &= \frac{\lambda}{N}\left(\frac{1}{s} - \frac{1}{s}\left(\frac{\lambda}{s+\lambda}\right)^N\right) \end{aligned}$$

Therefore, taking the inverse LaPlace transform, we obtain $f(t)$ for barrier wait times as

$$f(t) = \frac{\lambda}{N}\left(1 - \int_0^t f_N(\tau)d\tau\right) = \frac{\lambda}{N}\int_t^{\infty} f_N(\tau)d\tau \tag{10}$$

where $f_N(\tau)$ is obtained from Equation 8.

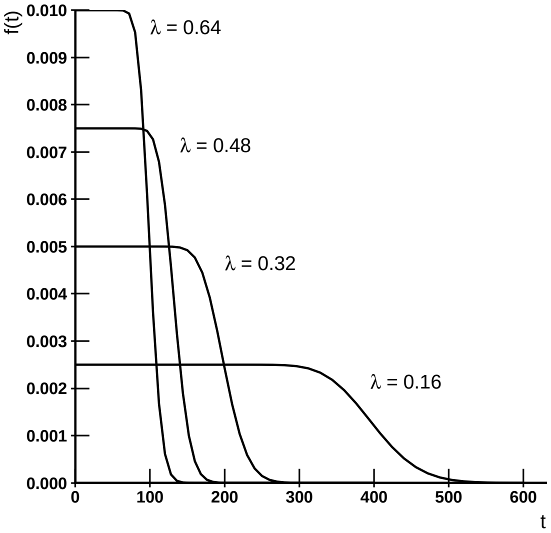

Figure 6: Modeled PDF for barrier wait times. $M = 64$ participants, varying λ.

What does $f(t)$ look like? The N^{th} Erlang distribution approaches the normal distribution with mean N/λ and variance N/λ^2. For scalable performance, both N and λ must increase at the about same rate. Under these conditions, $f(t)$ approaches a uniform distribution with a value of λ/N from $t = 0$ to N/λ. Figure 6 plots the PDF for $M = 64$ while varying λ. The expected wait time is $(N+1)/2\lambda$. Note that for a fixed arrival rate, the expected wait time increases linearly with the number of participants.

Figure 7 presents the wait-time profiles for **CGrad** and **Jacobi-Bar**. These distributions do not look uniform due to the software overhead of the combining tree implementation of barriers which introduce additional delays when arriving at and leaving from the barrier.

To filter out this software overhead, we ran a version of **Jacobi-Bar** with a simple counter implementation of barriers. We executed this benchmark on a simulation of a perfect memory system to eliminate the effect of hardware contention on this simple barrier implementation. Figure 8 presents the resulting wait-time profile which is close to uniform except at the tails. Most of the overhead of the software combining tree has been eliminated; what remains is software contention for the barrier counter.

A.3 Mutual Exclusion

As mentioned in Section 4, the wait time for mutual-exclusion locks without queues can be modeled as either exponential or uniform. Here, we derive a model of wait times for locks that enforce a FCFS queueing order.

The wait time for a lock requester is the time between requesting a lock and actually acquiring it. Assuming Poisson arrivals of lock requesters, exponential service times for lock holders, and a finite number of lock requesters, M, a mutex can be modeled as an $M/M/1//M$ queue. Let p_q represent the probability that an arriving thread finds the queue length to be q. The probability mass function of the queue length encountered by an arriving thread for such a queueing system is

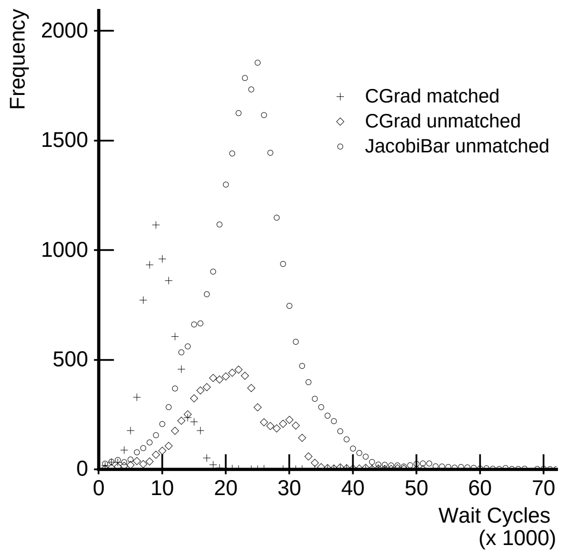

Figure 7: Measured barrier wait times for **CGrad** and **Jacobi-Bar**.

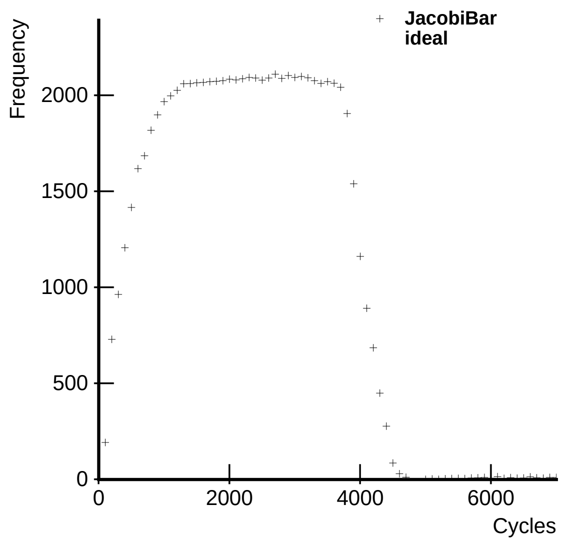

Figure 8: Measured barrier wait times for **Jacobi-Bar** on an ideal system.

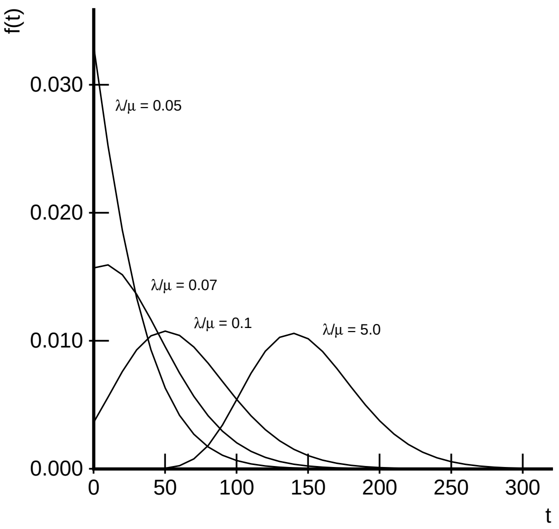

Figure 9: Modeled PDF for mutex wait times. $M = 16$, $\mu = 0.1$, varying λ.

$$
\begin{aligned}
p_q &= \begin{cases} p_0(\frac{\lambda}{\mu})^q \frac{M!}{(M-q)!} & 0 \leq q \leq M \\ 0 & q > M \end{cases} \\
\text{where} \quad p_0 &= \left[\sum_{q=0}^{M} (\frac{\lambda}{\mu})^q \frac{M!}{(M-q)!}\right]^{-1}
\end{aligned}
\tag{11}
$$

λ is the arrival rate of each lock requester and μ is the departure rate of lock holders.

If waiting threads are served in a first-come first-served (FCFS) fashion, a thread requesting a mutex with a queue length of q has to wait for q threads to depart. The conditional PDF of the waiting time for a lock requester given q threads ahead of it is the q^{th} order Erlang PDF with arrival rate μ.

$$
f_q(t) = \frac{\mu(\mu t)^{q-1} e^{-\mu t}}{(q-1)!} \tag{12}
$$

Therefore, the PDF of wait times at a mutex is

$$
f(t) = \frac{1}{(1 - p_0 - p_M)} \sum_{q=1}^{M-1} p_q f_q(t) \tag{13}
$$

Figure 9 plots this PDF for $M = 16$, $\mu = 0.1$, and various values of λ. The PDF ranges from an exponential when λ/μ is very small to an $(M-1)$th-order Erlang when λ/μ is very large. This fits our intuition that expected wait times get longer as lock utilization increases. It also implies that the analysis for exponential wait times applies when lock contention is low enough.

Figures 10 and 11 present the measured wait times for the mutual-exclusion benchmarks. These profiles help us explain the experimental results for mutual exclusion presented in Section 6. The

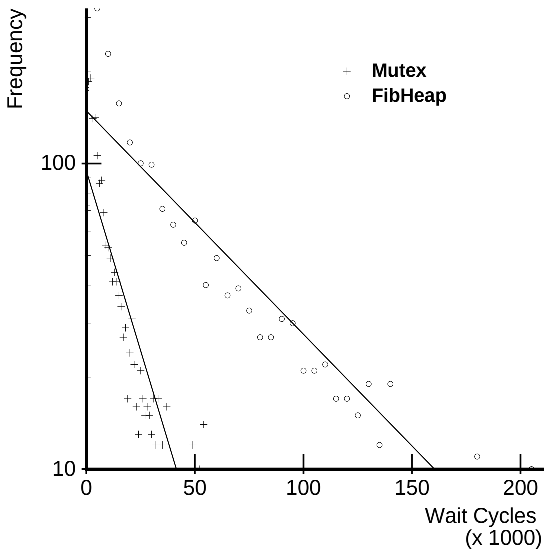

Figure 10: Semi-log plot of measured mutex wait times in **FibHeap** and **Mutex**.

model proposed above does not apply to the measured profiles for the following reasons. First, mutex waiters were not served in FCFS order in those benchmarks, and second, contention for hardware resources due to high lock utilization had a significant effect on wait times. The effect of contention in the experiments was discussed in Section 6.3. For **FibHeap** and **Mutex**, the wait times appear to be exponential due to randomness introduced by contention. Note the long wait times in **FibHeap** and **Mutex** as compared to **CountNet**.

B Deriving Optimal L_{poll} for Uniform Distributions

In this section, we prove that under uniformly distributed wait times, a static two-phase algorithm with $\alpha = (\sqrt{5}-1)/2$ has a competitive factor no larger than $(\sqrt{5}+1)/2$, and that no other value of α yields a lower competitive factor over the entire range of the parameter of the uniform distribution. Refer to Figure 3 for an illustration of these results.

Theorem 4 *Under uniformly distributed wait times from $t = 0$ to U, with $\alpha = (\sqrt{5}-1)/2 \approx 0.618$, the competitive factor of two-phase waiting, $E[C_{Ss/b/\alpha}]/E[C_{Ss/b/Opt}]$, is at most $(\sqrt{5}+1)/2 \approx 1.618$, regardless of the parameter, U, of the distribution. Furthermore, if $\alpha \neq (\sqrt{5}-1)/2$, then the competitive factor under uniformly distributed wait times is larger than $(\sqrt{5}+1)/2$.*

Proof: Let wait time be uniformly distributed from $t = 0$ to $t = U$. From Equations 1–2, we can derive the following expressions for the expected costs of static two-phase waiting algorithms and the optimal off-line algorithm.

$$E[C_{Ss/b/\alpha}] = \begin{cases} \int_0^U \frac{t}{\beta}\frac{1}{U}dt = \frac{U}{2\beta} & \text{if } U \leq \alpha\beta B \\ \int_0^{\alpha\beta B} \frac{t}{\beta}\frac{1}{U}dt + \int_{\alpha\beta B}^U (1+\alpha)\frac{B}{U}dt = \frac{1}{U}\left[(1+\alpha)BU - (1+\frac{\alpha}{2})\alpha\beta B^2\right] & \text{otherwise} \end{cases}$$

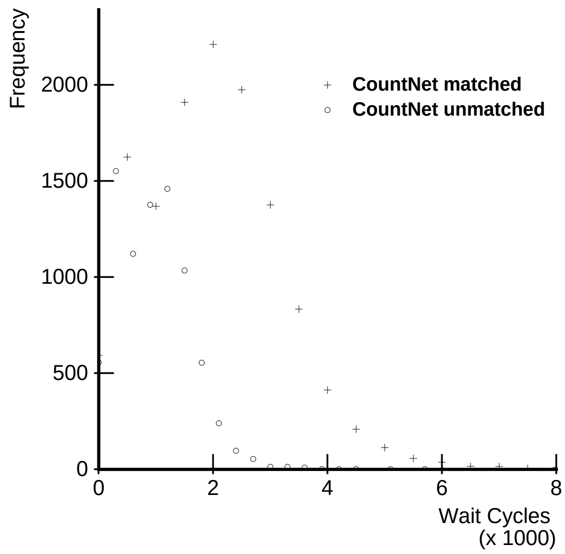

Figure 11: Measured mutex wait times in **CountNet**.

$$E[C_{Ss/b/Opt}] = \begin{cases} \int_0^U \frac{t}{\beta}\frac{1}{U}dt = \frac{U}{2\beta} & \text{if } U \leq \beta B \\ \int_0^{\beta B} \frac{t}{\beta}\frac{1}{U}dt + \int_{\beta B}^{U} \frac{B}{U}dt = \frac{1}{U}\left[BU - \frac{1}{2}\beta B^2\right] & \text{otherwise} \end{cases}$$

Let us consider the case when $\alpha \leq 1$. Substituting $x = U/\beta B$, we get the following expressions for the expected competitive factor, $c = E[C_{Ss/b/\alpha}]/E[C_{Ss/b/Opt}]$.

$$c = \begin{cases} 1 & \text{if } x \leq \alpha \\ \left[2(1+\alpha)x - \alpha(\alpha+2)\right]/x^2 & \text{if } \alpha \leq x \leq 1 \\ \left[2(1+\alpha)x - \alpha(\alpha+2)\right]/(2x-1) & \text{if } x \geq 1 \end{cases}$$

Also,

$$\frac{\partial c}{\partial x} = \begin{cases} 0 & \text{if } x \leq \alpha \\ 2\left[\alpha(2+\alpha) - (1+\alpha)x\right]/x^3 & \text{if } \alpha \leq x \leq 1 \\ \left[2(\alpha^2+\alpha-1)\right]/(2x-1)^2 & \text{if } x \geq 1 \end{cases}$$

In the range $x \geq 1$, $\frac{\partial c}{\partial x} = 0$ when either $x = \infty$ or $(\alpha^2 + \alpha - 1) = 0$. This implies that when $\alpha = (\sqrt{5}-1)/2$, the value of c is $(\sqrt{5}+1)/2$ over the entire range $x \geq 1$.

In the range $\alpha \leq x \leq 1$, $\frac{\partial c}{\partial x} = 0$ when either $x = \infty$ or $x = \alpha(2+\alpha)/(1+\alpha)$. Also, $\frac{\partial^2 c}{\partial x^2}$ is negative. These imply that when $\alpha = (\sqrt{5}-1)/2$, c has a maximum value of $(\sqrt{5}+1)/2$ at $x = 1$. Therefore, $c \leq (\sqrt{5}+1)/2$ when $\alpha = (\sqrt{5}-1)/2$.

We now have to show that no other setting of α yields a competitive factor of less than 1.618 over the entire range of U, so that α is the optimal setting for uniformly distributed wait times.

As $x \to \infty$, c approaches $1+\alpha$. Therefore the competitive factor is larger than $(\sqrt{5}+1)/2$ when $\alpha > (\sqrt{5}-1)/2$.

Now consider the case when $\alpha < (\sqrt{5}-1)/2$. In the range $x \geq 1$, $\frac{\partial c}{\partial x} < 0$ so that c monotonically decreases with x. Therefore the maximum value of c in this range is $(2-\alpha^2)$ when $x = 1$. Since $\alpha < (\sqrt{5}-1)/2 \Leftrightarrow (2-\alpha^2) > (\sqrt{5}+1)/2$, the theorem also holds for all $\alpha < (\sqrt{5}-1)/2$. $\square$