Barrier Synchronization on a Loaded SMP using Two-Phase Waiting Algorithms

By Dan Tsafrir, Supervised by Dror Feitelson

September 2001

Acknowledgments

My warmest thanks go to my teacher and supervisor Dror Feitelson. His leadership, scholarship, sense of clarity, and insight lie at the heart of every aspect of this thesis. Working with him has been a privilege and an honor.

I have greatly appreciated working with all my colleagues and friends at the Parallel Systems Laboratory; they have been very helpful throughout the few years that we have spent together. In particular I would like to thank Uri Lublin, David Er-el, Avi Kavas, Daniel Citron, Anat Batat, Maayan Geffet and especially Yoav Etsion (etsman).

I am also thankful to my friends from the System-group for their patience and constant help with all kinds of administrative problems: Tomer Klainer (mandor), Alex Kremer (kresa), Ely Levy, and mostly to Tomas Winkler.

Finally, I wish to thank my parents and Zohar for their unconditional, faithful, and loving support.

Abstract

Little work has been done on the performance of barrier synchronization using two-phase blocking, as the common wisdom is that it is useless to spin if the total number of threads in the system exceeds the number of processors. We challenge this view and show that it may be beneficial to spin-wait if the spinning period is set to be a bit more than twice the context switch overhead (rather than being equal to it). We show that the success of our approach is due to an inherent property of general-purpose schedulers, which tend to select threads that become unblocked for immediate execution. We find that this property causes applications based on barriers to fall into a previously unnoticed pattern, denoted “alternating synchronization”, which is quite different from the patterns typically assumed in theoretical analyses. By merely choosing an appropriate spinning period, we leverage alternating synchronization to implicitly nudge the system into simultaneously co-scheduling the application’s threads, thereby dramatically reducing the overhead of synchronization and significantly improving the performance.

Contents

Chapter 1

Introduction

1.1 Parallel Computers and Applications

A parallel computer is "a collection of processing elements that communicate and cooperate to solve large problems fast" [13]. The main motivation for developing such computers is that "whatever the performance of a single processor at a given time, higher performance can, in principle, be achieved by utilizing many such processors" [6]. Nowadays, high performance multiprocessors range from the fastest and most expensive supercomputers to scalable Internet servers to individual desktops.

Simple parallel applications are composed from a number of independent sequential programs that are executing simultaneously. However, the more interesting parallel applications involve cooperation, communication and synchronization between the concurrently computing entities. The former applications, namely those that seldom synchronize or communicate, are usually referred to as *coarse grain*. The latter, namely applications that perform frequent synchronization, are known to be *fine grain*. The smaller (finer) the application's *granularity*, the greater the potential for parallelism, and hence speedup. Keckler et al. [17] have shown that parallelism can be exploited with grain size as small as 20 machine cycles.

The manner in which threads of a parallel application communicate is dependent on the *machine class* on which they execute. Parallel machines may be divided into two classes:

Shared memory machines A collection of processors that may access a collection of memory modules through some kind of hardware interconnect. The key property of this class is that communication takes the form of conventional memory access instructions. Threads can be configured in such a way that portions of their address space are shared. Coordination among threads is therefore accomplished by reading from and writing to shared variables located within the shared memory portion.

Message passing machines Such multiprocessors employ a cluster of complete computers or *nodes* as building blocks. Each node has its own microprocessor, memory and I/O devices, though typically it doesn't have a monitor and a keyboard. Nodes are interconnected by a high performance network (often with much higher capability than the standard local area network). Communication between threads executing on different nodes is done via explicit I/O operations i.e. by message passing.

This work is related to the first machine class. Of particular interest are the dominant bus based *symmetric shared memory multiprocessors* or *SMP*s that "form the bread and butter of modern commercial parallel machines" [6]. Such multiprocessors are of small to moderate scale and provide a global physical address space and symmetric access to the main memory from any processor.

1.2 Synchronization Mechanisms

A common operation in SMP synchronizing programs is acquiring a *lock* to achieve mutual exclusion, thus protecting access to shared data (a thread my access some predefined memory location only if it has

acquired the appropriate lock). Nowadays, all microprocessors support instructions that allow this type of synchronization.

In addition, most numerical parallel applications make heavy use of collective communication known as *barrier synchronization*. These type of applications obey the following computation model: Each thread from the *job* (the parallel application) computes alone for a while. Then, it reaches a point where it should communicate with its peers. Since communication is done via global data structures located on shared memory, the thread must somehow make sure its peers have already written the information it needs. In this case we say that the thread has reached a *synchronization point.* The barrier synchronization mechanism will allow threads to continue to compute only upon the arrival of the last thread to the synchronization point.

1.3 TradItional Waiting Algorithms: Spin & Block

Synchronization between peers in a parallel application is therefore a common operation. Assuming other threads are waiting for a processor, a thread that needs to wait for synchronization is faced with a dilemma: how should it wait for the synchronization-event ?
The two canonical waiting algorithms are:

1. *busy-wait*, a thread *spins* in a loop while repeatedly testing some condition that indicates whether the synchronization event occurred.

2. *block*, when a thread becomes aware that it needs to wait, it blocks (suspends) itself by releasing its processor and is enqueued to some waiting queue, until such time in which the awaited event occurs and the thread is made ready-to-run again.

Note that the choice of the waiting algorithm is quite independent from the particular nature of the synchronization mechanism. If the awaited event will happen within a short period of time, it is usually better to spin, both in terms of progress of this thread and in terms of the system resources lost to overhead due to *context switch* (saving thread's state, deciding which will be the next thread to run and dispatching it). However, for longer time periods, it's presumably better to immediately block and avoid wasting valuable CPU time otherwise utilized by other ready-to-run threads.

Drawbacks of making the wrong choice are obvious: When parallel applications synchronize frequently, the overhead of synchronization can be very significant. Jiang & Singh [14] have examined where parallel applications (from SPLASH-2 [28] and more) spend their time when executed on a real large scale machine (SGI Origin2000 [20] with up to 128 CPUs). Their findings indicated some applications may spend up to 55% of their elapsed time while trying to synchronize (usually due to barriers). Karlin et al. [16] have witnessed that with immediate blocking, some applications spend over a third of their elapsed time on context switches. Theoretically, the penalty of *always-block* may converge to 100% of the execution time (for very long context switch durations). Likewise, on systems with no preemption, the worst case performance of *always-spin* is arbitrarily bad since it might introduce a deadlock (if thread t_1 spins while waiting for synchronization with t_2, and consequently t_1 deprives t_2 of a processor).

1.4 Competitive Waiting Algorithms and Other Related Work

The two waiting algorithms mentioned above are seemingly reconciled by using *two phase waiting*, in which a period of busy waiting L_{spin} is followed by blocking. Two phase waiting was first proposed by Ousterhout [23] in 1982 who observed that blocking should be avoided if wait times are short, and suggested "pausing" a waiting thread for some (user defined) fixed time before blocking.

For lock synchronization, Karlin et al. [15] (1990) have shown that a variant of this method, where the time spent spinning is equal to a context switch duration C, is *2-competitive*. This means the amortized cost of this strategy is at most twice that of the optimal off-line algorithm (which has complete knowledge of synchronization wait times). The justification of this claim is trivial. They have also analytically proven that there is no deterministic algorithm that has a competitive ratio smaller than 2. Finally, they have proven that

a randomized algorithm can achieve strongly competitive ratios approaching $\frac{e}{e-1} \approx 1.58$ (when compared to the optimal off-line algorithm) under the assumption that wait time distributions obey some unknown but time invariant probability distribution.

Later, Karlin et al. [16] (1991) have empirically evaluated a collection of parallel applications on a small scale SMP (7-processor Firefly [25]) while using a suite of two phase waiting competitive algorithms that included various *fixed spinning* deterministic algorithms with constant maximal spin durations (among them $L_{spin} = C$ and $L_{spin} = \frac{C}{2}$), and various *variable competitive* algorithms with an adaptive maximal spin duration (among which the algorithm with the competitive ratio that converges to $\frac{e}{e-1}$). They have concluded that fixed spin algorithms are usually better than traditional always spin or block, and that adaptive algorithms are usually better than fixed spinning.

Inspired by the work presented in [15] and [16], Lim & Agarwal [21] (1993) have investigated two-phase waiting algorithm through analysis and experiments in the context of a larger machine (a simulator of the 64-processor MIT Alewife machine called ASIM [1]). Each processor in the Alewife has four *hardware contexts*. A hardware context is a set of registers that implement the processor-resident state of a thread. A waiting thread on such a multithreaded processor, can switch rapidly to another processor-resident thread in a round-robin fashion. This type of waiting, where the wait time is interleaved with executions of other threads, is called *switch-spinning*, and was the only type of spinning used in this work. Consequently, Lim & Agarwal focused on methods for statically determining L_{spin}, arguing that the run-time overhead of doing it dynamically can be comparable to the cost of blocking on machines similar to the Alewife.

A key difference between the work of Karlin et al. and the work of Lim & Agarwal was that the latter were motivated by the observation that different synchronization mechanisms (barriers, locks) exhibit different wait time distributions, and therefore need separate evaluation. Under the conjecture of Poisson arrivals of synchronizing threads, they have shown that the exponential and uniform distributions are reasonable models of wait times for lock and barrier synchronization, respectively. They have proven that a static choice of L_{spin} can yield close to optimal on-line performance against an "adversary" algorithm that is restricted to choosing wait times from a fixed family of probability distributions. This result allowed them to make an optimal static choice of L_{spin} based on synchronization type. For exponentially distributed wait times (associated with locks), they have proven that setting $L_{spin} = \ln(e-1)C \approx 0.54C$ resulted in a competitive ratio of $\frac{e}{e-1}$ in comparison to the optimal off-line algorithm. For uniformly distributed wait times (associated with barriers), they have proven that setting $L_{spin} = \frac{1}{2}\left(\sqrt{5}-1\right)C \approx 0.62C$ results in a competitive ratio of $\frac{1}{2}\left(\sqrt{5}+1\right) \approx 1.62$ (the golden ratio).

In their practical experiments, Agarwal & Lim have differentiated between *matched* and *unmatched* programs. 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. In practice, unmatched programs were simulated by running the original (matched) programs while reducing the number of hardware contexts from four to two. Always-block ($L_{spin} = 0$) was found to be a good waiting algorithm with performance that was usually close to the best of the algorithms compared. Fixed spinning of $L_{spin} = C$ or $L_{spin} = \frac{C}{2}$ usually produced slightly better results. A particularly relevant conclusion to this work was that unmatched barrier based application should always use $L_{spin} = 0$ (i.e. always-block). Since most popular microprocessors have only one hardware context, the practical meaning of this recommendation is that any barrier based application should usually choose the always-block waiting algorithm when other ready threads are waiting for a processor.

Based on the work described above, Kontothanassis et al. [18] (1997) developed a set of *scheduler conscious* synchronization algorithms, that contrary to earlier work, are supposedly suited for a multiprogrammed preemptive environment. Such algorithms may interact with the kernel in order to ensure (for example) that a lock holding thread will not be preempted. Unfortunately, their proposed barrier algorithm doesn't handle multiprogramming and assumes processors are partitioned among applications. However, it does allow an application with T threads to execute on P processors even when $T > P$. Kontothanassis et al. have pointed out that fixed spin algorithms lead to uniform policy for all threads: either all will spin, or all will block. Their suggested barrier algorithm makes the trivial optimal spin-versus-block decision in each individual thread: for a job composed of T threads running on P processors, the first $T - P$ threads to reach a barrier will block while the remaining P will spin.

1.5 Motivation

Though theoretically pleasing, the analytical results presented in [15] and [21] have an inherent flaw: the "optimal off-line algorithm" does not qualify to be the *actual* optimal algorithm. The following is a simple example that demonstrates this. Assume a 2-processors machine executing two jobs: J_1 and J_2, each composed of two threads. Let $\left\{t_1^1,\, t_2^1\right\}$ and $\left\{t_1^2,\, t_2^2\right\}$ be the threads composing J_1 and J_2 respectively. Assume all threads profiles are identical, such that each thread computes for μ cycles and then needs to synchronize with its peer (this can be viewed both as lock and barrier synchronization). Further assume that t_1^1 and t_1^2 are currently executing and that the waiting algorithm used is $L_{spin} = C$ (which was proven to be 2-competitive compared to the optimal off-line algorithm). t_1^1 and t_1^2 compute for μ cycles and reach a synchronization point, they spin for C cycles and block. t_2^1 and t_2^2 take their place, compute for μ cycles, spin for C cycles and also block, only to be replaced by t_1^1 and t_1^2 ... and so on. Let N denote the total number of synchronization points in the computation. Consequently, the total duration of the execution described above is:

$$|J|\, N \left(C_{(spin)} + C_{(block)} + \mu_{(compute)}\right) \;=\; 2N\left(2C + \mu\right) \;=\; 4NC + 2N\mu$$

Of course the duration of the optimal off-line algorithm would have been:

$$2N\left(C + \mu\right) \;=\; 2NC + 2N\mu$$

because it would have avoided all the useless spinning. Indeed, a factor of approximately 2 (when C is considerably bigger than μ) as implied by the competition ratio. However, the duration of the actual optimal algorithm would have been $\approx 2N\mu$, if J_1's threads would have spun (for example) for $2C$ instead of C, because this would allow t_1^1 and t_1^2 to execute together and therefore to avoid the context switch associated with each synchronization point (same goes for J_2). It follows that the optimal off line algorithm may be arbitrarily bad in comparison to the actual optimal algorithm. For example, if $\mu = 1$ and $C = 100$, then the former is approximately 100 times slower than the latter.

The scenario specified above is the simplest we could find in order to demonstrate our claim. However, it fails to emphasize the source of the defect of the optimal off-line algorithm. Let $\{W_i\}_{i=1}^{N}$ denote the wait times series of some thread t within an execution of some parallel application. The philosophy behind the optimal off-line algorithm (compared for example to a waiting algorithm with $L_{spin} = C$) is the following: If $W_k < C$, than it was advantageous for t to spin while it was trying to synchronize for the k-th time. If on the other hand $W_k \geq C$, then t should have immediately blocked. The underlying assumption that lies beneath this concept is that a change of the k-th spin-vs-block decision will not affect $\{W_i\}_{i=k+1}^{N}$ which is of course erroneous.

The focus of this work is on barrier synchronization. Indeed, Karlin et al. [16] have claimed that spinning is only worth while if the awaited thread is currently executing (because when waiting for "a thread which is waiting for a processor, it makes little sense to spin"). This implies that barrier based applications will have little chance to synchronize when running on a heavily loaded SMP. In addition, Lim & Agarwal concluded that unmatched barrier based applications should never spin, which for most processors means that whenever the number of threads exceeds the number of processors, then threads should always-block (which in any case is a good waiting algorithm according to Lim & Agarwal). However, the flaw that we have identified in the analysis related to the optimal off-line algorithm, leads us to believe that it's possible that there exists a waiting algorithm which will achieve better exploitation of the SMP, by allowing barrier based application to compute without having to always-block at each synchronization point. Our experience has indicated that threads either always manage to synchronize during their busy-waiting period, or else they always block. We set out to examine this phenomenon by a detailed study of how two-phase waiting for barrier synchronization depends on system load. Our goal is to gain a better understanding of parallel barrier based application operating in a multitasking environment, and check the implications of high loads on such applications. We hope these understandings will serve designers and implementors of barrier algorithms and will allow better utilization of SMPs.

1.6 Choosing a Scheduler

Aspiring for this work to have a practical value, we wanted to evaluate barrier synchronization on a SMP that uses a popular and common scheduler. Nowadays, most popular operating systems (even non UNIX) conform to *POSIX1.b* (formally known as POSIX.4) which is the *Portable Operating System Interface* [12]. POSIX1.b defines three types of scheduling policies:

SCHED_FIFO Processes running under this policy, run until they give up the processor, usually by blocking for I/O, waiting for a semaphore, or executing some other blocking system call.

SCHED_RR Operates just like SCHED_FIFO, except that processes run with a system given quantum (RR stands for Round-Robin).

SCHED_OTHER Is not defined by POSIX but its presence is mandated. This is the default (and thus, arguably, the most important) timesharing scheduler used by the operating system.

Of course, since it is the most common and widely used, we are primarily interested in the last policy. Unfortunately, its semantics are not defined by POSIX. However, we do know that the default schedulers of general purpose operating systems such as all the flavors of UNIX and Windows are:

1. preemptive, based on quanta, and
2. priority-based, such that I/O bound (interactive) processes are favored.

Our chosen method to evaluate barrier synchronization within loaded systems is therefore:

1. Evaluating the standard (and relatively simple) SCHED_RR scheduler, which can be thought of as a simple version of SCHED_OTHER where all the priorities of the various processes are equal all the time, and
2. Choosing a popular scheduler as a representative from within the various operating systems' implementations of SCHED_OTHER; use the previous step's understandings in order to evaluate this scheduler, and hopefully be able to project our findings to any preemptive priority-based scheduler.

1.7 Thesis Outline

We will evaluate barrier synchronization within a loaded system in a number of stages, with increasing complexity and realism:

1. One synchronizing job with a compute-bound background load,
2. A set of identical synchronizing jobs,
3. A set of heterogeneous jobs, with different synchronization behavior,
4. Same as above with priority based scheduling rather than round-robin scheduling.

Chapter 2 will give a detailed description of the SMP simulator we have used in this work. Chapters 3, 4 and 5 will evaluate the SCHED_RR scheduler according to the steps described above. Chapter 6 will introduce the Linux scheduler, which we have chosen as a representative for SCHED_OTHER. Chapters 7, 8 and 9 will evaluate this scheduler similarly to SCHED_RR. Finally, chapter 10 will discuss and conclude this work.

Chapter 2

The SMP Simulator

Throughout this work we use an event driven SMP simulator. This chapter describes this simulator in detail. Only SCHED_RR aspects of the simulator are reviewed. Details regarding the chosen SCHED_OTHER policy will be discussed in chapter 7 (after the Linux scheduler is introduced in chapter 6).

2.1 General

The simulator distinguishes between *synchronizing and non-synchronizing jobs*. Since threads of a non-synchronizing job do not interact with each other, it has no significance if we view each individual thread as an independent job. Consequently, within the context of the simulator, a non-synchronizing job is always composed of a single thread. The only form of interaction between threads belonging to a synchronizing job is barrier-synchronization. Synchronizing jobs may vary in many parameters, among which are size, granularity and more. A fixed-spinning waiting algorithm is used in order to perform barrier-synchronization.

At any time instance, each executing thread within the simulator is in exactly one of four possible states:

Ready: The thread is found in the ready queue waiting to be allocated a processor.

Running: The thread is currently running (possibly spinning).

Blocked: The synchronizing thread has decided to block until its peers will reach the current barrier. The thread will be moved back to the tail of the ready-queue when the current barrier is completed (upon the arrival of the last thread from the job).

Finished: The program executed by the thread has terminated.

Figure 2.1 demonstrates how threads may move between the above states.

The simulator is event driven. Future events are held in a global priority-queue, ordered according to their execution time. Execution times are expressed in cycles (integral number). Obviously, at most one event may be executed, in any given time instance, for each processor within the SMP. This means that two different events may have the same execution time iff they are executed within the context of different processors. When an event is executed, it updates the global SMP clock with its time of execution (since the simulator is a serial program, there is only one clock). Each transition between the various thread states described in figure 2.1 is associated with an event. In addition, synchronizing threads use events to simulate the end of a computation-phase and the implementation of the barrier synchronization mechanism:

- repeatedly polling to check for barrier completion (maximal spin duration is fixed)
- upon success, push the next end-of-computation-phase event
- upon failure, trigger a yield-event and enqueue to blocked state.

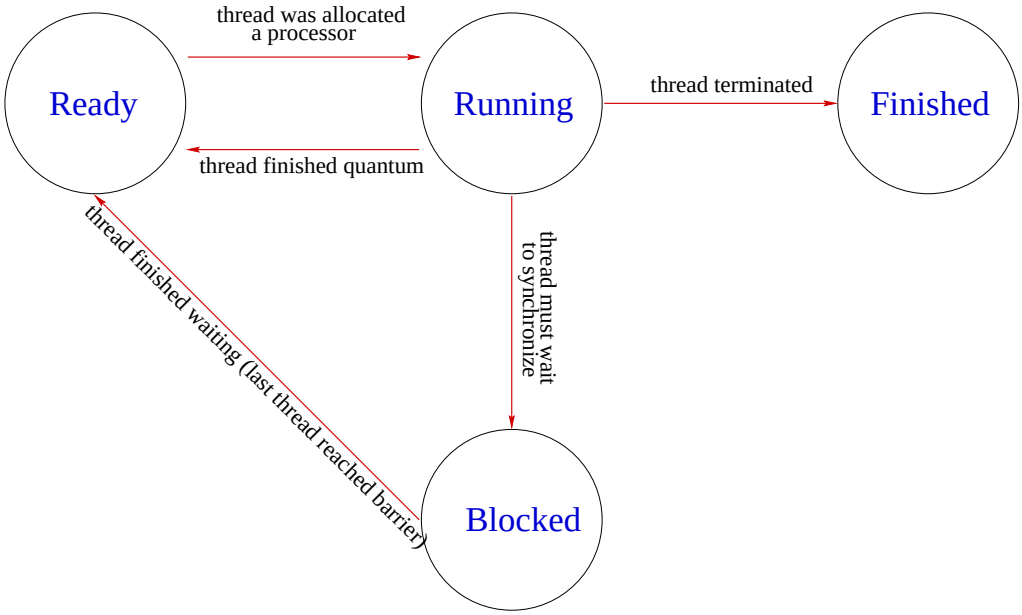

Figure 2.1: possible states and state-switches for a synchronizing thread. The diagram associated with non-synchronizing threads is similar, with the difference that the 'blocked' circle and the arrows attached to it should be removed.

We remark the contention due to synchronization was not simulated. This is a reasonable simplification when assuming that a barrier completion time (with contention) is significantly shorter than a context switch duration. This work focuses on SMP systems for which this is indeed the case. Chapter 6 will demonstrate that the popular general purpose operating systems with priority based schedulers (like Linux), are such systems.

Each execution of the simulator gets as input a configuration file which describes the various SMP parameters (e.g. how many processors it has, how long is a context switch etc.) and the parameters of the jobs it executes (e.g. granularity, number of barriers, number of threads etc.). The configuration file is described in section 2.2.

The output of the simulator is a table describing how well the synchronization policy performed, as a function of the load on the SMP. The output is described in section 2.3.

2.2 The Configuration File: Simulator's Input

The configuration file specifies all the information necessary for the simulator to simulate a complete SMP run and output a "graph" describing the results of this execution. There are three types of lines acceptable in the configuration-file:

1. **Global lines**: These type of lines may contain specification of general properties of the output graph such as the title etc. (not significant).

2. **Curve lines**: Each such line describes one curve in the resulting graph. This line should contain the description of the machine: the number of processors composing the SMP, the quantum duration, how long it takes to perform a context switch etc. Each curve line is followed by one or more synchronizing-job lines.

3. **Synchronizing-job lines**: Each such line relates to the curve-line that preceded it. Such a line describes a profile of a synchronizing job (size, granularity etc) within the mix of jobs that will eventually be displayed as one curve in the graph.

Each line is composed out of a sequence of tokens in the format *Parameter=Value* separated by white space. As the simulator evolved, a large number of acceptable parameters was defined. Table 2.1 specifies the more important parameters that may appear in a configuration input file and are used throughout the following chapters.

#	parameter name	mandatory	percent	description
curve-line parameters				
1	`p`	*		Processor number.
2	`q`	*		Cycles per quantum.
3	`in`	*	*	Cycles per context switch - in (wakeup).
4	`out`	*	*	Cycles per context switch - out (preempt).
5	`nosync`	*		Number of non-synchronizing jobs (each with one thread).
6	`rand_ord`			Nonzero if the threads in the 'ready-queue' should be shuffled beforethe execution begins, zero otherwise. If this parameter is not given than noshuffling will be performed.
7	`job_configs`			The number of sync-job-lines following the curve-line. These lines willspecify the synchronizing-job mix which will participate in the simulationdefined by this curve-line.
synchronizing-job-line parameters				
8	`sync`	*		Number of threads composing the synchronizing job.
9	`barrier`	*		Number of barriers performed by the threads of the specified job.
10	`spin`	*	*	Maximum number of cycles for a thread to spin while tryingto synchronize (before entering the blocked-state).
11	`cmput`	*	*	The expectation - μ - (in cycles) of the time interval a thread "computes"between barriers. The computation-intervals are **normally distributed**around this value.
12	`rand_cmput_sigma`			The standard deviation - σ - of the normal distribution used to generate computation intervals.σ can be specified in various formats(see section 2.2.1 for detailed description).If this parameter is not given it is assumed to be zero.
13	`instance_num`			Number of instances of the job with the profile defined in the current sync-job-line.If this parameter is not given it is assumed to be 1.

Table 2.1: Simulator's main parameters.

- All parameter marked with 'percent' may be followed by a '%' sign which means the associated value specifies a percent out of the quantum interval (value associated with `q`).
- All parameter marked with 'mandatory' must be specified.

2.2.1 The σ Used to Generate Computation Intervals

This subsection describes the acceptable formats of σ, the value associated with the parameter `rand_cmput_sigma`. If this parameter appears, it means computation intervals of the synchronizing job represented by the sync-

job-line should be *normally distributed* around μ (the value associated with `cmput`). The associated value may be given in 2 formats:

1. *Direct-format*: Only one floating value is given. This value is the actual σ of the normal distribution used (must be positive).
2. *Symbolic-format*: The value must be in the format: DENSITY/INTERVAL
   - *DENSITY*: A float from the open domain: (0,100) specifying the *percentage* of the points to be found in the interval which is defined as follows:
     Λ = [μ-INTERVAL , μ+INTERVAL]
   - *INTERVAL*: A number which defines the domain of Λ as described above. It must be positive. In addition, if this number is followed by '%' it means the number is given as percentage out of μ (the value associated with `cmput`) in which case it must be in the open domain: (0,100). Note that in any case the lower bound of Λ must be positive.

If this parameter doesn't appear in the job-config-line then the computation interval is always μ.

2.2.2 Converting σ's Symbolic-Format to Direct-Format

Let X be a random variable such that $X \sim N(\mu, \sigma)$. Given $p = \frac{DENSITY}{100}$ and $z = INTERVAL$, we want to compute the value of σ. According to the definition of p and z there exists:

$$Pr(\mu - z \leq X \leq \mu + z) = p$$

$$\Rightarrow Pr(-z \leq X - \mu \leq z) = p$$

$$\Rightarrow Pr\left(\frac{-z}{\sigma} \leq \frac{X-\mu}{\sigma} \leq \frac{z}{\sigma}\right) = p$$

$$\Rightarrow Pr\left(\frac{-z}{\sigma} \leq Z \leq \frac{z}{\sigma}\right) = p \text{ where } Z \sim N(0,1) \text{ i.e } Z \text{ is normalized,}$$

$$\Rightarrow Pr\left(Z \leq \frac{z}{\sigma}\right) - Pr\left(Z \leq -\frac{z}{\sigma}\right) = p$$

and because of the normal distribution symmetry:

$$\Rightarrow 1 - 2Pr\left(Z \leq -\frac{z}{\sigma}\right) = p$$

$$\Rightarrow 1 - 2k = p \text{ where } k = Pr\left(Z \leq \frac{z}{\sigma}\right),$$

$$\Rightarrow k = \frac{1-p}{2}$$

Since p is given then k is know. Now, we may look for the value of $y = \frac{-z}{\sigma}$ such that $k = Pr(Z \leq y)$ using the *unit normal distribution's area table*. Finally, we may compute σ using the formula $\sigma = -\frac{z}{y}$.

2.3 The Output Of the Simulator

The output of the simulator is a table specifying how well the synchronization policy performed as a function of the load on the SMP. The following subsections will describe this output in detail.

2.3.1 The SSR Metric

Throughout this work, we will use the *successful-spin-rate (SSR)* in order to evaluate the advisability of spinning. This metric is defined to be the percentage of cases in which a process succeeds to synchronize

while spinning, excluding the last one to arrive. More formally it is:

$$SSR = \frac{\sum_{t \in ST} successfulSpin(t)}{\sum_{t \in ST} totalSpin(t)} \times 100$$

where:

- ST is a group containing all the **S**ynchronizing **T**hreads participating in the simulation.
- $successfulSpin(t)$ is the number of times *t* did not block after spinning (because all the threads of its job have reached the barrier while it was spinning). This number does not include the times *t* was the last thread of its job to reach a barrier, since no spinning was performed. The last thread to reach a barrier always succeeds, we are not interested in these cases.
- $totalSpin(t)$ is the number of times *t* entered a spin mode. Again, this number does not include the times *t* was the last thread of its job to reach a barrier.

As a rule of thumb, if the SSR is smaller than 50%, we will consider spinning as not worth while, because threads failed more than they succeeded to synchronize due to spinning.

2.3.2 Elapsed Time

We remark that SSR is not a perfect metric and should be used carefully. For example, if the always-spin waiting algorithm is used, jobs executing on a preemptive scheduler (which is what we use in this work) will always achieve SSR of 100%. This subject will be further elaborated later on. Throughout chapters 3-5 we will solely use the SSR metric in order to evaluate the profitability of spinning and to gain understandings of the computation patterns of barrier based applications. However, throughout chapters 7-9 we will also expend our discussion to include applications' elapsed execution time, and we will establish a firm connection between speedup and high SSR.

2.3.3 The Load

Usually, when we discuss SSR (or speedup, when comparing various waiting algorithms) it is associated with or displayed as a function of load. Various aspects of load are considered. For example, in chapter 3, $|ST|$ is constant, and therefore load is usually associated with $|NST|$ (*NST* is defined to be the group that contains all the **N**on **S**ynchronizing **T**hreads participating in the simulation). Another example is the size of the set $THREADS = ST \bigcup NST$, which defines the *total load* on the SMP. This measure is usually used when we discuss heterogeneous job collections.

Chapter 3

Synchronizing Job in a Non-Synchronizing Environment Under the Round-Robin Scheduler

3.1 Introduction

This chapter describes the behavior of a single synchronizing job in a very simple scenario: the job is "disturbed" by an increasing number of non-synchronizing threads. We will show and explain the effects of adding two types of randomization to such a simulation:

1. shuffling the order of the threads in the ready-queue on startup, and
2. adding imbalance by normally distributing the computation intervals of the synchronizing threads (as explained in section 2.2.1).

We will establish a connection between the SSR (successful spin rate), the length of the spinning-interval and the σ-interval (as defined in 2.2.1). We will also introduce the *alternating synchronization* concept.

The X axis of all the graphs presented in this chapter are the number of non-synchronizing threads that participated in the simulation.

3.2 Simulation with No Randomization

In this section we will analyze a very simple scenario of some simulation in which there is no randomization. The purpose of this section is to allow us to get familiar with the subject of barrier based application in a loaded environment, as fast as possible.

The parameters used in the simulations are:

p	q	in	out	sync	nosync	barrier	spin	μ (cmput)
32	100	3%	3%	11	0...200	50	6%	1%,10%,100%

The chosen length of the quantum is arbitrary, it has no meaning on its own outside the context of the other parameters. It was chosen to allow easy conversion from parameters that are expressed as percentage of a quantum, to the actual number of cycles they represent. The total duration of context switch (6% of quantum) may seem (and is probably) too long. However, aside from being considerably shorter than a quantum duration, it too has no actual meaning of its own in the context of this work. Its real importance lies in the manner we classify granularity of jobs. Roughly, and without (currently) addressing the role of σ, the following specify how we classify jobs' granularity:

1. We usually consider jobs with μ value which is smaller than the duration of a context switch to be *fine grain* (and therefore a job with μ=1% of quantum, is fine grain within this simulation).

2. Jobs for which μ is longer than the duration of context switch but not longer than 4 or 5 times this duration, is classified as being *medium grain* (and thus in this simulation, $\mu = 10\%$ is considered as medium granularity).

3. Jobs with μ values higher than that, are considered *coarse grain* (e.g. $\mu = 100\%$ is coarse).

The resulting graph is presented in figure 3.1.

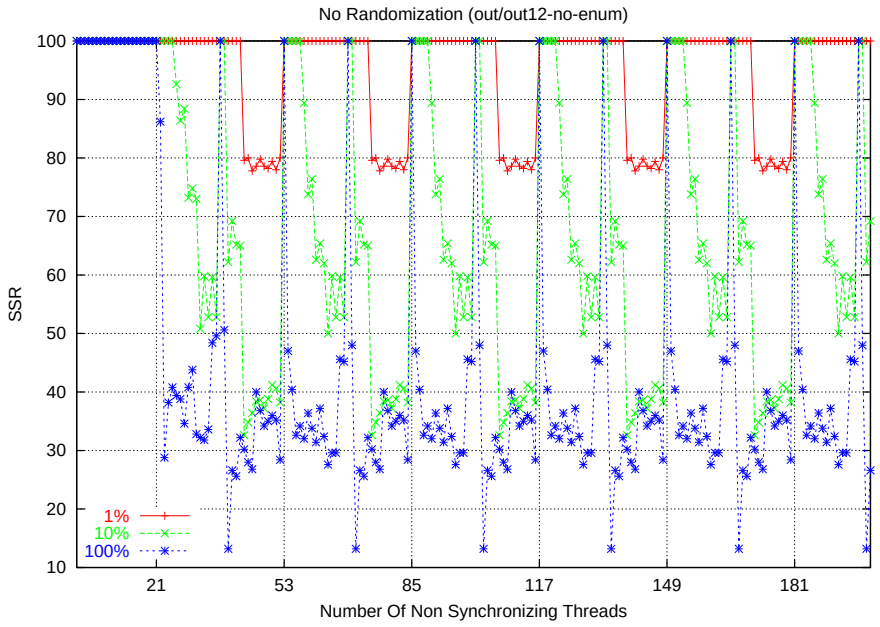

Figure 3.1: The result of a simple simulation with no randomization. The fine-grain jobs ($\mu = 1\%$, red line) achieve the best SSR. The coarse-grain jobs ($\mu = 100\%$, blue line) achieve the worst SSR. The SSR achieved by the medium-grain jobs ($\mu = 10\%$, green line) is somewhere in between.

3.2.1 Analyzing the Fine-Grain Job's Behavior: $\mu = 1\%$

The fine-grain job's behavior is presented by the red line in figure 3.1. This job must synchronize each 1% out of quantum i.e. each cycle (because the quantum was taken to be 100 simulator cycles). The following subsections constitute a detailed analysis of the behavior of this job.

3.2.1.1 The First Quantum

To understand the graph we must first understand what happens in the first quantum from the point of view of - *J* - the synchronizing job:

- On startup *J*'s threads occupy the beginning of the ready-queue [1] .
- Immediately after the execution begins, ALL the threads are allocated processors and move to running-state (because *J*'s size is 11 and there are 32 processors).

[1] This is true because the simulator decides the startup order of the threads in the ready queue according to the ordered pair <job-id, thread-id> and the id of J in this simulation was chosen to be 0.

- By the time the first quantum ends, *J*'s threads have finished the 33rd barrier. This is true because:
  - It takes one cycle for each thread to reach the "next" barrier.
  - It takes two cycles for the job to synchronize: all the threads try to synchronize; only the "last one" succeeds; the other threads start to spin and after one cycle (their first "polling") they succeed and continue computing ...
  - considering the above, the number of barriers performed in the first quantum is approximately:

$$\frac{cycles\,per\,quantum}{1(compute)+2(synchronize)}=\frac{100}{3}\approx 33$$

The above scenario is not dependent on the number of non-synchronizing threads participating in the simulation, what happens next however, does.

3.2.1.2 The Second Quantum for $|NST| = 1...42$

Recall that NST was defined in section 2.3.3 to be the group of non-synchronizing threads participating in the simulation. We will now show that for $|NST| = 1...42$, all of *J*'s threads will immediately be allocated a processor after returning to the ready-queue. It's enough to show this for $|NST| = 42$ and the argument used may be applied to $|NST| = 1...41$ in a similar manner.

Along with *J*'s threads there are 21 non-synchronizing threads that have also finished their quantum. The number of ready threads found in the ready queue after startup and before the first context switch is:

$total\,thread\,num - 32(running) = \langle 42(nosync) + 11(sync)\rangle - 32(running) = 21$

This means that after the first quantum ends and these 21 threads get allocated a processor, there are $11 = 32 - 21$ vacant processor to use, just enough for all of *J*'s threads. Unsurprisingly these processors are indeed allocated to them [2] . Since *J*'s threads will complete all the barriers in the second quantum (we chose 50 barriers for the simulation), the simulation will end with 100% SSR.

3.2.1.3 The Second Quantum and Onwards for $|NST| = 43...52$

The above "ideal" ends of course when $|NST| = 43$. In this situation there aren't enough processors to allocate for all of *J*'s threads: ten threads get allocated processors but the eleventh remains in the ready queue. As a result, the jobs fall into an *alternating synchronization pattern* in which the processes in the job occur as two contiguous groups in the ready queue. The first group reaches the barrier, spins, and blocks. When the second group runs, the barrier is completed, and all the processes in the first group are released again into the ready queue and so on. Alternating synchronization is discussed in detail in section: 4.3. As a result of the alternating synchronization, the SSR is reduced.

The argument explaining why *J* is divided to two thread-groups (of the sizes: 10-running + 1-ready) for $|NST| = 43$ after the first quantum, may also be applied to $|NST| = 44...52$:

- $|NST| = 44$: the job is divided to thread-groups: 9-running + 2-ready
- $|NST| = 45$: the job is divided to thread-groups: 8-running + 3-ready
- ...
- $|NST| = 52$: the job is divided to thread-groups: 1-running + 10-ready

The SSR achieved for $|NST| = 43...52$ is further discussed in section 3.2.1.5.

[2] This is true because J's threads occupied the beginning of the ready queue on startup. Therefore they were allocated processors first. Therefore they were the first to have finished the first quantum. Therefore they are again occupying the beginning of the ready queue.

3.2.1.4 The Cycle of the Simulation: $|NST| \geq 53$

From the point when $|NST| = 53$ - which means that the number of threads participating in the simulation is $|THREADS| = |NST| \cup |ST| = 64$ - we notice the simulation enters a cycle. After the first quantum finishes there are 32 non-synchronizing threads waiting in the ready queue and therefore synchronizing threads do not get allocated a processor. This means that the second "chunk" of 32 threads executing on the SMP doesn't influence the resulting graph at all, they simply consume one complete quantum. After the 2nd quantum is over, the other 32 threads (*J*'s 11 threads + the remaining 21 non-synchronizing threads) will get a processor and the simulation will behave exactly is it did when $|NST|$ was 21 (i.e. when $|THREADS|$ was 32) .We conclude that for $c = 1, 2, 3...$ and $i = 0, 1, \ ... \ , 31$ there exists:

$$SSR\left(|NST| = 21 + i\right) \quad = \quad SSR\left(|NST| = 21 + i + 32c\right)$$

or in other words:

$$SSR\left(|THREADS| = 32 + i\right) \quad = \quad SSR\left(|THREADS| = 32 \cdot (1 + c) + i\right)$$

The cycle is demonstrated in figure 3.2.

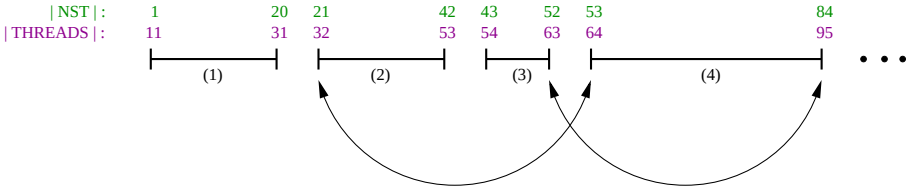

Figure 3.2: The various stages of *J* in the simulation and the simulation's cycle:

(1) The SMP is not fully utilized, there are more processors than threads.

(2) $|THREADS|$ isn't smaller than the number of processors but is small enough for *J*'s threads to be allocated processors immediately after returning to the ready queue (in the beginning of the second quantum).

(3) The load is big enough to cause *J* to alt-synchronize: in the begin of the second quantum only a portion of *J*'s thread are allocated processors.

(4) The simulation's cycle.

3.2.1.5 Increasing the Number of Barriers

The sections above imply that the high SSR of the fine-grain simulation (always more than 77%) was achieved purely because of the relatively low number of barriers that was performed by *J* and the relatively big weight the successful spins of the first quantum have. After this initial "grace period", the simulation starts to perform alternating-synchronization and the SSR drops below 50% (this 50% boundary is explained in section: 4.4). Therefore increasing the number of barriers should reduce the weight of the grace period and cause the SSR to decline. Figure 3.3 demonstrates this.

3.2.2 Analyzing the Medium and Coarse Grain Jobs' Behavior: $\mu = 10\%, 100\%$

All the arguments that were applied to analyze the case in which μ - the computation interval - is 1% of quantum, also hold for μ=10% and μ=100%. For μ=10% the difference is that by the time the first quantum ends, *J*'s threads have finished only 8 barriers or so because:

$$\frac{cycles\ per\ quantum}{10(compute) + 2(synchronize)} = \frac{100}{12} \approx 8$$

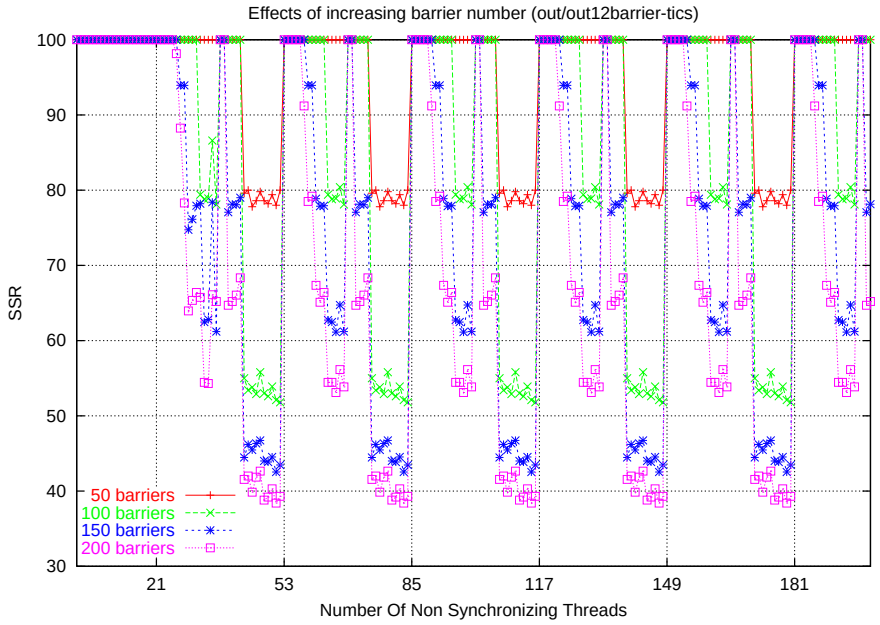

Figure 3.3: This figure shows the effects of increasing the number of barriers in the fine-grain simulation. It is obvious the more barriers there are, the lesser the SSR gets since the weight of the grace period diminishes.

For μ=100%, *J* didn't finish even one barrier in the first quantum.

The curve associated with μ=1% appears to present a far better SSR than the curves associated with μ=10% and μ=100%. But, as explained in section 3.2.1.5 (and demonstrated in figure 3.3), this happens due to the relatively small number of barriers in the simulation, causing the weight of the successful spins in the first quantum to be much smaller for 10% and 100% than for 1%.

To further demonstrate this, figure 3.4 shows the result of three simulations identical to those displayed in figure 3.1 (in the beginning of the chapter) with the difference that now, more than 50 barriers are used. By increasing the number of barriers we decrease the weight of the first few quanta successful spins and increase the weight of the unsuccessful spins caused by alternating synchronization. As expected, as we increase the number of barriers, the graph shows that the curve associated with μ=1% gets closer to the other curves.

3.3 Randomizing the Order of the Ready-Queue

This section describes the results of a simulation which is identical to the previous simulation we discussed (defined in the beginning of section 3.2) , with the difference that the order of the threads in the ready-queue was shuffled on startup. Figure 3.5 presents the result of this simulation.

The main effect of this shuffling was that the scenario of the first few quanta in the simulation - which was described in detail in the previous section - didn't happen: Since the threads aren't continuously ordered in the ready queue, *J* enters an alternating synchronization pattern from the beginning of the simulation and doesn't get the "grace period" as before. As the figure shows (and as expected), the results are fairly similar for the different computation intervals.

3.4 Adding Variability to the Computation Intervals

This section describes the results of a simulation identical to the one described in the previous section (in which we shuffled the order of the ready queue on startup), with the difference that now we add imbalance

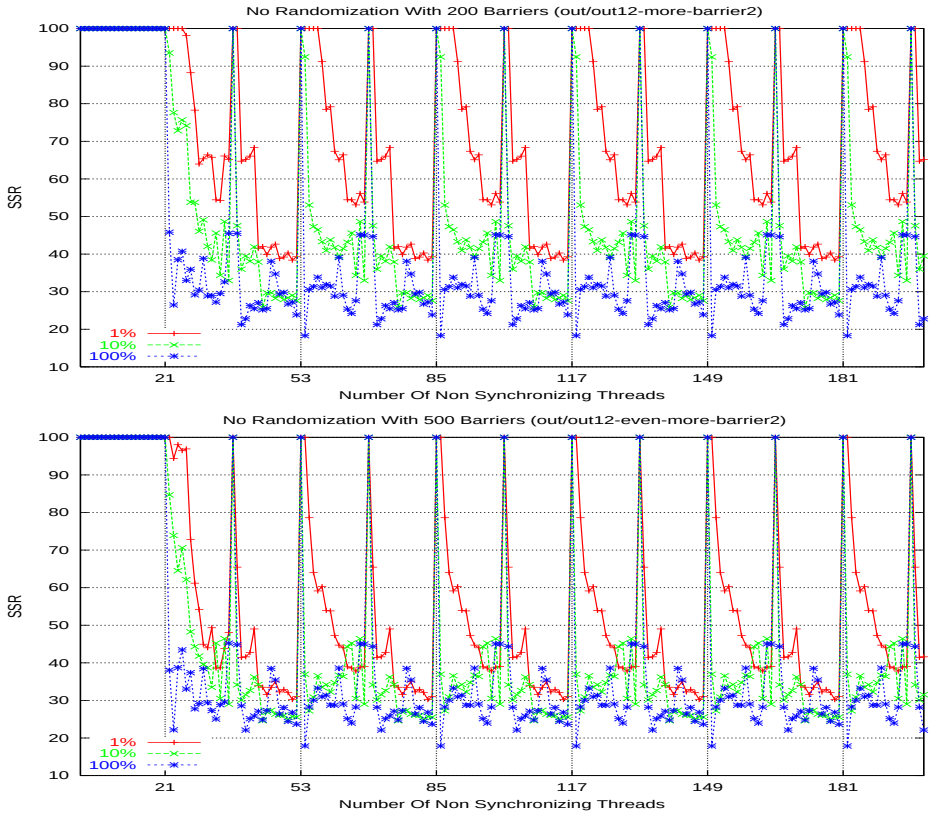

Figure 3.4: As we increase the number of barriers, the difference between the SSR achieved by fine grain jobs, gets closer to the SSR achieved by coarse grain jobs.

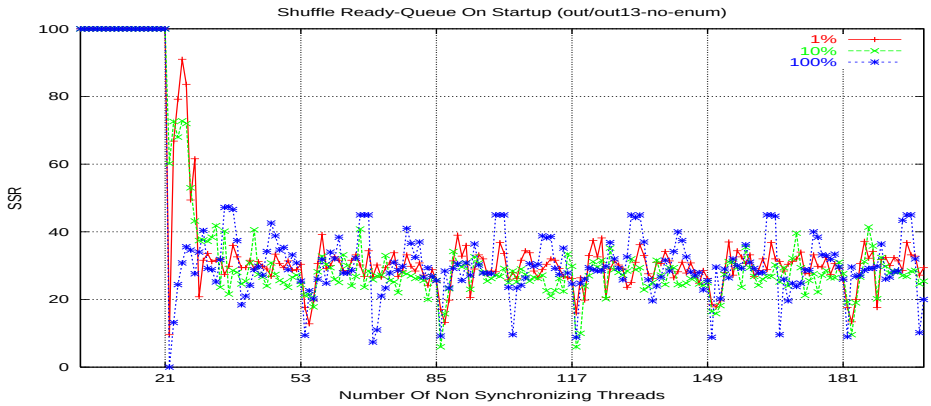

Figure 3.5: The results of the simulation after shuffling the ready queue on startup: no grace period causing the SSR to be low from the moment the number of running threads is bigger than the number of processors.

to the simulation by normally distributing computation intervals of the synchronizing threads (as explained in section 2.2.1). Figure 3.6 presents the result of this simulation. The computation intervals of the synchronizing threads were normally distributed with σ**=90/15%,** which means 90% of the computation intervals fall into [85%...115%] out of μ (the value associated with the `cmput` parameter). Making the length of the computation interval random, influences its values as follows:

- For 1-cycle-computation-interval (computation interval is 1% of quantum, quantum is set to be 100 cycles), 90% of the values will be in the range $[0.85...1.15]$. Since a computation interval represents a number of cycles, it must be discrete and therefore the values are always (or with probability close to 1) rounded to 1.
- For 10-cycle-computation-interval (i.e. computation interval is 10% of quantum), 90% of the values will be in the range $[8.5...11.5]$ which means that the effective values are in $[9...11]$.
- For 100-cycle-computation-interval (i.e. computation interval is 100% of quantum) the effective values are in $[85...115]$.

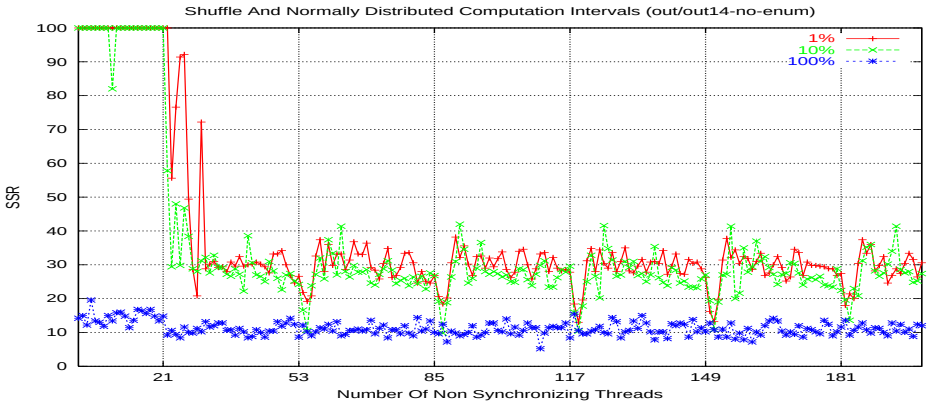

Figure 3.6: Adding imbalance to the simulation by normally distributing computation intervals around μ with σ=90/15%, resulted only in the decline of SSR associated with coarse grain jobs. The SSR of fine and medium grain jobs have (more or less) stayed the same.

The figure shows that the results for computation-interval 1% and 10% stayed more or less the same while the results for computation interval 100% have declined. To make sure the reason for the curves associated with μ=1%, 10% have stayed the same wasn't the discrete nature of the simulation, we ran the same simulation but with 10000-cycles-quantum instead of 100. Figure 3.7 shows that the results of this simulation (10000-cycles per quantum) which are almost identical to the results displayed in figure 3.6 (100-cycles per quantum). This fact verifies our hypothesis.

3.4.1 The Reason Why Only Coarse-Grain Jobs Were Affected By the Random Distribution of Computation Intervals

Let the quantum be 10000 cycles. The spinning interval in this case is 600 cycles (6% out of quantum). Let's examine the computation interval expectation (μ) and effective values, relatively to the σ-interval (= 90/15%):

μ		range of 90%	σ-interval	$\frac{\sigma\ interval}{spin\ interval}$
out of quantum	in cycles	of the values		
1%	100	$[85...115]$	30	0.05
10%	1000	$[850...1150]$	300	0.5
100%	10000	$[8500...11500]$	3000	5

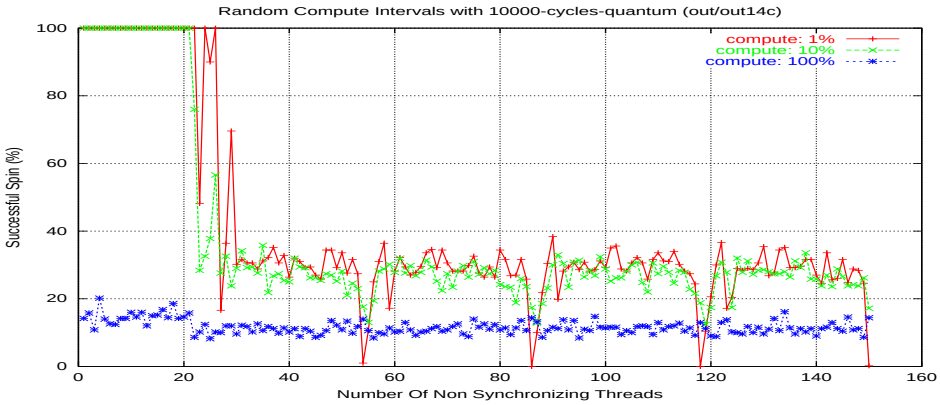

Figure 3.7: This figure presents the results of a simulation identical to the one presented in figure 3.6. The difference between them is that in this simulation the quantum used was 10000 cycles instead of 100. Since the two figures are almost identical, it proves that the discrete nature of the simulation wasn't the reason why only the coarse-grain jobs (blue line, μ=100%) was affected by the normal distributing of computation interval's length.

For coarse-grain jobs (μ=100%), the σ-interval is much bigger than the spin-interval. On the other hand, for fine and medium-grain jobs (μ=1%,10%), the σ-interval is smaller than the spin-interval. For a coarse-grain job, if we get two threads with different computation intervals that are far enough apart (say the first is near 8500 cycles and second is near 11500 cycles) then obviously the first thread will spin unsuccessfully. The σ-interval (= 3000 cycles) is big enough compared to the spin-interval (= 600 cycles) to make most of the spins unsuccessful. This is the reason why only the curve associated with coarse-grain jobs was affected by the random distribution of computation intervals.

Most of the SSR associated with the coarse-grain jobs is found around 10% which means that for each barrier, on average, only one out of 10 threads that actually spun performs successful spinning (the eleventh thread to reach the barrier doesn't count since it doesn't perform spinning) .

We will now demonstrate the principal stated above in the next section using a few simulations ...

3.4.2 Various Computation-Intervals with 15% σ-Interval

In this simulation we choose the σ-interval to be 15% out of various μ values. We will show (for a constant spinning-interval) that increasing μ - which means increasing the σ-interval - causes the SSR do decline.

The parameters of the simulations are:

p	q	in	out	sync	nosync	barrier	spin	rand_ord	μ (in percents out of quantum)	σ
32	1000	3%	3%	11	0...70	50	6%	1	20,25,30,35,40,45,50,60,70,90	90/15%

The σ-intervals and the effective values for the various computation-intervals are displayed in table 3.1.

The result of the simulation is displayed in figure 3.8. The figure strengthens the hypothesis stated above. It is clear that for bigger σ-intervals (with respect to the spinning-interval), lower SSR is achieved. Notice also the implied continuity - demonstrated by the figure - between the medium (μ=10%) and coarse (μ=100%) grain curves displayed in previous figures.

μ out of quantum	μ in cycles	range of 90% of the values	σ-interval	difference from spin-interval	$\frac{\sigma\ interval}{spin\ interval}$
15%	150	[127.5...172.5]	45	-15	0.75
20%	200	[170...230]	60	0	1
25%	250	[212.5...287.5]	75	+15	1.25
30%	300	[255...345]	90	+30	1.5
35%	350	[297.5...402.5]	105	+45	1.75
40%	400	[340...460]	120	+60	2
45%	450	[382.5...517.5]	135	+75	2.25
50%	500	[425...575]	150	+90	2.5
60%	600	[510...690]	180	+120	3
70%	700	[595...805]	210	+150	3.5
80%	800	[680...920]	240	+180	4
90%	900	[765...1035]	270	+210	4.5

Table 3.1: σ-interval values and computation interval effective values as a function of μ. Recall the the spin interval is 60 cycles (6% of the quantum which is 1000 cycles).

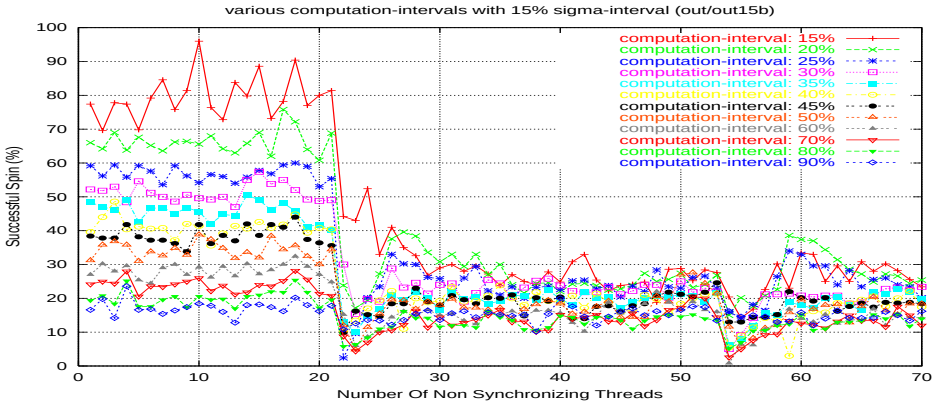

Figure 3.8: This figure shows 12 simulations with various computation-interval. The σ-interval is 15% of the computation-interval which means that if the computation-interval gets bigger then so does the σ-interval. The figure proves that for bigger σ-interval a lesser SSR is achieved.

3.4.3 Various σ-Intervals with a Constant Computation-Interval

The simulations performed in this section are similar to those performed in the previous section with the difference that now the σ-intervals changes with respect to a constant μ. We expect of course that for bigger σ-intervals we will get lower SSR.

The parameters of the simulations are [3] :

p	q	in	out	sync	nosync	barrier	spin	rand_ord	μ	σ
32	1000	3%	3%	11	0...70	50	6%	1	5% 10% 100%	90 / 20-100:20 % 90 / 5-55:5 % 90 / 5-55:5 %

The result of the simulation is displayed in figure 3.9. This figure also strengthen the hypothesis stated above. Again, it is clear that for bigger 'σ's, lower SSR is achieved.

[3] The meaning of: x-y:j is the list of values that starts from x, ends in y with a difference of j between each two successive elements in the list.

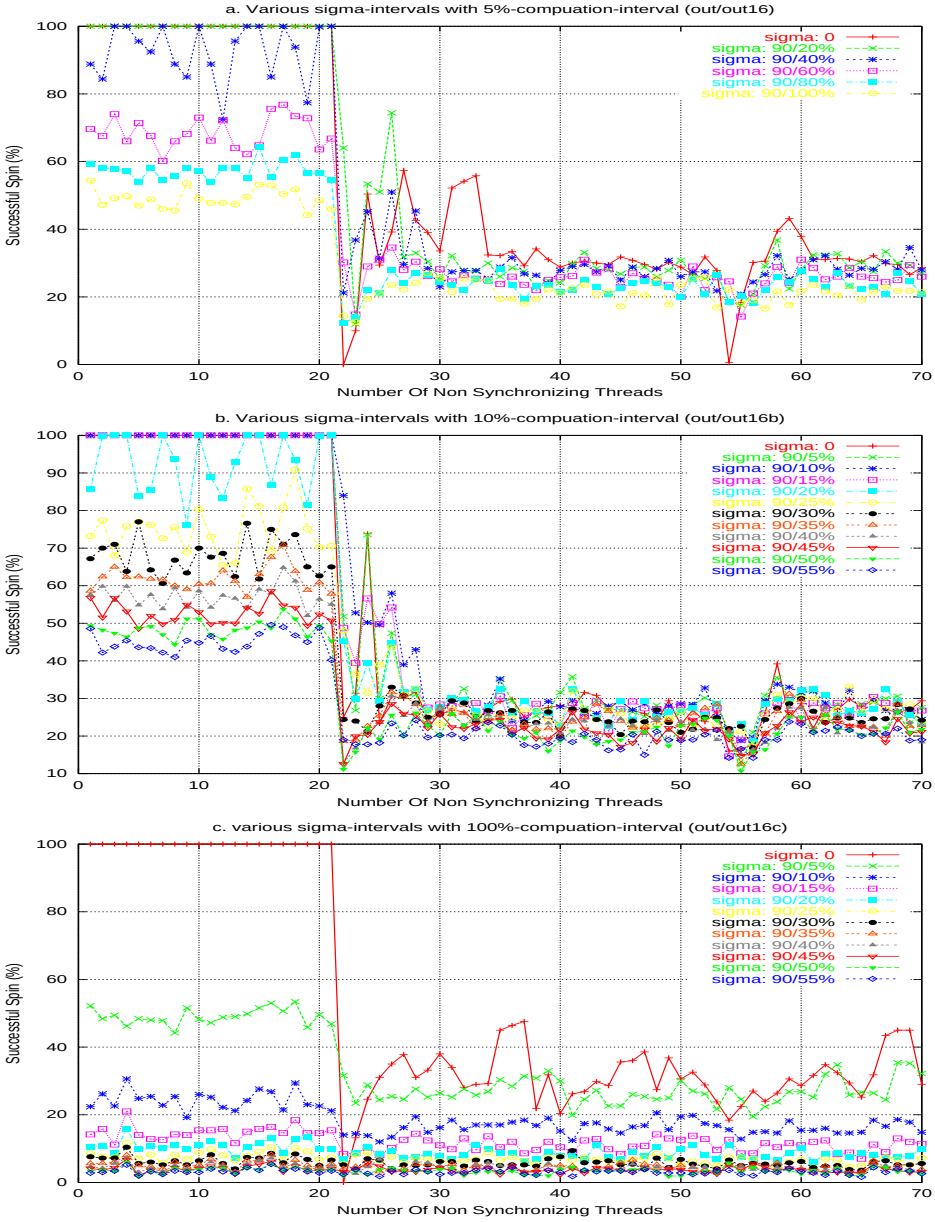

Figure 3.9: Figures a,b,c show simulations with μ=5%,10%,100% respectively: we can see for each μ that for bigger σ (interval) a lesser SSR is achieved.

Chapter 4

Homogeneous Collection of Synchronizing Jobs Under the Round-Robin Scheduler

4.1 Introduction

This chapter describes the behavior of a collection of homogeneous synchronizing jobs and explains the *alternating synchronization* concept mentioned in the previous chapter. Each simulation contains an increasing number of synchronizing jobs with an identical profile. We will show that such simulations behave more or less the same regardless of the chosen size of the synchronizing jobs and (from some point) regardless of the load, due to the alternating synchronization effect. There are no non-synchronizing-jobs participating in the simulations described in this chapter. The X axis of most graphs presented in this chapter displays the number of synchronizing jobs participating in the simulation.

4.2 Simulation and Results

The parameters Used in the simulations are:

p	q	in	out	sync	nosync	barrier	spin	μ	σ	rand_ord
32	100	3%	3%	2, 3, 4, 5, 10, 11, 15, 16, 22, 25, 32	0	50	6%	1%,10%,100%	90/15%	1

The resulting graphs are presented in figure 4.1.

4.3 Alternating Synchronization

4.3.1 Motivation

When examining figure 4.1, an immediate question that comes to mind is why do curves associated with μ=1%,10% converge to a fairly high value regardless of the load ? For instance, how come there's no difference between the SSR achieved by a system running 15 jobs of the size 10 (150 threads) and the SSR achieved by a system running 150 such jobs (1500 threads). This result is quite surprising as we

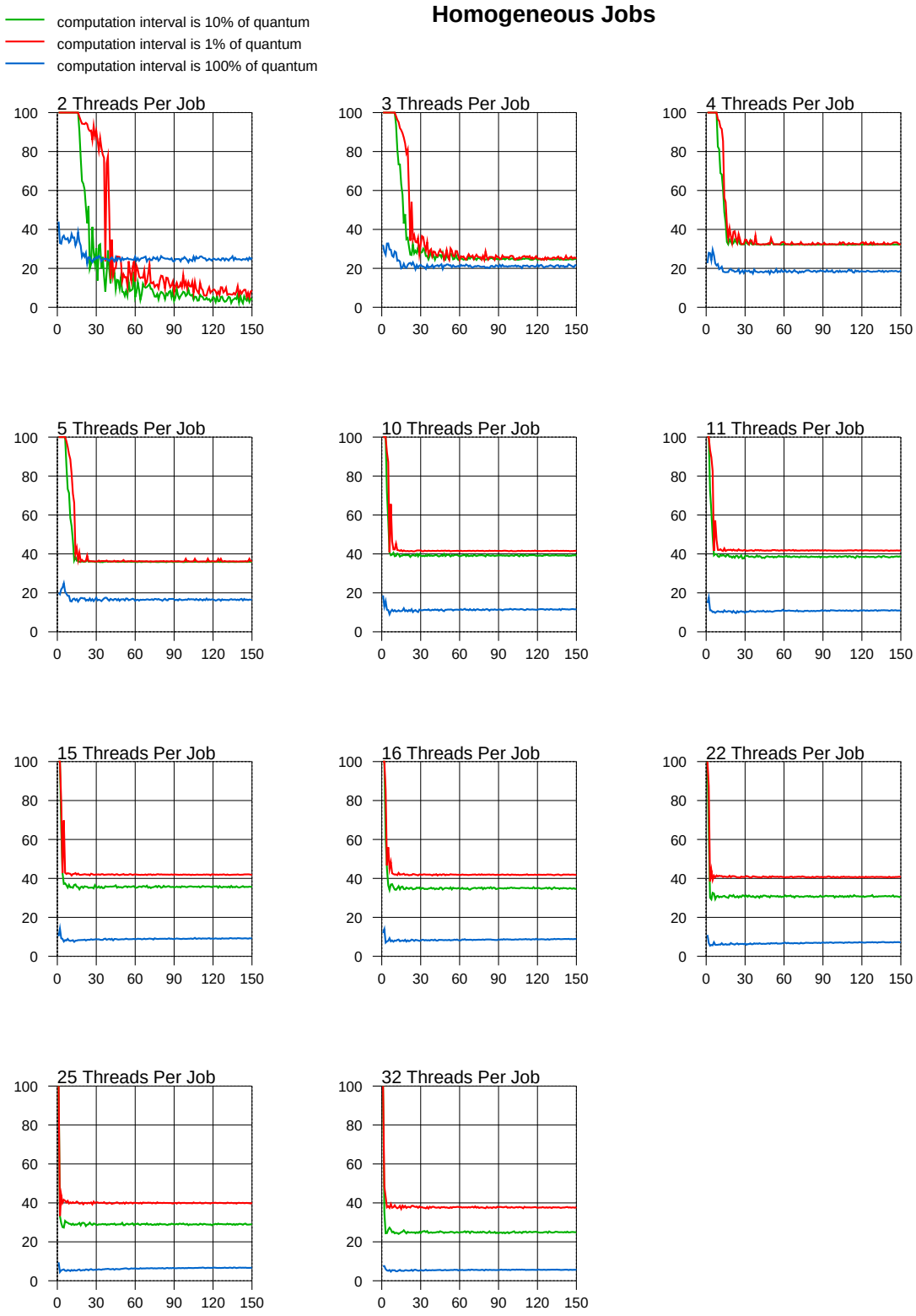

Figure 4.1: The **X axis** specifies the number of jobs participating in the simulation. Jobs' size is specified in the graph's title. The **Y axis** is the SSR. We notice that **(1)** the lines associated with μ=1,10% start at 100% SSR and **(2)** drop until they converge to some value below 50% but above 25% (except for jobs composed of two threads that converge to zero, the difference is explained in section 4.3.5).

expected that more jobs participating in the simulation will result in smaller SSR. As mentioned before - with the exception of jobs composed out of two threads - the SSR is non negligible both for 1% and for 10% computation intervals. Table 4.1 specifies the SSR to which the various curves converge.

Job's size	μ 1%	10%	100%
3	25	24	20
4	32.4	32.1	18.5
5	36.1	35.9	16.5
10	41.5	39.1	11.5
11	41.7	38.5	10.9
15	41.9	35.6	9.2
16	41.8	34.8	8.8
22	40.7	30.8	7.2
25	39.9	29.9	6.6
32	37.6	24.9	5.5

Table 4.1: The SSR to which the various simulation converge.

4.3.2 Threads' Dispersal in the Ready Queue

The first thing we did when we tried to understand this phenomenon was to check how do threads of some arbitrary job - J - "scatter" in the ready queue. The simplest thing to do was to measure the distance $Dist(J)$ between T_1 and T_2 where T_1 is J's thread which is closest to the tail of the ready queue and T_2 is J's thread which is closest to the head of the ready queue.

For example: in the following figure, J is composed out of four threads: T_1, T_2, T_3 and T_4, all of them are in the ready queue. In this example $Dist(J) = 8$:

The Ready Queue

Tail		T1	H1	H2	T3	H3	H4	T4	H5	T2	H6	H7		Head

Distance between T1 and T2 is 8

We've changed the simulation such that for μ=1%,10%, whenever **ALL** of J's threads are found in the ready queue, $Dist(J)$ will be printed. To our surprise, the simulation didn't print even a single number. This of course means that there's no time instance in which all of J's threads are in the ready queue simultaneously (with the exception of the beginning of the simulation). We refined the simulation such that whenever there's a change in the dispersal of J's thread among the various SMP's states, it will print the pair: $<Dist(J), |RQ(J)|>$ where:

$$RQ(J) = \{J's\ threads\ found\ in\ the\ ready\ queue\}$$

We found that if we define $Dist(J)$ to be (-1) when $RQ(J) = \emptyset$, through out all the simulation (except on startup) the following invariant holds:

$$Dist(J) - |RQ(J)| \ = \ -1$$

which means the threads in $RQ(J)$ are contiguous in the ready queue at any time instance.

4.3.3 Reason Why $RQ(J)$ is Always Contiguous

After carefully examining the events generated by the simulation, we discovered why jobs are contiguously ordered in the ready queue:

The first step: is to show that the probability that all of *J*'s threads will be allocated a processor on startup is very small:

- Let J be some job in the homogeneous job collection.
- Let α be the above probability (that all of *J*'s threads are allocated a processor on startup).
- Let p be the number of processors in the SMP (in our case $p = 32$).
- Let $j = |J|\,,\ 2 \leq j \leq p$.
- Let k be the number of jobs participating in the simulation.
- Let n be the number of threads participating in the simulation: $n = k \times j,\ n > p$.

Before the simulation begins, the ready queue is shuffled by uniformly choosing a permutation of the threads. This means:

$$\alpha \ = \ \left[\left(\begin{array}{c} p \\ j \end{array} \right) \cdot j! \cdot (n-j)! \right] \cdot \frac{1}{n!} \ = \ \left(\begin{array}{c} p \\ j \end{array} \right) \cdot \left(\begin{array}{c} n \\ j \end{array} \right)^{-1}$$

because $\left[\left(\begin{array}{c} p \\ j \end{array} \right) \cdot j! \cdot (n-j)! \right]$ is the number of permutations in which all of J's threads are found at the p "first" places of the ready queue (i.e. all of them will be allocated processors on startup) and $n!$ is the total number of permutation. By developing the above expression we get:

$$\alpha \ = \ \frac{p!}{j! \cdot (p-j)!} \cdot \frac{j! \cdot (n-j)!}{n!} \ = \ \frac{p \cdot (p-1) \cdot (p-2) \cdot \ldots \cdot (p-j+1)}{n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n-j+1)} \ = \ \prod_{i=0}^{j-1} \frac{p-i}{n-i}$$

which actually also have a combinatorial explanation because the numerator is the number of combinations to arrange j items in (the first) p places and the denominator is the number of combinations to arrange j items in n places. By further developing the above expression we get:

$$\alpha \ \leq \ \left(\frac{p}{n} \right)^j$$

since for $b > a \geq 0,\ r > 0$ there exists:

$$\frac{a+r}{b+r} - \frac{a}{b} = \frac{(a+r) \cdot b - (b+r) \cdot a}{(b+r) \cdot b} = \frac{ab + rb - ab - ra}{(b+r) \cdot b} =$$

$$= \frac{r \cdot (b-a)}{(b+r) \cdot b} = \frac{positive}{positive} > 0$$

which means that $\frac{a+r}{b+r} > \frac{a}{b}$ and therefore we can replace each $\frac{p-i}{n-i}$ in $\prod_{i=0}^{j-1} \left(\frac{p-i}{n-i} \right)$ with $\frac{p}{n}$.[1]

By using the above formula we conclude that in our simulation (for $n \geq 100,\ j \geq 3$) α is smaller than 0.03 and for larger values of *n* (bigger load) and *j* (bigger jobs), α converges to zero.

[1] Actually, we could have approximated α's upper bound to be $\left(\frac{p}{n} \right)^j$ from the beginning, because this expression's combinatorial meaning may be: the probability to choose p items (in our case the "first" p locations in the ready queue) from within a collection of n items (all the possible locations in the ready queue). This probability is bigger than α because it allows the same location to be chosen more than once.

Now, we will explain why $RQ(J)$ is contiguous: Let *A* denote the group of *J*'s threads that were allocated a processor immediately when the simulation began. Assume (without loss of generality) that $A \neq \emptyset$. Also assume that $A \subset J$ i.e. *J* contains *A* but is not equal to it. As stated in the previous paragraph this assumption holds with probability close to 1.

Since μ - the expectations of the computation intervals we're discussing - is 1% or 10% (out of the quantum), *A*'s threads reach the first barrier in the first quantum. They start to spin, but fail because threads in $J \setminus A$ aren't running yet. As a result *A*'s thread move to "blocked" state.

Let T_0 be the last thread of *J* to be allocated a processor for the first time. T_0 starts to compute and soon enough it reaches the first barrier. It immediately succeeds to synchronize with the rest of J's threads as it is the last one to reach the first barrier. Note that currently for every $T \neq T_0,\ T \in J, T$ is either "running" (spins while waiting for T_0) or "blocked".

Let B denote the group of J's threads that are currently in "blocked" state. We know that $B \neq \emptyset$ because $A \subseteq B$ and $A \neq \emptyset$. B's threads move from "blocked" state to the end of the ready queue due to the fact T_0 has reached the first barrier. Of course when B's threads are moved to the end of the ready queue, it is done in a contiguous manner and therefore $RQ(J)$ is continuous.

In the meanwhile T_0 (along with the other threads in $J \setminus B$ if exist) continue to compute and reach the second barrier. Now, because B's threads are currently at the end of the ready queue: $J \setminus B$'s thread start to spin, fail and block. They will remain in this state until B's thread will be allocated processors again.

This scenario repeats itself until J finishes. When J performs this type of computation we say that J performs *alternating-synchronization* or that J is *alt-synchronizing*.

4.3.4 Illustration

Figure 4.2 illustrates alternating synchronization.

The data presented in the figure was taken from a simulation containing 19 jobs where each job is of the size 10 and μ is 10% out of the quantum. The figure shows how threads of three arbitrary jobs are divided between the three states: "ready", "running" and "blocked" as a function of time.

Let's focus on the rectangle associated with J_0 (job number 0). On the simulation's startup all of J_0's threads are in the ready queue. After a while as indicated by the green color (time=22...23) we can see that two threads of the job are allocated CPUs. These threads run for a while, reach the first barrier, spin, fail (since the other threads aren't running yet as indicated by the red color) and go to blocked state as indicated by the blue color (time=44...45). The width of a green stair is approximately 22 cycles:

$$3(context\ switch\ in) + 10(compute) + 6(spin) + 3(context\ switch\ out)$$

This scenario continues to happen as indicated by the green staircase and every time the executing threads end up in the blocked state. This is true until T_0 - the last thread of J_0 - gets allocated a CPU and reaches the first barrier (time=141) in which case all the blocked threads move to the ready queue as indicated by the red "wall".

T_0 continues its computation and reaches the second barrier, spin, fail and block as indicated by the blue "corridor" with the green beginning above the red "wall" (time=141...259).

At time=260...268 the threads in $RQ(J_0)$ reach the head of the ready queue and are allocated CPUs. Note that they don't get the CPUs all at once but rather get them gradually because there aren't nine CPUs available at the same time instance. This is the reason why at time=281...282 two threads join T_0 in the blocked state (the thin blue line) only to move immediately to the end of the ready queue because the last thread of J_0 has finally reached the second barrier.

Note that at time=283...284, three more threads join the three threads that are already at the end of the ready queue (height of the red "wall" at this point is 6 not 3). It happened because the SMP has finished preempting them (due to unsuccessful spinning) but after the context-switch-out ended, the second barrier was already complete, so instead of placing these threads in a blocking mode, the SMP moved them to the end of the ready queue. The five threads that joined T_0 **didn't contribute even one successful spin** at that time.

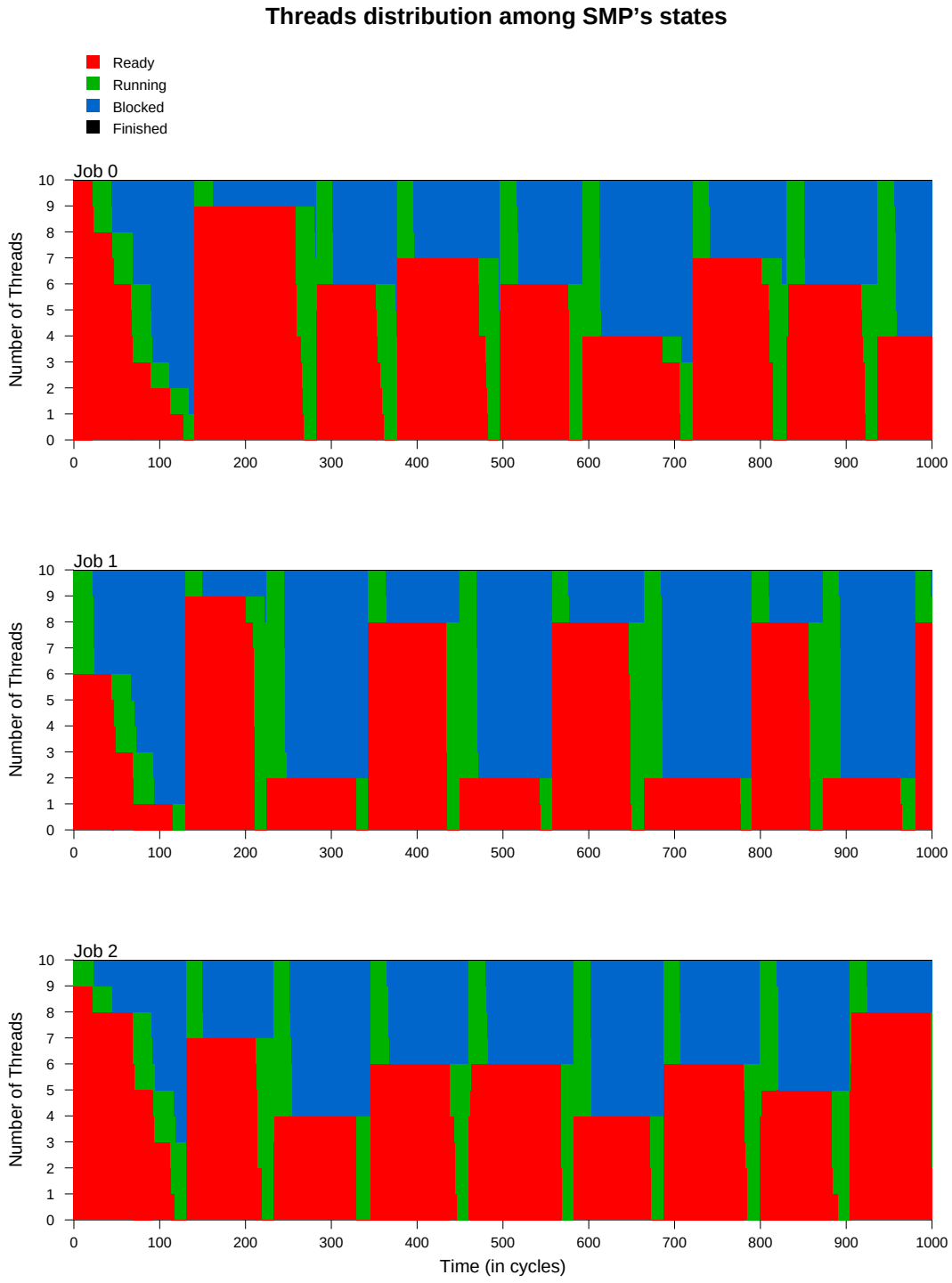

Figure 4.2: This figure describes how the threads of three arbitrary jobs are divided between the SMP states: ready, running and blocked. Each job has 10 threads. The simulation was composed of 19 such jobs. The μ of these jobs is 10% of the quantum.

4.3.5 Difference Between Jobs Composed of Two Threads and the Rest

In figure 4.1 we notice that the graph associated with jobs of the size 2 doesn't behave as jobs composed out of more threads. Instead of converging to some value between 25 to 50 the curves converge to zero. This should come as no surprise in light of alternating synchronization since - as mentioned before - the SSR metric doesn't embody the "successful spin" of the last thread to arrive to a barrier because this thread actually didn't perform spinning. This means that only the "unsuccessful spin" of the first thread to arrive to a barrier affects the SSR thus making it converge to zero.

4.4 The Consequences of Alternating Synchronization

4.4.1 Expected SSR

We've seen that jobs with relatively small computation intervals (with respect to the quantum) i.e. jobs that perform "a lot" of synchronization have a tendency to fall into an alternating synchronization pattern. For this type of computation the SSR has a 50% upper bound. This is true because the best we can expect from a thread is to successfully spin at the first barrier it reaches in the quantum (causing the blocked threads in its job to move to the end of the ready queue) and fail spinning at the next (causing it to be preempted and enter blocked state). For every successful spin a thread performs, it also performs an unsuccessful one.

As shown at the end of section 4.3.4, sometimes a thread doesn't contribute even a single successful spin in a quantum. Instead it joins the group of threads that are in blocked state. This is the reason why the overall SSR is always less than 50%.

4.4.2 CPU-Time Wasted

When a job is alt-synchronizing, the SMP system pays a heavy price in terms of CPU time:

Let J be a job that performs alternating synchronization. This means that J's threads are divided in to two subsets - A and B - which compute alternately such that each group causes the other to move from blocked state to the end of the ready queue. We will (optimistically) assume that each thread in J always manages to contribute one successful spin when it is allocated a processor.

The following is a description of what happens when the threads of a subset (A for example) are allocated processors:

- Each thread in A is scheduled. This consumes $t_{context\ switch\ in}$ cycles per thread.
- All of A's threads begin to compute, reach the first barrier in the current quantum and spin while waiting for T_0 - the last thread to reach that barrier. If t_0 is the number of cycles it took T_0 to reach the first barrier then this stage consumed t_0 cycles per thread. Since the computation intervals are normally distributed, it is safe to say that in most cases: $t_0 \geq t_{cmput}$ where t_{cmput} is the expectation of the normal distribution. We therefore conclude that this stage consumed no less than t_{cmput} cycles per thread.
- A's threads continue to compute, reach the second barrier, spin, fail and block. This stage consumes on average:
$$t_{cmput} + t_{spin} + t_{context\ switch\ out}$$
cycles per thread.

The above scenario also holds for subset B. In each such scenario the subset "passes" two barriers and therefore the overall CPU time consumed by *J* is:

$$|J| \cdot (t_{context\ switch\ in} + t_{cmput} + t_{cmput} + t_{spin} + t_{context\ switch\ out}) \cdot \frac{b}{2} =$$

$$|J| \cdot (2t_{cmput} + t_{spin} + t_{context\ switch}) \cdot \frac{b}{2}$$

where b is the total number of barriers performed by J. Since in most systems: $t_{spin} \geq t_{context\,switch}$ we get that the overall CPU time consumed by *J* isn't less than:

$$|J| \cdot b \cdot (t_{cmput} + t_{context\,switch})$$

This means each of *J*'s threads wastes $t_{context\,switch}$ cycles on context switching (or spinning) for **every** t_{cmput} cycles of computation it performs or in other words, for every barrier it performs.

4.5 The Role of Shuffling the Ready Queue on Startup

So far we haven't discussed why the ready queue was shuffled by the startup procedure of most simulations. This seemed reasonable at the beginning of the discussion because we assumed threads aren't arranged in any particular order in the ready queue. This assumption has proven to be wrong when we've witnessed alternating synchronization. Shuffling however has an important effect on the simulations: it causes the jobs to alt-synchronize right from the beginning of the simulations. This is true because in most cases (from a certain load) a job's threads aren't allocated processors all at once. Therefore, some threads of the job end up in blocked state (failing to synchronize at the first barrier) which is all that is needed for a job to begin alternating synchronization.

In section 3.2.1.3 (page 23) we've seen that the SSR achieved by a job is very high when its threads are allocated processors all at once. When the ready queue isn't shuffled, this is indeed the case during the first few quanta (which we've named the "grace period"). After a while however, the grace period ends, the job starts to alt-synchronize and the SSR drops dramatically (as demonstrated in figure: 3.3 page: 25).

The shuffling of the ready queue therefore allows us to perform shorter simulations and concentrate on the effects of alternating synchronization while masking the "white noise" of high SSR achieved in the initial grace period.

Figure 4.3 shows the results of the original simulation with the difference that the ready queue is not shuffled on startup. We can see that graphs associated with jobs that have sizes different than a power of 2 behave like the original graphs but achieve a higher SSR (due to the "grace period"). As expected these SSRs are dramatically reduced when we increase the number of barrier from 50 to 500 (and thus reducing the weight of the grace period). This is demonstrated in figure: 4.4.

For obvious reasons, this isn't the case for jobs with sizes that are a power of two: For 1% computation interval, the SSR is 100% because the jobs' threads always manage to execute together (one might say that the jobs are gang scheduled [10]). For 10% computation interval, the SSRs get smaller as the size of the jobs get bigger. This happens because the σ-interval is big enough to allow the scenario described in 4.3.4 (in which a thread doesn't contribute even a single successful spin) to occur and the bigger the job is, the chances this scenario will happen increase.

4.6 Intermediate Load: the 50%-SSR Threshold

When we examined figure 4.1 (page 34) the most obvious phenomenon was indeed the asymptote caused due to alt-synchronizing. However, there is another interesting phenomenon presented in this figure:

There exists an interval between the point where the system is full (32 threads) and a point where there are more threads than processors, for which the SSR is fairly high i.e. bigger than 50%. This 50% threshold seems a natural limit for examining whether spinning was justified: we can conclude for certain that for $SSR \leq 50\%$ the parallel jobs would have consumed less CPU time had they used the "alway-block" policy (the opposite however is not certain because we can't be sure that a high SSR ensures better throughput; this issue will be addressed later).

Figure 4.5 displays the data that were presented in figure 4.1 but with a smaller x-range (the first 100 threads) which allows us to focus on this phenomenon.

Figure 4.6 summarizes these results specifying - N - the maximal number of surplus threads for which the SSR is bigger than 50%, and the average SSR from the point where there are more threads than processor until N. We can see that fine grain jobs achieve bigger surplus than medium grain jobs. We can also see

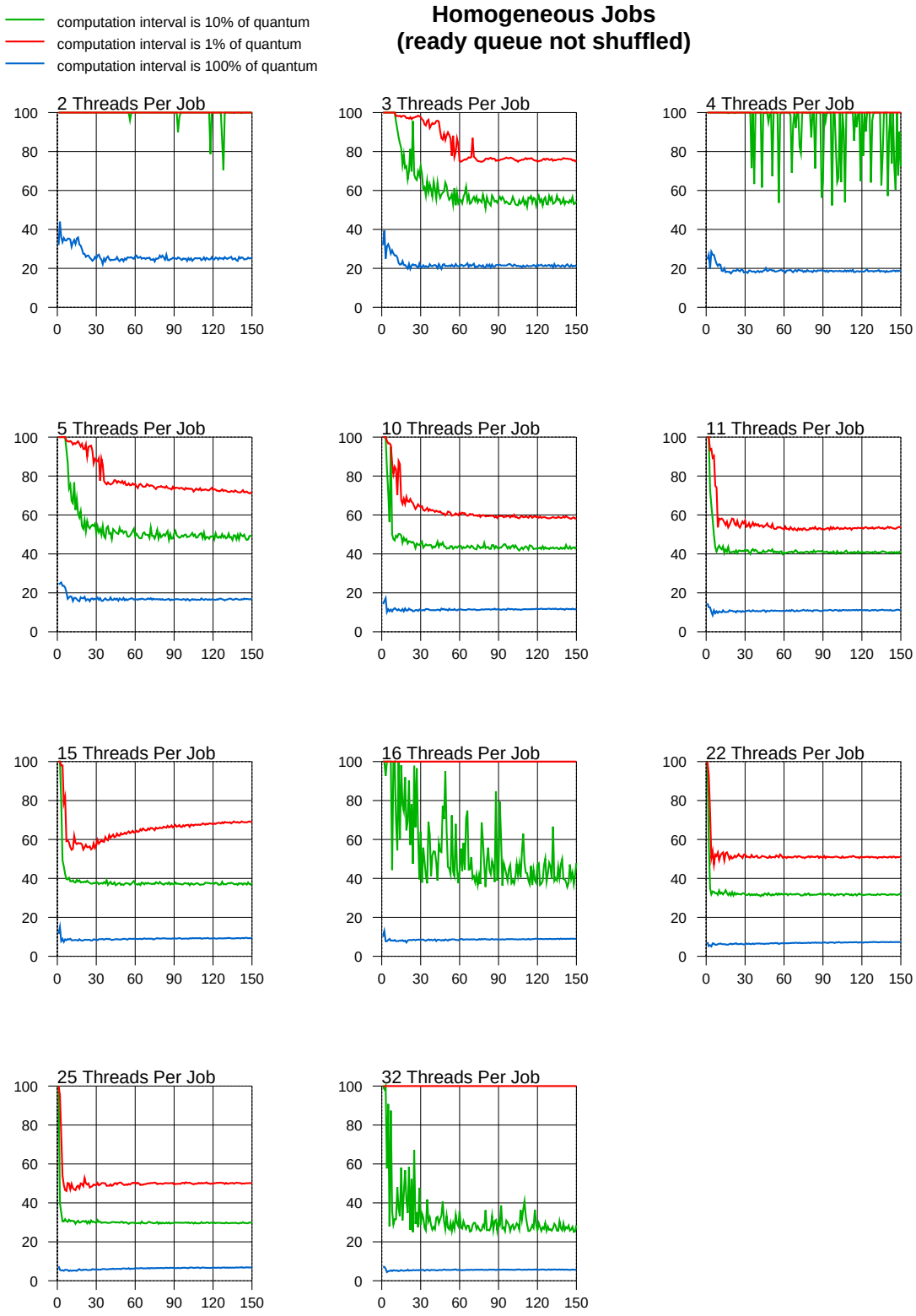

Figure 4.3: This figure presents the result of a simulation identical to the original (see figure: 4.1) with the difference that the ready queue is not shuffled on startup. For sizes which are not powers of two, the behavior is similar but SSRs are higher because of the “grace period”. For power-of-two sizes: jobs with 1% computation intervals are gang scheduled and therefore achieve 100% SSR. The SSR achieved by jobs with 10% computation intervals gets smaller as jobs’ sizes get bigger.

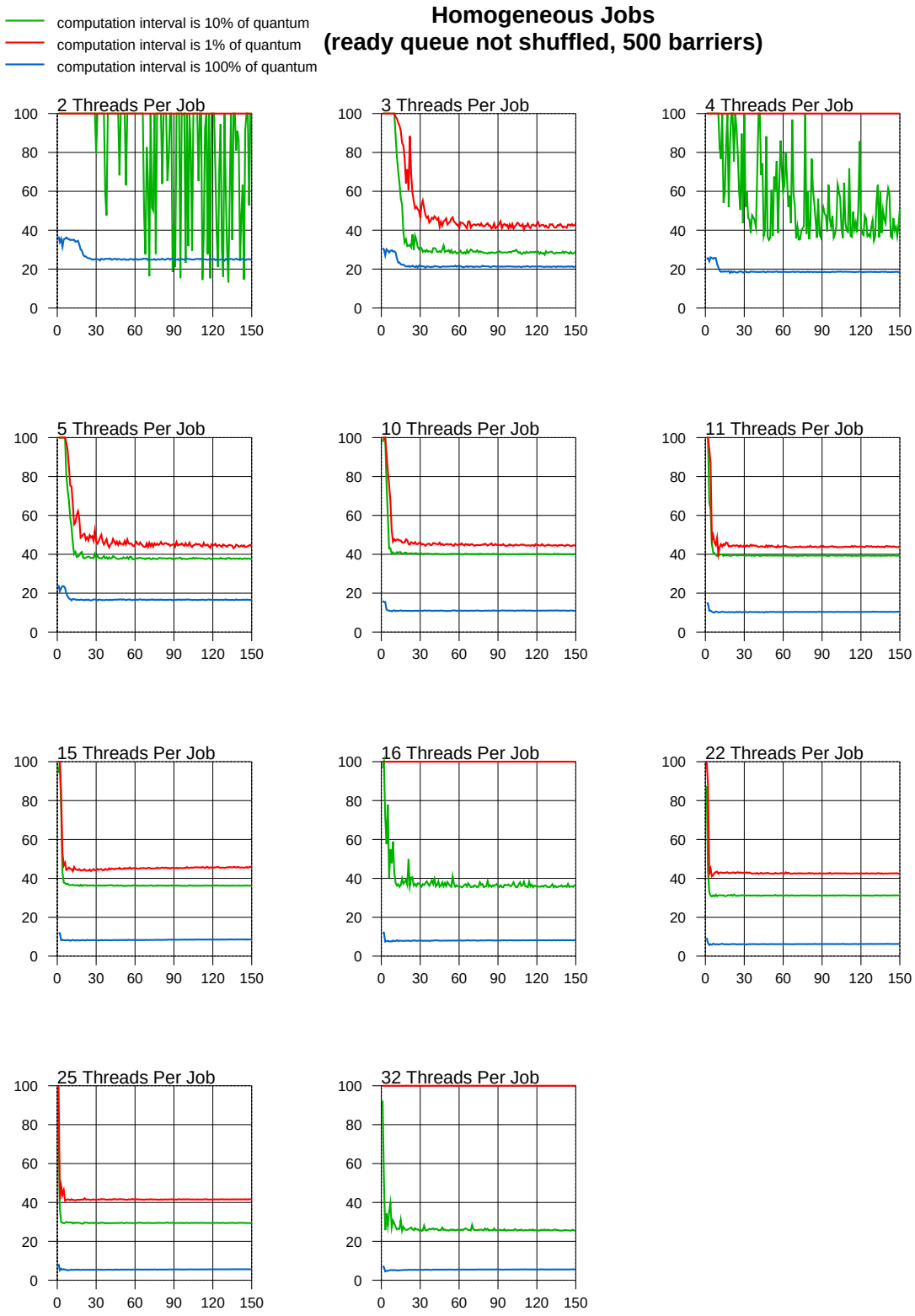

Figure 4.4: This figure is similar to figure: 4.3 but the simulation uses 500 barriers instead of 50. SSRs are reduced because the weight of the grace period is lessened.

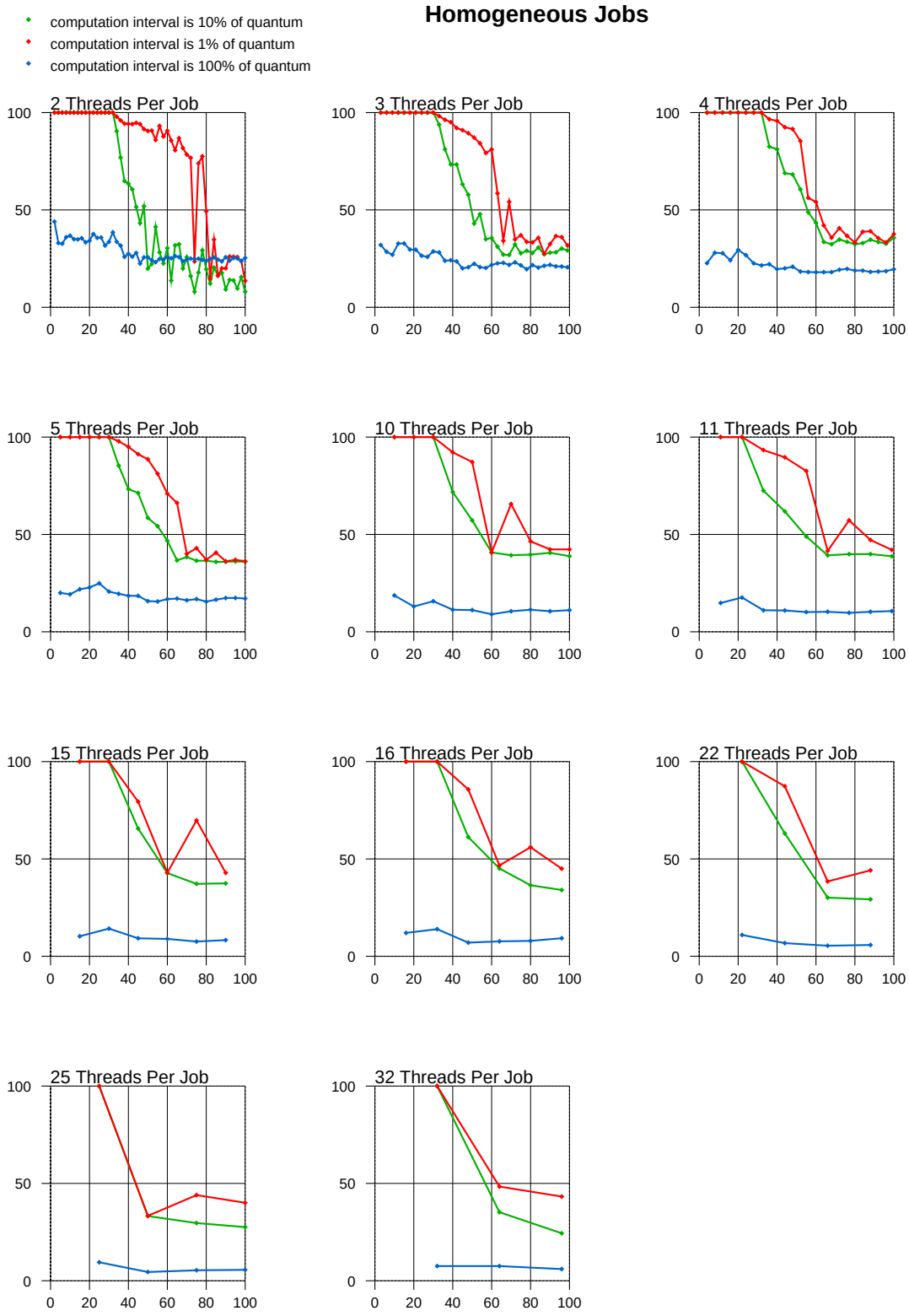

Figure 4.5: The X-axis presents the number of threads participating in the simulation. The Y-axis is the SSR as usual. This figure presents the data that was displayed in figure 4.1 but uses a smaller maximal-x-range of 100 threads. It is clear that jobs with sizes less then 25, still manage to achieve a decent SSR when there are more than 32 active threads.

that smaller fine grain jobs do better than larger ones. The maximal surplus appears bounded from above by the number of CPUs (aside from jobs with size 2, surplus is smaller than 32).

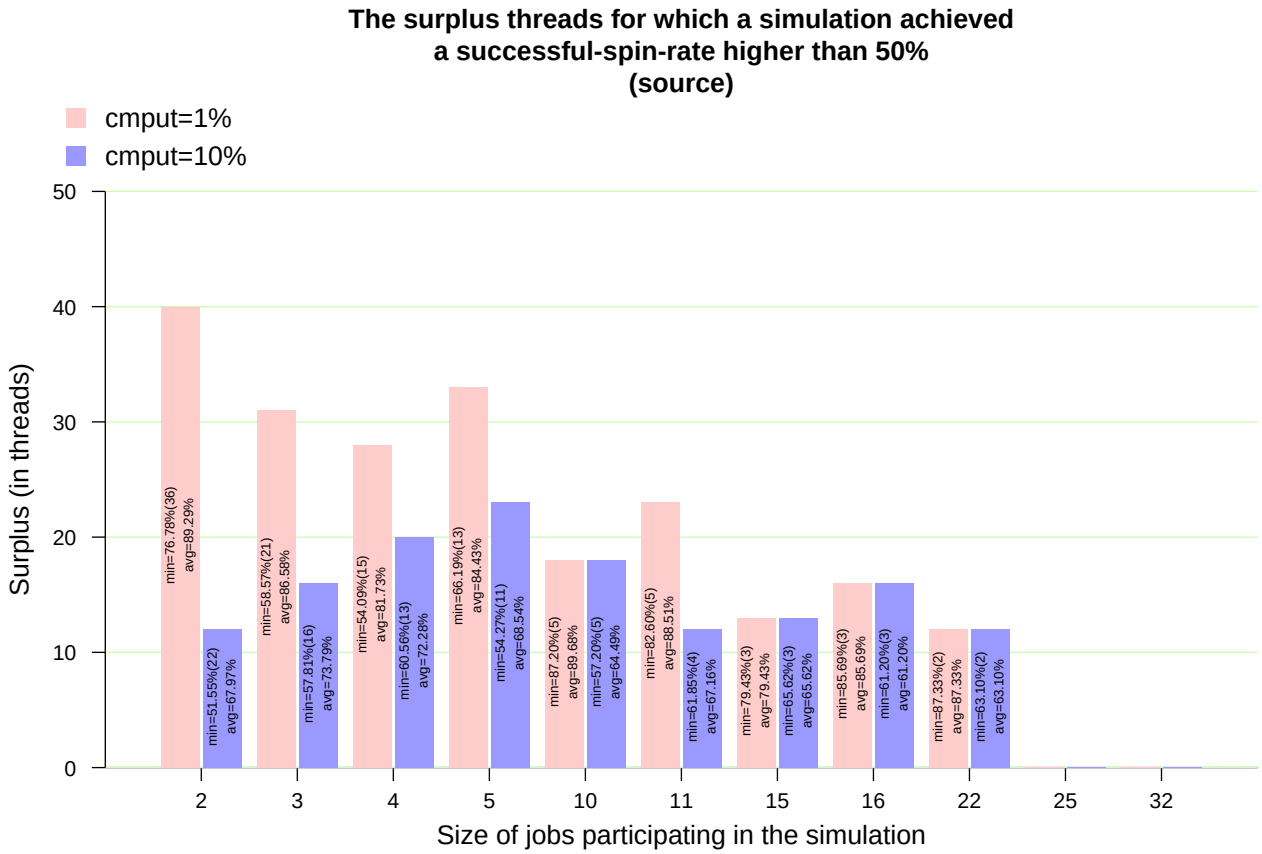

Figure 4.6: This figure summarizes the results displayed in figure 4.5. The height of the bar - N - is the maximal number of surplus threads for which the simulation achieved SSR $> 50\%$. The 'avg' and 'min' on each bar specifies (respectively) the average and the minimum SSR achieved by the simulations composed from [33...32+N] threads. The data specified in the parentheses is the number of jobs (not threads) that participated in the simulation when the 'min' SSR was achieved.

Chapter 5

Heterogeneous Collection Of Synchronizing Jobs Under the Round-Robin Scheduler

5.1 Introduction

5.1.1 Motivation

In the previous two chapters we have examined two fairly simple scenarios: (1) a synchronizing job in a non-synchronizing environment and (2) a homogeneous collection of jobs all having an identical profile. The analysis of these scenarios suggested that when using a round-robin scheduler, fine/medium grain jobs behave as follows:

1. When the load is smaller or equal to the number of CPUs, spinning is almost always successful and worth while (trivial). Of course for such a load it's always preferable to spin (regardless of grain) as there aren't any other threads waiting for execution.

2. For intermediate load (more threads than #CPU but less than 2#CPU), the SSR gradually drops from the neighborhood of 100% to some value below 50% as it falls into an alt-synchronization pattern.

3. For bigger (than 2#CPU) loads, the computation is always done in an alt-sync computation pattern.

It is far more likely that an executing job collection will be more complicated and diverse than the two scenarios described above. We would like to show that our conclusions are independent of the job collection i.e. that regardless of its specifics, there exists an intermediate load interval in which fine/medium grain jobs will achieve SSR $> 50\%$, gradually entering an alt sync computation pattern.

5.1.2 Method

Numerous simulations were conducted with all sorts of job collections — usually generated using some sort of randomization mechanism — and all of them (aside for some exceptions) verified the above conclusions. This chapter will present the results of some of these simulations and will point out the "exceptions to the rule".

The parameters that determine the grain of a job are mainly μ and σ. Therefore, our main focus in this chapter will be on those two parameters. Section 5.2 will present the result of a simulation that preserves the connection we used so far between μ and σ namely: σ=90/15% (i.e. the σ-interval will always be 15% out of μ). However, jobs participating in this simulation will have various sizes and μ-s. Section 5.3 will "break" this connection between μ and σ for the first time: We will use a constant σ-interval across different (relatively big) μ-s. This simulation will present the only exceptions we found to the conclusions

stated above. Finally, section 5.4 will present the result of a simulation that for each job chooses randomly and independently both μ and σ.

5.1.3 Distribution Representation

In this chapter the following notation is used to represent distributions of μ, σ and size of jobs. A distribution is specified as a comma separated list of pairs in the form:

$$value_0 : weight_0\ ,\ value_1 : weight_1\ ,\ ...\ ,\ value_n : weight_n$$

such that:

$$\frac{weight_k}{\sum_{j=0}^{n} weight_j}$$

is the probability that $value_k$ will be chosen. Each value may be expressed as an interval of the form:

$$begin - end$$

which means that some number - α - that satisfies: $begin \leq \alpha \leq end$ should be uniformly chosen. For example , μ may be defined to be:

$$1 - 20\ :\ 3\ \ ,\ \ 21 - 30\ :\ 2\ \ ,\ \ 31 - 40\ :\ 1$$

This means that there's a 50% chance that μ will be some (uniformly chosen) number from the interval 1-20 , 33.33% chance that μ will be from 21-30 , and 16.66% chance that μ will be from 31-40.
σ may be expressed like this:

$$80/1 - 15\ :\ 3\ \ ,\ \ 90/16 - 30\ :\ 1$$

which basically means the same thing as explained above but now the σ-interval is chosen uniformly e.g. there's a 75% chance that $\sigma = 80/\alpha$ where α is a number chosen uniformly form the interval 1-15.

5.1.4 End Point of Simulations

All the simulations presented in this chapter was configured to end along with the first thread that finishes its computation. This way the results of the simulations will not be distorted by the load that gradually decreases towards the end of the simulation (when only part of the threads have finished and the rest operate in a less loaded system).

5.1.5 Simulator's Random Permutation Mode

Contrary to previous chapters in which each curve in each graph was associated with an independent simulation sequence, curves in this chapter describe portions of the job collection executing in parallel within the same simulation. Each curve describes the average SSR of a different *job class*. Each job class is associated with one pair from either the μ or the σ distributions (as defined above in section 5.1.3). In each simulation we must choose the distribution according to which the simulator will classify the jobs. A sequence of simulations is constructed as follow:

1. As usual, the simulator receives a configuration file specifying all the parameters describing the simulation. These include the various distributions as defined in section 5.1.3.
2. The simulator also receives two additional parameters:
   (a) A number - N - that specifies the maximal number of threads to participate in the simulation sequence, and
   (b) The parameter according to which jobs will be classified to job-classes (μ or σ).

3. Then, by using the given distributions, the simulator iteratively chooses jobs and add them to the job sequence up till the point where the total number of threads composing the jobs in the sequence exceeds N.

4. Let j_1 , j_2 , ... , j_k denote the randomly chosen job sequence. The simulator will conduct k simulations such that the i-th simulation will be composed from jobs: j_1 , j_2 , ... , j_i . The number of threads in the i-th simulation is the i-th x-value displayed in the graphs. Each y-value associated with this x, denotes the SSR of some job class (achieved upon load = x).

When the simulator is instructed to behave as described above we say that the simulator runs in *random permutation mode*.

5.2 The σ-Interval as a Percentage of μ

5.2.1 Description

In this section we chose to preserve the connection used so far between μ and σ and therefore define σ to be 90/15%. However, the μ and the size of the jobs in this simulation are given as distributions. The parameters used in the simulations are:

p	q	in	out	sync	nosync	barrier	spin	μ	σ	rand_ord	seed
64	100	3%	3%	4-7 : 9	0	50	6%	1-20% : 1	90/15%	1	0
				8-12: 1		200		21-30% : 1			1
						1000		31-40% : 1			2
								41-50% : 1			3
								51-60% : 1			

- Note that the simulation described in this section was executed on a 64-processors SMP (as opposed to 32 in previous chapters).

- The simulation involves 5 gradually increasing classes of μ (all with equal weight) from which the first one - μ=1...20% - represents the fine and medium grain jobs. The 20% was chosen as the biggest value of the fine & medium grain job-class because the effective dispersal of the computation intervals of a job with μ=20% and σ=90/15% is 6% of quantum, which is exactly the chosen spin (and context-switch) length. Recall that (as explained in the previous chapter) jobs with bigger μ are expected to achieve very low SSR even when the number of threads is smaller than #CPU.

- We've chosen the μ classes to demonstrate the intermediate-load principal discussed above: We expect that only the curve associated with μ=1...20 will achieve SSR$>$50 in the intermediate-load. As for the other μ interval: based on the results from the previous chapter, we expect that bigger values of μ will result in smaller SSR.

- The chosen sizes of the jobs participating in this simulation is relatively small in comparison to the number of CPUs (about 5-20%). This sizes distribution was chosen so that the load will increase gradually thus avoiding big (X-axis) leaps.

- As the randomization elements quantity gets bigger, it becomes more important to examine the results across various different seeds so as to make sure these results are indeed the common case. This is the reason why we've chosen to display graphs associated with more than one seed: Each simulation results will be displayed using 4 graphs, each graph is associated with a different seed (many more seeds that are not displayed here were used and produced relatively similar results).

- Note that this simulation uses an increasing number of barriers. The justification for this is empirical: we've noticed that for some seeds, a small number of barriers (50) is not enough to produce a consistent picture of the SSR. As we prolong the length of the computation (i.e. increase the number of barriers) the picture tends to stabilize, all the SSR peaks perceived in simulations using a small number of barriers disappear, and the SSR-curves become smooth.

5.2.2 Results

The results of the simulation are presented in figure 5.1. Figure 5.2 zooms in on the intermediate load of figure 5.1's last row's graphs. These results confirm our expectations:

When examining the 1000 barrier graphs we can see that the curves associated with the fine/medium grain jobs manage to sustain a SSR bigger than 50% in the intermediate load, until some point between load=80...88 (i.e. surplus of 25-40% of CPU#). These findings coincide with the findings of the previous chapter presented in figure 4.6 (page 44). Some of the surplus displayed in figure 4.6 is bigger than the surplus displayed here in figure 5.1 but this can be explained when considering that the SSR displayed here is an average between fine and medium grain jobs and "medium-grain" is defined to be $\mu \leq 20\%$, whereas in figure 5.1 the SSR achieved by fine and medium grain is displayed separately and medium grain is defined to be μ=10%.

Also note that the other curves behave as we anticipated earlier: Bigger μ values result in smaller SSR. The only other curve that sometimes (barely) manages to display SSR > 50 (though very close to 50) for load slightly bigger than CPU# (surplus of 2-6 threads) is the one associated with μ=21...30 . This is understandable because we use a normal distribution for computation interval which have the nature of being condensed around the expectation. Thus making it possible for jobs with μ slightly bigger than 20% to still (sometimes) have an effective dispersal of computation intervals that is not bigger than 6% of quantum (= spin interval = context switch length).

5.2.3 Other Values For the Parameters

Many simulations similar to the one defined above - but with different values for some of the parameters - were conducted. All of these simulations' results coincided with the results presented here. The following is a description of some of these simulations:

- Simulations using μ-intervals containing bigger values (than 60%) achieved results consistent with our findings here: e.g. μ=71...80 produced lower SSR than μ=61...70 , which produced lower SSR than μ=51...60 etc.
- In the above simulation we used equal (uniform) weight for each μ-interval. Altering the various weights of these intervals to several configuration didn't produce a fundamental change in the result. Again we received results supporting our understandings as stated in section 5.1.1.
- Bigger job sizes (up till the number of CPUs) were used. Whenever the random choosing of the job collection managed to grow steadily in the intermediate load (instead of skipping it), these simulations also produced similar results to the ones displayed here.
- Bigger numbers of barriers were used (2000, 5000, 10000) and produced curves that are almost identical to those displayed here when 1000 barriers are used.

5.3 A Constant σ-Interval

5.3.1 Description

Up till now, all the simulations we've conducted used σ=90/15% i.e. the σ-interval was always expressed as a percentage of the μ. The immediate result of this was that any job with a relatively big μ (i.e. big enough so that $2 \times 0.15 \times \mu$ is bigger than the spin interval) wasn't at all interesting in terms of the behavior

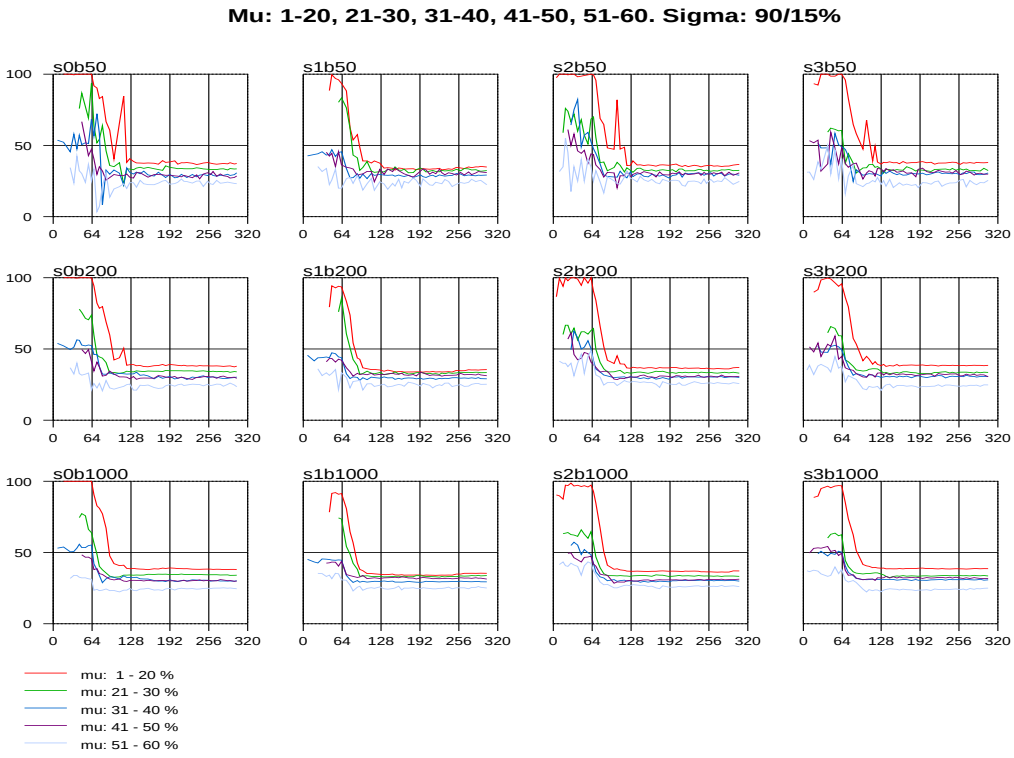

Figure 5.1: This figure displays the results of the simulation defined in section 5.2.1 and analyzed in section 5.2.2. The Y axis is associated as usual with the SSR. The X axis displays the total number of threads participating in the simulation. Each graph title - sNbK - specifies the seed (=N) and the barrier number (=K) that were used in the simulation. We can see that curves get "smoother" and peaks are eliminated as the number of barrier is increased. It is also apparent that the only curve displaying SSR > 50 in the intermediate load is the one associated with the fine/medium grain jobs (μ=1...20). A "zoom in" on the intermediate load in the graphs of the last row is displayed in figure 5.2.

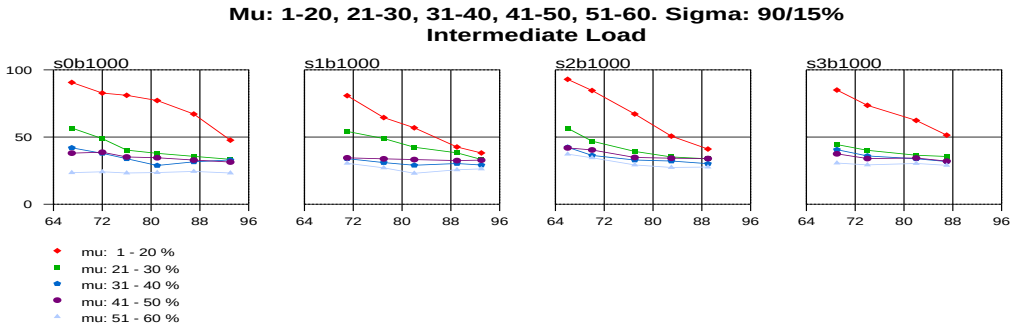

Figure 5.2: This figure "zooms in" on the intermediate load of the graphs displayed in the last row of figure 5.1. We can see that fine/medium grain jobs (red line) manage to achieve SSR > 50 until the load is somewhere between 80...88 (25-40% thread surplus). We can also see that sometimes jobs with μ=21...30 manage to achieve a 2-6 surplus with SSR just above 50%, a fact that is explained in section 5.2.2.

of its SSR (which was always close to zero). This led us to conduct a number of simulations in which the connection between the σ-interval and the μ was arbitrary.

The first unsurprising result was that any job with a σ-interval that was bigger than $\frac{spin}{2}$, resulted (by definition) with computation intervals with a dispersal bigger than the spin interval, which in turn resulted in a SSR close to zero (even for very small μ values).

The second unsurprising result was that jobs with small σ-intervals and small μ values produced similar results to those already demonstrated for fine and medium grain jobs (i.e. SSR $>$ 50% for some intermediate load after which SSR drops to some fairly constant value below 50% due to alt-synchronization).

The only new question was how do jobs with relatively big μ and relatively small σ-interval behave.

The configuration of the simulation presented here is similar to the configuration used in the previous section:

p	q	in	out	sync	no sync	barrier	spin	μ	σ	rand ord	seed
64	100	3%	3%	4-7 : 9 8-12: 1	0	1000	6%	51-60%:1 61-70%:1 71-80%:1 81-90%:1 94-100%:1	90/0.15-15:1	1	0 1 2 3

The difference is in μ and σ:

- The μ values were chosen to be bigger: all job classes have $\mu \geq 51\%$ and cover most of 50-100% of a quantum.
- The σ is set to be:

  $$90 \,/\, 0.15 - 1.5 \;:\; 1$$

  Note that the σ-interval is not expressed as a percentage of μ but rather given directly as a constant interval. This σ-interval was chosen to match the interval derived from when:

  - μ=1%,10% , and
  - σ=90/15%

  i.e. the lower bound of the interval - 0.15 - was the σ-interval of μ=1% (fine grain) and the upper bound of the interval - 1.5 - was the σ-interval of μ=10% (medium grain) when σ=90/15% was used.

Recall that previous simulations proved that for μ=1%,10% and σ=90/15%, there exists an intermediate load with SSR $>$ 50 . . . , now the question is how will jobs with bigger μ value but equally small σ-interval will perform.

5.3.2 Results

The results of the simulation are presented in figure 5.3 . Same as in the previous section, figure 5.4 zooms in on the intermediate load. From a quick look at these figures, we can see that:

1. The job class associated with μ=94. . . 100% achieves surprisingly high SSR.
2. The order of the other curves is reversed with respect to the order we got used to: Up till now bigger μ implied lesser SSR. Now it's the other way around.
3. The SSR achieved by all the job classes (other the one associated with μ=94. . . 100%) suggest that spinning will not be profitable (virtually no intermediate load with SSR $>$ 50).

The following subsections will explain the above.

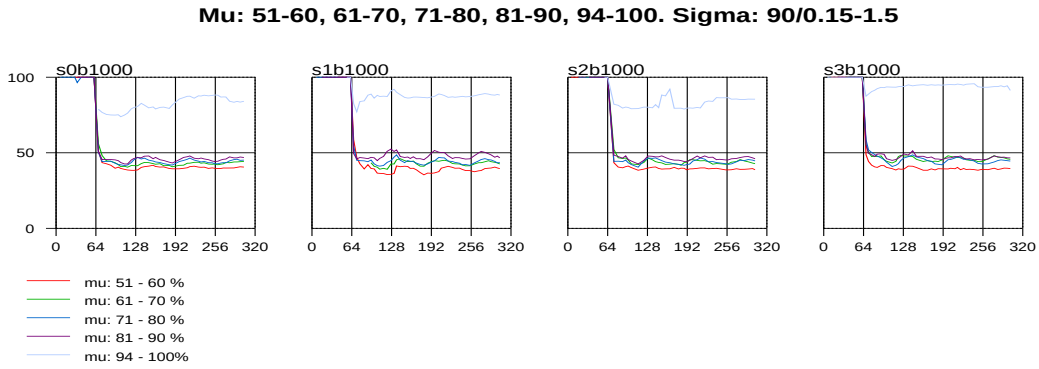

Figure 5.3: This figure displays the result defined in section 5.3.1 . As usual, X-axis displays load (number of threads), Y-axis displays SSR, and the title of each graph denotes the seed and the number of barriers used. The job class associated with μ=94...100% achieves high SSR for every load. The other curves are ordered such that curve associated with bigger μ is closer to the 50%-SSR-threshold.

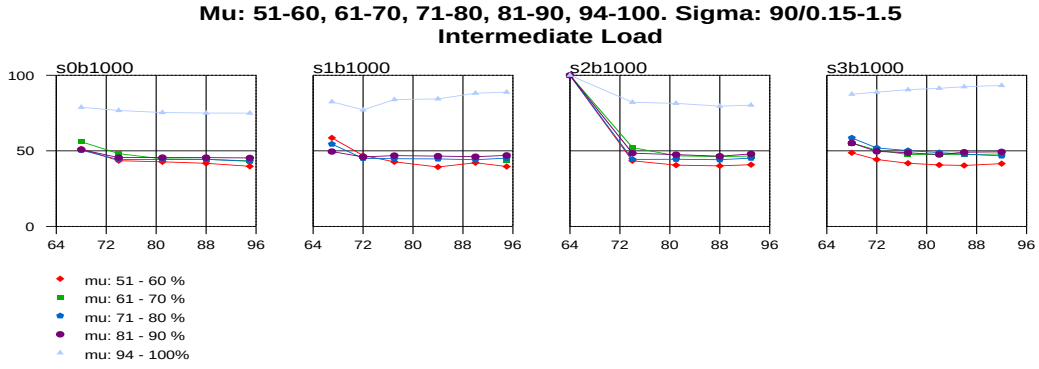

Figure 5.4: This figure displays a "zoom in" on the intermediate load of the simulation's results presented in figure 5.3 . All job classes (aside from the one associated with μ=94...100%) achieve SSR that negates spinning in the intermediate load.

5.3.2.1 The High SSR of the Job Class Associated with μ=94...100%

The fact that jobs with $\mu \in \{q - spin\ ,\ ...\ ,\ q\}$ and a small σ-interval achieve very high SSR is easily explained. Let J be such a job. Let T be a thread of J. When T finishes a computation phase and starts to spin, chances are that it will be preempted while it is spinning and moved to the tail of the ready queue. Let $X \subset J$ denote J's threads that are waiting for a processor at the time instance T was preempted (obviously T is waiting for these threads to complete the current barrier). By the time T is rescheduled to execute, it will usually continue to spin (because in its previous quantum it was stopped by the scheduler in the middle of this process) and immediately succeed since X threads were already allocated a processor (they were ahead of T in the ready queue).

5.3.2.2 The Reversed Order of the Other Job Classes

Since its behavior was explained earlier, our current discussion excludes the job class associated with μ=94...100%. Within the other job classes we notice a strange phenomenon: for load bigger than CPU#, the curves' order is reversed with respect to what we've got used to see. Usually (when the σ-interval was expressed as a percentage of μ), bigger μ implied lesser SSR, and here we see exactly the opposite. Let us represent μ as: $\mu = q - d$ (where q is the length of the quantum, $d < \frac{q}{2}$ and $\mu > \frac{q}{2}$). As mentioned, jobs with smaller d values achieve slightly better SSR. The job class with d=10...19% (μ=81...90%) sometimes even achieves SSR > 50! However, in most cases these jobs achieve SSR which is smaller than 50 and even when the SSR is more than 50 it's not high enough to be worth spinning. Having said that, and after concluding that the jobs discussed in this section should not spin (for load $>$ CPU#) it is still interesting to understand what is the cause for this "reversed order". After carefully examining the events of these simulations we've concluded that the reason is the following:

- Let's examine some job - J - when the system is very loaded.
- Unavoidably, J's threads divide to two subsets X and Y as described in the alt-synchronizing scenario.
- Assume X has now began its quantum: X computes for μ cycles and reaches b_k (the k-th barrier) causing Y to move from blocked to ready state. X then continues to compute for d cycles, finishes its quantum and gets preempted back to the tail of the ready queue (recall that $d < \frac{q}{2}$ and $\mu > \frac{q}{2}$and therefore X will not reach b_{k+1} within the current quantum). Note that when X is preempted it has $\mu - d$ more cycles to compute until reaching b_{k+1} .
- Now, in order for J to "beat the system" and break the alt-sync pattern, X and Y should be dispatched d-cycles apart (as Y has μ cycles and X has $\mu - d$ cycles until reaching b_{k+1}).
- At this point there are exactly 2 possibilities:
  1. The difference between the dispatching of Y and X is smaller/bigger than d, enough to make Y fail on b_{k+1}.
  2. The difference between the dispatching of Y and X is in the proximity of d which will result in the reunion of X and Y until such time when J splits up again to two subsets (this time is actually very soon, namely the end of the current quantum, since X will be preempted d cycles after Y). In this case J has (temporarily) managed to break the alt-sync pattern and all of its threads successfully completes b_{k+1} .
- The second possibility is exactly the reason why the curves in this simulation get really close (from below) to the 50% SSR threshold and sometimes even exceed it. In the first possibility however, lies the explanation for the fact that jobs with bigger μ values are closer to that threshold.
- When the first possibility occurs (which is what happens more often than not), the scenario described above will repeat itself with the difference that now in order for X and Y to reunite, they should be

dispatched $2d$-cycles apart: This is true because after Y fails on b_{k+1} , and X successfully completes it, X will have

$$q - (\mu - d) \;=\; (\mu + d) - (\mu - d) \;=\; 2d$$

more cycles to complete its quantum. This means that at the beginning of X's next quantum, it will have $\mu - 2d$ cycles until reaching b_{k+2} ...

- This argument can be applied again and again i.e. if the $2d$ interval didn't work the scenario will repeat itself with a $3d$ interval etc. The argument may no longer be applied when the interval - $n \cdot d$ - is bigger than $\frac{q}{2}$ in which case X will reach a second barrier in the same quantum, fail and block. As a result, Y and X will simply flip the roles they play and everything will start from the beginning.
- Smaller values of d result in a more refined interval series, i.e. the smaller d is, the bigger chance the job has to "get the dispatch interval right" and thus to unite the two subsets for a successful barrier. The refined interval series is the reason why jobs with smaller d values are closer to the 50% SSR threshold.

5.3.2.3 The Intermediate Load

Contrary to jobs with small μ and σ-interval values (i.e. fine and medium grain jobs), the jobs in the simulation currently discussed do not achieve SSR > 50 in the intermediate load. The reason for this is related to the explanation given in the previous subsection: A high SSR in the intermediate load is a function of the job's threads' success to execute simultaneously from time to time . When this happens, X and Y (using the terminology of the previous subsection) have a chance to complete a number of barriers until the scheduler splits them up again. The smaller μ is, the more barriers J's threads may complete on that period. However, in the current simulation $\mu > \frac{q}{2}$ and therefore the maximal number of barriers that J's thread may complete on that period is bounded by 2, after which it will be a while before X and Y will manage to execute simultaneously again.

5.4 A Random σ-Interval

By now, it seems that most SMP (round-robin) executions involving most job collections are well understood. However, we haven't yet conducted a simulation that randomly chooses both μ and σ. Even though we can probably predict what will be the result of such a simulation, for completeness, we conduct such a simulation and present its results. The parameters used in this simulations are:

p	q	in	out	sync	no sync	barrier	spin	μ	σ	rand ord	seed
64	100	3%	3%	4-7 : 9 8-12: 1	0	1000	6%	1-20%:1 21-30%:1 31-90%:1	90/0.15-0.75:1 90/0.9-1.5:1 90/1.65-3:1 90/3.15-15:3	1	0 1 2 3

The classification of the jobs will be done according to the σ-interval. Let α denote the effective range of the computation-intervals' dispersal, then we get:

σ-interval			associated α			bigger than spin
0.15	...	0.75	0.3	...	1.5	no
0.9	...	1.5	1.8	...	3	no
1.65	...	3	3.3	...	6	no
3.15	...	15	6.3	...	30	yes

Half of the jobs, those associated with the 4th σ, don't have a chance to achieve SSR > 50 regardless of their μ. This is the only concrete thing we may predict. As for the other σ-s: it depends on the chosen μ. A job with smaller μ will achieve better SSR in the intermediate mode. On the other hand, a job with a very big μ may achieve SSR slightly bigger than 50 even after the intermediate load. Contrary to simulations conducted so far, these parameters were chosen with the intent not to have a clear "winning" job class or a clear job classes hierarchy. We therefore expect that different seeds will produce different winners that may change as the load increases. The result of the simulation are displayed in figure 5.5 (only the intermediate load is displayed) and exactly coincide with our predictions.

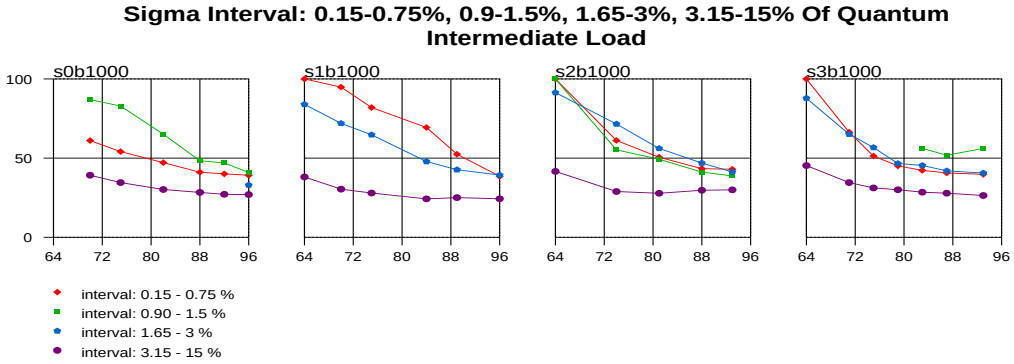

Figure 5.5: This figure displays the intermediate load of the results of a simulation composed from a job mix that was created by randomly choosing both μ and σ. The axes and title were defined in previous figures in this chapter.

Chapter 6

A more Realistic Algorithm: the Linux Scheduler

6.1 Introduction

So far, we have simulated and analyzed the Round Robin algorithm, which, though useful, common (especially in realtime applications/systems [12, chapter 5]), and formalized by POSIX.1b, isn't remotely as popular and widely used as priority based algorithms. The analysis of Round Robin has a value of its own. However, an important outcome of this analysis is that it allows us to gain intuition and insights as a first step towards understanding barrier-synchronization within the context of priority-based scheduling algorithms. The next step is to conduct and analyze simulations similar to those performed in previous chapters while using a priority based scheduler. The immediate question that follows is which scheduling algorithm to use. In order for the result of this work to have an additional practical value, we wanted a real world system scheduler. So the question became which common operating system scheduler to use. The most convenient choice was Linux (a) because it's an open source project, a fact that helps a lot (to say the least) when one needs to re-implement the scheduler of the OS, and (b) because of the vast amount and wealth of books, articles, websites and other resources documenting all its aspects. Aside from Linux accessibility, our decision was also influenced by the fact that nowadays it is the most popular flavor of UNIX.

There are many books and other resources that describe the Linux scheduling algorithm. Some of these which we relied upon are [4], [22], [3] and most importantly, the Linux kernel source code itself [26]. However, many readers will find it hard to extract all details relevant to this work from these references. Though lots of resources review the Linux scheduler, many of them tend to drown the reader with a lot of low level details that obstruct the actual algorithm, while others are doing it in a too high level manner (at least in th parts that are important in this work's context). Some give very good description but only on partial aspects. We found no single resource that presents all the pieces of the puzzle needed for the following chapters. The goal of this chapter is therefore to review the Linux-scheduler aspects relevant to this work and describe their implication on it. Sections 6.2 and 6.3 specify the version of the Linux kernel used and the scheduling details that are not covered by this chapter. Section 6.4 deals with definitions that are needed in order to present the scheduling algorithm. Section 6.5 presents the algorithm. Finally, section 6.6 points out the misfeatures that we have identified in the scheduling algorithm and their implication on the following chapters.

We remark that in the Linux-kernel, thread/process entities are indistinguishable. The conventional term used to represent them both is a *task*. Consequently, this is the term used throughout this chapter.

6.2 Linux Kernel Version

The Linux kernel code is constantly updated and revised. New minor version releases are a very common event (sometimes more than one in one month). However, though minor details have changed form time to

time, the core scheduling algorithm has essentially remained the same since Linux-2.2 (released in 1999). While the simulator code that implements the Linux scheduler was written, the latest kernel version was 2.4.5 (May, 2001) and this was the latest version we've consulted with. Currently, as these lines are written (Aug, 2001), the latest version is 2.4.9 .

6.3 Ignored Details

As mentioned before, POSIX1.b mandates three scheduling policies. Since our focus now is on Linux's SCHED_OTHER, details regarding the other two scheduling policies will be omitted from the scheduler description. Note that though in general each task may separately be assigned with one of the three scheduling policies, we assume all tasks' policies are SCHED_OTHER throughout this work. Policies may be assigned via the `sched_setscheduler` system call, tough one must have superuser privileges in order to set a policy to SCHED_FIFO or SCHED_RR. Another omitted issue is the nice value of tasks (which may be set through the `nice` system call) . Again, throughout this work, we assume that all tasks have a zero traditional nice value. The algorithm sections that determine the behavior of the scheduler on a uniprocessor were also omitted, only SMP related code is described.

6.4 Definitions

Before describing the algorithm, we first need some background definitions ...

6.4.1 Epoch

The Linux scheduling algorithm works by dividing the CPU(s) time into *epochs*. In a single epoch, every task has a specified time quantum whose duration is computed when the epoch begins (each task's quantum is refreshed exactly once in a single epoch). In general, different tasks may have different time quantum durations though this is possible only if tasks have different nice values or different scheduling policies which we assume isn't the case. The time quantum value is the maximum CPU time portion assigned to the task in that epoch. When a task has exhausted its time quantum, it is preempted and replaced by another runnable task. Of course a task can be selected several times by the scheduler in the same epoch, as long as its quantum has not been exhausted (for instance, if a task has blocked while waiting for a barrier, it preserves some of its time quantum and can be later selected again during the same epoch). The epoch ends when all the ready-to-run tasks have exhausted their quantum; in this case the scheduler recomputes quantum durations of all tasks and a new epoch begins.

6.4.2 Priorities

SCHED_OTHER tasks have two different kinds of priority, a *static priority* and a *dynamic priority*. Priorities are simply integers expressing the relative weight that should be assigned to a task when deciding which process should be allowed to spend some time on the CPU; the higher its priority, the better its chances:

static priority Called static because it doesn't change with time, only when explicitly modified by the user via a system call like `nice`() . The scheduling algorithm derives from this value the maximum duration of the quantum a task should be allowed, before forcing it to yield and allowing other tasks to compete for the CPU.

dynamic_priority Declines with time as long as the task is assigned a CPU; when it reaches 0, the task is marked for rescheduling. This field indicates the task's amount of time remaining in this quantum. It is reinitialized at the beginning of each new epoch according to the static priority value.

6.4.3 Dynamic Priority Resolution

The Linux-kernel sets things up such that it will get `HZ` clock interrupts per second. `HZ` is a macro with platform dependent value, though on most platforms (Intel's 80x86, Sun's SPARC and more) this value is set to be 100. A *tick* is defined to be the time that passes between each invocation of the kernel's interrupt handler ($= \frac{1}{HZ}$ second). The kernel's clock interrupt handler performs all sorts of administrative work, among which the updating of the dynamic priority. It follows that the dynamic priority resolution is in ticks.

6.4.4 Data Structures

Each task descriptor (a C structure) contains five fields used by the SCHED_OTHER scheduling algorithm:

`nice` This is the field that holds the static priority of the task. It's initialization value is 20 (the macro `DEF_PRIORITY`). The only way this field can be changed is through system calls like `nice` and `sched_setscheduler`. As mentioned before, this work assumes that the `nice` field never changes. Although related, this field is not to be confused with the traditional `nice` system call argument: the former is always positive and actually may hold $1 \ldots 40$ (when SCHED_OTHER is used); the latter has the possible values: $-20 \ldots 19$.

`counter` This field holds the dynamic priority of the task. When a task is created this field is initialized with half the value of its parent's `counter` (and the parent's `counter` is reduced by half). Whenever a new epoch is started, this field is reinitialized as follows:

$$\texttt{task.counter} \leftarrow \texttt{NICE\_TO\_TICKS(task.nice)} + \frac{task.counter}{2}$$

The definition of the macro `NICE_TO_TICKS` is dependent on the value of `HZ`. In Linux-2.4 it's defined to scale `DEF_PRIORITY` (=20) to the number of ticks composing 50ms. Since on most platforms `HZ` is defined to be 100 (and therefore a tick is 10ms), the definition of `NICE_TO_TICKS` is usually $\frac{nice}{4}$ (i.e. 5 ticks). As explained earlier, `counter` is decremented upon each invocation of the kernel's interrupt handler, and a task is marked as "needs rescheduling" whenever the `counter` becomes 0. It follows that `counter` holds the remaining number of ticks a task has till its quantum is exhausted and that in Linux-2.4 the default quantum duration is 50ms.

`processor` The logical id of the last CPU upon which the task has executed. If the task is currently executing, `processor` is the logical id of the CPU upon it's executing now.

`need_resched` This is a boolean flag checked by the kernel just before it switches back from system to user mode (e.g. after termination of kernel's interrupt handler). If `t.need_resched` is set, the kernel checks for `t.processor` whether a more desirable task than `t` exists, in which case a context switch is performed. Since this flag is checked only for currently-running tasks, it's usually more convenient to think of it as associated with a processor (rather than a task). This is true because if a context switch will take place due to a set `need_resched`, it will be on the processor that previously ran the task that was marked as `need_resched`.

`mm` A pointer to the memory page table of the task. If two different tasks have equal `mm` it means they have the same address space i.e. they are both threads belonging to the same parallel job.

6.5 The Algorithm

Most of the Linux scheduler is implemented in a single file in [26, kernel/sched.c]. There are four functions we must cover in order to understand the Linux scheduler. These are:

goodness Given a task, return how desirable it is: this is the value according to which tasks are compared in order to decide which will run next.

schedule Actual implementation of the scheduling algorithm. This function uses `goodness` to decide which task will run next on a given CPU.

__wake_up_common Wakeup a task when the event it has been waiting for happened. This event may be the arrival of all tasks of a parallel job to a synchronization point (barrier).

reschedule_idle Given a task, check whether it can be scheduled on some CPU (preferably on an idle one, but if there aren't any, by preempting a less desirable task). This function is used both by `__wake_up_common` and by `schedule`.

The following subsections describe each function in detail.

6.5.1 The `goodness` Function

Every time `schedule` is invoked it tracks the task with the best "goodness" in the ready-queue. A task with the best "goodness" is the one with the best claim to the CPU. Higher goodness values are better. A goodness value of 0 indicates that the task has exhausted its quantum. The `goodness` function is quite a simple function, yet it's a crucial part of the Linux scheduler. It is called for every task in the ready-queue every time `schedule` executes, so it has to be quick. But if it makes a bad decision, the whole system suffers. The pseudo code of `goodness` is presented in algorithm 1.

Algorithm 1 The `goodness` function pseudo code.

```
 1  goodness(task t, cpu this_cpu) {
 2     weight ← t.counter
 3     if( weight == 0 )
 4         return 0
 5     if( t.processor == this_cpu )
 6         weight ← weight + PROC_CHANGE_PENALTY
 7     if( t.mm == this_cpu.current_task.mm )
 8         weight ← weight + SAME_ADDRESS_SPACE_BONUS
 9     return weight
10 }
```

Line 1 Indicates that goodness is a function of both the task and the CPU it's a candidate to run upon ! As will shortly be demonstrated, the same task may have different goodness values on different CPUs.

Line 2 Initializing the local variable `weight` with the number of the remaining ticks `t` has in the current epoch.

Lines 3-4 If `weight` is 0 then `t` has exhausted its quantum in the current epoch. `goodness` returns 0 to indicate this.

Lines 5-6 `t` gets a huge bonus if the last CPU that executed it is the CPU upon which it is a candidate to execute now. Giving `t` this bonus is equivalent to penalizing tasks migration. Migration is penalized because a migrating task will unavoidably have TLB and cache misses when it starts to execute on a different CPU. However, the value of `PROC_CHANGE_PENALTY` is 15 (at least since Linux-2.2). This means it is 3 times bigger than the maximal value of `counter` ! a fact that seems very strange and is discussed later in section 6.6.2.

Lines 7-8 `t` gets a small bonus if its address space is the same as of the task that is currently executing on `this_cpu`. This bonus may encourage less memory-pages swaps in the very near future. In this work context it may also help a fine grain parallel job's tasks to synchronize (this will be further elaborated in the following chapters). The constant `SAME_ADDRESS_SPACE_BONUS` doesn't really appear in the original code and was named by us for future references as one of the scheduler's parameters. Instead, its value — which is 1 — is hard coded in the algorithm.

Line 9 Finally, the goodness value is returned.

6.5.2 The `reschedule_idle` Function

The `reschedule_idle` function is invoked both by `schedule` and by `_wake_up_common` as will be described later. It gets a task as its argument and checks whether it can be scheduled on some CPU; preferably on an idle one, but if there aren't any, by preempting a less desirable task. The pseudo code of `reschedule_idle` is presented in algorithm 2. The reader shouldn't be overly impressed by the apparent simplicity of the pseudo code, as the real implementation has little resemblance to it. However, the pseudo code does faithfully describe the essence of the algorithm. This is the appropriate place to mention that in Linux, each CPU has a a special task which is called the *idle task*. These tasks are special in the sense that they are different (each CPU has a different task), but share the same id which is 0 (no, it is not "the swapper", it's the idle task). Whenever a CPU is idle, the "current" executing task is the idle task.

Algorithm 2 The `reschedule_idle` function pseudo code.

```
 1 reschedule_idle(task t) {
 2
 3   next_cpu ← NIL
 4   if( t.processor is idle )
 5       next_cpu ← t.processor
 6   else if( there exists an idle cpu )
 7       next_cpu ← least recently active idle cpu
 8   else
 9       max_prio ← PREEMPTION_THRESHOLD
10       foreach cpu c in [all cpus]
11           diff ← goodness(t,c) - goodness(c.current_task,c)
12           if( diff > max_prio )
13               max_prio ← diff
14               next_cpu ← c
15
16   if( next_cpu ≠ NIL )
17       prev ← next_cpu.current_task.need_resched
18       next_cpu.current_task.need_resched ← true
19       if( (prev = false) and (next_cpu ≠ this_cpu)  )
20           interrupt next_cpu
21 }
```

Line 3 The purpose of the `next_cpu` variable is to hold the CPU on which `t` will possibly be scheduled in a short while. `next_cpu` is initialized to a non valid value. Towards the end of the function (line 16) this value will be tested, if it's still `NIL` this means that no suitable CPU was found for `t`. Otherwise, `schedule` will be invoked for `next_cpu` in a short while.

Lines 4-5 `t.processor` is the best CPU for `t` to run on because this CPU's cache may still hold relevant values for `t`'s context. These lines ensures that if `t.processor` is idle, it will indeed be chosen as the next CPU.

Lines 6-7 If the `t`'s previous CPU isn't idle, try to find another idle CPU. The algorithm prefers the least recently active idle CPU because "it will have the least active cache context" (quote from the actual code).

Lines 8-14 If reached here, there are currently no idle CPUs. `diff` is defined to be the difference between the goodness of tasks `t` and `c.current_task` (on CPU `c`). The algorithm searches for `c` with

the maximal `diff`. There is an initial constraint on such a `c`: it's not enough that `diff` will be positive (which means that `t` is more desirable than `c.current_task` on `c`), it is also required that the goodness difference will be above some threshold, namely: `PREEMPTION_THRESHOLD`. Similarly to `SAME_ADDRESS_SPACE_BONUS`, The constant `PREEMPTION_THRESHOLD` doesn't really appear in the original code and was named by us for future references as one of the scheduler's parameters. Instead, its value — which is also 1 — is hard coded in the algorithm.

Lines 16-18 As stated above, if a `next_cpu` was found, then either it's idle or its currently running task is less desirable than `t`. In any case the `next_cpu.current_task.need_resched` flag is set (not before saving its old value) which means `schedule` will be invoked on `next_cpu` in a very short while. Note that there's no guaranty `t` will be the next task chosen by `schedule`, only that it will be invoked. This is true because it's possible there are even more desirable tasks than `t` on `next_cpu` in the ready queue.

Lines 19-20 These lines take care of the case in which the `need_resched` flag of `next_cpu` was indeed changed (and it's a different processor then the one which is currently executing the `reschedule_idle` code). In this case `next_cpu` must be somehow notified that its current task's `need_resched` flag was updated. For this purpose the `this_cpu` interrupts `next_cpu` using some interprocessor interrupt instruction.

6.5.3 The `__wake_up_common` Function

When a task is waiting for some event to occur (e.g. input arrival, semaphore increment etc.), it is removed from the ready-to-run task list and placed in some queue - `q` - associated with this event (in the context of this work, `q` is what we refer to as "blocked mode"). The pseudo code of `__wake_up_common` is presented in algorithm 3 and is self explanatory. Although the code is very simple and straight forward, it seems we have detected a bug (or a serious misfeature) in it. This will be further elaborated in section 6.6.1

Algorithm 3 The `__wake_up_common` function pseudo code.

```
__wake_up_common(wait_queue q) {
    foreach task t in [q]
        remove t from q
        add t to ready-to-run-list
        reschedule_idle(t)
}
```

6.5.4 The `schedule` Function

The `schedule` function implements the scheduler proper. Its objective is to find a task in the ready queue and then assign the CPU (that actually executes the code) to it. This function is invoked directly or indirectly by several kernel routines:

Direct invocation The scheduler is invoked directly when the current task must be blocked right away because the resource it needs is not available. In this case the kernel routine should insert itself to the proper wait queue, change its state from runnable to interruptible and invoke `schedule`. As described in the `__wake_up_common` section, the routine will be resumed exactly from where it left of when the resource will become available. The scheduler is also directly invoked by many device drivers that execute long iterative tasks. At each iteration cycle, the driver checks the value of the `need_resched` flag and if necessary, invokes schedule to voluntarily relinquish the CPU.

Lazy invocation As explained earlier, the scheduler can also be invoked in a lazy way by setting the `need_resched` field of the current executing task to 1. Since a check on the value of this field is always made before resuming the execution of a user mode task, `schedule` will definitely be

invoked at some close future time. For example, this flag is set by the kernel's interrupt handler whenever the `counter` of the current executing task reaches 0.

The pseudo code of `schedule` is presented in algorithm 4. It is stripped from all synchronization, accounting and other administrative details.

Algorithm 4 The `schedule` function pseudo code.

```
 1 schedule(cpu this_cpu) {
 2
 3   prev ← this_cpu.current_task
 4
 5   if( prev's state is runnable )
 6       next   ← prev
 7       next_g ← goodness(prev, this_cpu)
 8   else
 9       next_g ← -1
10
11   foreach task t in [runnable and not executing]
12       cur_g  ← goodness(t, this_cpu)
13       if( cur_g > next_g )
14           next   ← t
15           next_g ← cur_g
16
17   if( next_g = -1 )                    /* no ready tasks  */
18       end function
19   else if( next_g = 0 )                /* start new epoch */
20       foreach task t
21           t.counter ← t.counter/2 + NICE_TO_TICKS(t.nice)
22       goto 5
23   else if( next ≠ prev )
24       next.processor    ← this_cpu
25       next.need_resched ← false       /* 'next' will run next */
26       switch contexts: from prev to next
27       if prev is still runnable: reschedule_idle(prev)
28
29   /*
30    * 'next' (which may be equal to 'prev') will run next ...
31    */
32 }
```

Line 3 Throughout this function, `prev` is the task that up till now was executing on `this_cpu`.

Lines 5-9 These lines ensure that in case of a (goodness) tie, the scheduler will always prefer `prev` over another runnable task with equal goodness. There's noting to gain by context switching between tasks with equal priorities. It is best to avoid the context switch.

Lines 11-15 This is the loop the tracks the best task to run on `this_cpu` by iterating through all the runnable tasks that are not currently executing and choosing the one with the highest goodness. Note that if even one runnable not executing task exists, `next_g` will be nonnegative at the end of the loop.

Lines 17-18 If `next_g` is negative then there are no ready tasks and there's nothing else schedule can do (in this case the real algorithm sets the idle-task as the "current" task).

Lines 19-22 If `next_g` is zero it means there are runnable ready tasks but they have all exhausted their quantum. This means a new epoch should be started and all the quantum durations are refreshed as explained in section 6.4.4. Note the the loop in line 20 iterates though all the tasks (not just the runnable ones). This is the only means in which the Linux scheduler favors I/O bound over CPU bound tasks. Also note that the formula in line 21 prevents `t.counter` from ever exceeding twice the value of `NICE_TO_TICKS(t.nice)`. After starting the new epoch, the function is restarted.

Lines 23-27 If the condition in line 23 evaluates to be true, then a context switch will soon take place. The scheduler (**a**) updates `next` with its new CPU and turns off its `need_resched` bit, (**b**) performs the context switch [1], and (**c**) tries to assign another CPU to `prev` — the task that had just been preempted.

6.6 Linux-2.4 Scheduler Misfeatures

No scheduler is prefect. There is always the need to balance between different aspects of the system leading to unavoidable tradeoffs. It is probably correct to assume that for almost every proposed scheduler algorithm, it is possible to derive a mix of events that will lead to poor system results. The Linux scheduler is no exception. Many of the Linux scheduler faults have already been discussed (e.g. in [4, chapter 10, pages 291-293]) and it seems pointless to mention them here.

However, while implementing the Linux scheduler in the simulator, we came across (what we consider) misfeatures that were never documented (to our knowledge) and seem strongly related to our work. These misfeatures are described in sections 6.6.1 and 6.6.2. Section 6.6.1 describes the most serious misfeature (or rather, a bug) we've encountered, which is a race condition in `__wake_up_common`. In this section, two algorithm improvements will be suggested to overcome the problem. These improvements will be referenced from later chapters. Section 6.6.2 will discuss the problem with the Linux scheduler parameters' values as mentioned earlier in section 6.5.1 and 6.5.2.

The `schedule` drawback of iterating through the runnable task list in a linear fashion is obvious. As the number of runnable task grow, the cost of context switching becomes grater, a consequence that effects a "spin or block" decision greatly. Section 6.6.3 discusses and demonstrates this effect.

6.6.1 Race Condition in `__wake_up_common`

The algorithm in `__wake_up_common` iterates though the awakening tasks (see algorithm 3), for each such task it invokes `reschedule_idle` (see algorithm 2). `reschedule_idle(t)` is essentially divided to four steps:

1. If `t.processor` is idle then it is chosen as the next CPU.
2. Otherwise, the least recently active idle CPU is chosen, if one exists.
3. Otherwise, the CPU of the task which is least desirable in comparison to `t` is chosen, if one exists.
4. If a next cpu was found, interrupt it if necessary.

The most obvious effect of the race condition is associated of course with step 2. The following is an example of a trivial scenario:

- 4 tasks - $\{t_0, t_1, t_2, t_3\}$ - are awakening on a SMP with 8 processors $\{c_i : i = 0...7\}$ and c_0 is the CPU which executes the code of `__wake_up_common`.
- Assume that for each $i = 0...3$ there exists: $t_i.processor = c_i$.

[1] When line 26 returns, we have changed contexts, and are currently in the context of `next`. A little magic is involved here: It's the 'much more previous' `prev` that is on `next`'s stack, but `prev` is set to (the just run) 'last' process by the procedure that actually performs the context switch. This might sound slightly confusing but really makes tons of sense: for one thing, it makes line 27 operate on the correct "previous" task.

- Further assume that $\{c_i : i = 0...3\}$ are currently busy, while $\{c_i : i = 4...7\}$ are idle because there aren't any other runnable non executing tasks.
- Finally, assume that c_7 is the least recently active idle CPU.

Worst case scenario goes like this:

- The first invocation of `reschedule_idle` is on t_0 which is of course mapped to c_7.
- c_7 is interrupted and begins to execute the `schedule` code.
- However, way before c_7 reaches line 25 (algorithm 4), `__wake_up_common` has already finished its work.
- Since the `need_resched` the status of c_7 was "idle" all through the execution time of `__wake_up_common` , t_1, t_2 and t_3 were also mapped to c_7 .
- The result: there are three idle CPUs - $\{c_i : i = 4...6\}$ - and 3 ready non executing tasks. In other words, three CPUs "got lost".

6.6.1.1 Possible Implications of the Race Condition

The above scenario isn't the only drawback of `__wake_up_common`. The fundamental problem is that an awakening tasks' set might be assigned a processors' set which is smaller than possible. The *possible* implications of this problem are:

1. CPUs get lost (as shown above).
2. Only part of the awakening tasks that may get a hold of a CPU, indeed get one. This doesn't necessarily involves lost CPUs. For example: t_0 and t_1 are awakened while only c_0 is idle. However t_0 has enough priority to preempt t_2 which is the task that is currently executing on c_1. `__wake_up_common` invokes `reschedule_idle`(t_0) which is assigned to c_0 (because its idle) even though t_0 can preempt t_2. Afterwards `reschedule_idle`(t_1) is invoked and fails to find an assignment for t_1. Had the order of the iteration through the awakening tasks been reversed (i.e. first t_1 and then t_0), both tasks would have been assigned a CPU.
3. Context switch overhead might be doubled. This problem is also related to the order of the iteration. For example: t_0 and t_1 are awakening and the only possible assignment for both of them is c (by preempting t_2 which is currently executing on it). Assume that:

   $$goodness(t_0, c) > goodness(t_1, c) + PT > goodness(t_2, c) + 2PT$$

   where *PT* stands for `PREEMPTION_THRESHOLD`. At first, `reschedule_idle`(t_1) is invoked and as a result t_2 is preempted in favor of t_1. Afterwards, `reschedule_idle`(t_0) is invoked and as result t_1 is preempted in favor of t_0. Again, this extra context switch would have been avoided if the order of the iteration through the awakening tasks had been reversed.

6.6.1.2 Wakeup Schemes Used in the Simulator

Simulating this race is extremely hard. However, we may simulate the worst and best case scenarios, in recognition that the truth is somewhere in between. Another possibility is to introduce the obvious fix to `reschedule_idle` (which is described shortly). In light of that, we defined in the simulator the following three wakeup schemes:

SILLY Implements the worst case scenario: `__wake_up_common` first assigns idle CPUs to *all* of the awakening tasks and only then `schedule` is invoked on the chosen CPUs. In this scheme, if an idle CPU exists, all tasks will be assigned to *p*, the least recently active idle CPU (with the exception of tasks that will be assigned their previous CPU, if idle). Therefore only the `need_resched` flag associated with *p* will be set and thus `schedule` will be invoked only for *p*.

SMART Implements an approximation of the best case scenario: All of the per task local considerations done by `_reschedule_idle` are made global across all the awakening tasks. The complexity of this algorithm seems to make it unfit to use in a real system. Nevertheless, it provides us a perspective that will help us evaluate how crucial the wakeup scheme is in the context of this work. Note that an implementation of such an algorithm is a non trivial task (it took us $\approx$ 400 lines of C++ code to do it efficiently). As an example, think of the last phase of the algorithm where it decides which awakened task will be assigned to which busy CPU. The algorithm should somehow generate the following set:

$$TRI \ = \ \Big\{(d,t,c) \ : \ d = goodness(t,c) - goodness(c.current\_task, c) \bigwedge d > PT \Big\}$$

sort it according to the *d* value of the triplets (larger *d*-s come first), and then execute the code specified in algorithm 5. Note that this scheme eliminates the race condition by (**a**) first deciding which CPUs' `need_resched` flag should be set (without actually setting them), (**b**) adding all the awakened tasks to the ready queue, and (**c**) only after that setting the `need_resched` flags decided upon in phase (a).

AIP Implements the obvious fix. AIP stands for: Avoid Idle Pitfall. This greedy algorithm simply modifies `reschedule_idle` such that instead of searching for "just" an idle CPU, the function searches for an idle CPU with an associated off `need_resched` flag. This wakeup scheme is practical and is guaranteed to eliminate the problem of CPUs getting lost (which is an immediate fix to the bug). It may also help with the other problems mentioned earlier.

Algorithm 5 A piece of (pseudo) code from the SMART wakeup scheme.

```
foreach (d,t,c) ∈ TRI
    if( (task t wasn't already assigned a CPU) and
        (CPU c wasn't already assigned to some other task) )
            assign t → c
```

Note that SMART is not the "optimal" wakeup scheme: The problem presented in section 6.6.1.1 is actually equivalent to a maximum-bipartite-matching problem [5] namely finding a maximum bipartite match between the two disjoint sets *T* and *C*, where:

- *T* contains the awakening tasks,
- *C* contains (all) the processors, and
- *E* is the set of edges between *T* and *C* and is defined to be:
  $E \ = \ \{ \ (t,c) \in T \times C \ : \ (\ c$ is idle) or (t may preempt $c.current\_task$) $\}$

However, such an algorithm was not used because it ignores the actual priorities of the awakening tasks. This leads to the following two unwanted results:

1. When given two tasks - t_1, t_2 - which may both run on processor c ; in order to achieve a bigger match, such an algorithm may prefer assigning t_1 to c and leave t_2 without a processor even though:
   $$goodness(t_2, c) \ > \ goodness(t_1, c)$$
2. `schedule` of course is not aware of such an algorithm's considerations (recall that it is lazy invoked) and will choose t_2 anyway (there's a race condition here too).

Implications on next chapters' simulations: For each simulation that we will conduct in the following chapters, we will specify the wakeup scheme used.

6.6.2 Tunable Scheduler Parameters

While describing the algorithm in former sections, we came across three tunable scheduler parameters:

1. `PROC_CHANGE_PENALTY` (=15, used in `goodness`)
2. `SAME_ADDRESS_BONUS` (=1, used in `goodness`)
3. `PREEMPTION_THRESHOLD` (=1, used in `reschedule_idle`)

The first one actually appears in the code whereas the other two are hardcoded (and therefore named by us). At this point, it is important the reader would be aware of the fact that the default quantum length was changed from 20 ticks (=200ms) in Linux-2.2 to 5 ticks (=50 ms) in Linux 2.4. However, the above parameters didn't change accordingly. It follow that:

Parameter	value	% of quantum in Linux-2.2	% of quantum in Linux-2.4
PROC_CHANGE_PENALTY	15	75%	300%
SAME_ADDRESS_BONUS	1	5%	20%
PREEMPTION_THRESHOLD	1	5%	20%

which means a considerable change in the scheduler behavior. The most obvious change is that in Linux-2.2 an awakening task t_1 had the ability to preempt an executing task t_2 on a CPU c , even if $t_1.processor \neq c$: In terms of goodness values it's possible when: $t_1.counter - t_2.counter > 16$ (=SAME_ADDRESS_SPACE_BONUS+PROC_CHANGE_PENALTY). Recall that an I/O bound task may (and usually does) accumulate a `counter` value of up to twice the default quantum duration (see line 21 in algorithm 4) which translates in Linux-2.2 to an upper bound of 40 ticks on `counter`. As a result, a task with `counter`=18...40 had a chance to preempt another task, even when migration was involved. It's therefore safe to speculate that a preemption of a CPU-bound-task in favor of an I/O-bound-task involving the I/O-bound-task's migration, wasn't a rare event in Linux-2.2. However in Linux-2.4 such an event is impossible. After consulting with some Linux developers, we believe somebody simply forgot to update these values along with the change of the quantum default duration.

Another important point to make is that the resolution of the scheduler (as explained in section 6.4.3) seems to be too coarse. For example, even if we change `PREEMPTION_THRESHOLD` to 0 in Linux-2.4, the threshold will still be bigger than it was in Linux-2.2 (because the difference must be at least one tick, which is 20% of quantum in Linux-2.4). Recent research [7] coincides with this conclusion.

Implications on next chapters' simulations: Unless stated otherwise, the scheduling algorithm used in the simulations in the following chapters use the parameter values of Linux-2.2 (rather then Linux-2.4) as specified above. In addition, the `counter` field of each task participating in a simulation has cycle accuracy.

6.6.3 Linearity of `schedule`

After reviewing the Linux scheduler, we got tempted to measure the cost of the linear iteration through the ready to run task list in `schedule` (algorithm 4, line 11). Though not having a direct implication on this work, it was interesting to see "how bad is it" since the duration of a context switch has direct implications on a "spin vs. block" decision. This is the sole practical adventure we've embarked upon within this work. In order to measure the duration of quantum as a function of load, we did the following:

- Modified `schedule` code as follows:
  - A cycle measurement is taken at the beginning of `schedule` and just before line 26 (the lines that actually switched contexts). Note that the procedure that actually performs the context switch is quite short. The main complexity factor of `schedule` is its linear iteration through the ready queue.

  - The difference between the two cycle measurements and the current number of runnable tasks are copied to the kernel cyclic log buffer.
  - A boolean flag `do_log` was added to `schedule` to "guard" the additional code we've added to it. The flag was needed because we wanted to achieve a certain load before starting the logging process and to stop logging before this load decrements. This flag was initialized to false.

- Added a module to the kernel which allows controlling the value of `do_log` from user space (using the /proc mechanism).

- The klogd daemon is a user level process the "listens" to the log messages printed by the kernel (to its cyclic log buffer) and forwards them (via the syslog system call) to the syslogd (it actually a kind of proxy). The syslogd in turn usually writes the messages it gets in an unbuffered fashion to some log file. This logging mechanism was unsatisfying because our modified `schedule` version prints very fast *a lot* of messages and many of them got lost along the way due to all the various overheads (the extra copy from klogd to syslogd via a system call and the unbuffered write of syslogd). We therefore modified the code of klogd to intercept our log messages and (buffered) write them to some log file.

- Next we've written a script that generates a certain load and after this load is achieved, turns on `do_log`, waits for a few seconds, and turns it off (starting the logging only after the designated load is reached turned out to be a non trivial assignment).

- Finally, the log was analyzed and the graph presented in figure 6.1 was produced.

The measurements were taken on a quad Pentium III 550MHz IBM NetFinity server with 1GB RAM running Linux-2.2.18 (which was the latest release when these measurements were taken). Approximately 1.5×10^6 measurements were taken. The cost of schedule with a load of up to 800 runnable tasks is unacceptable. The Y-axis ends at 120,000 cycles but the worst results measured was 566,584 cycles ! (which is $\sim$ 1 millisecond on this machine).

When reviewing the results, the question that comes to mind is "why create a load of 800 runnable tasks on a 4 CPU machine" ? The answer is divided to two parts. The first one is that the results were just as bad even if the measurements were performed on a 64 CPU machine, because the runnable list was of the same size. In fact the results would probably be worse due to contention for the synchronization mechanism. The second part of the answer will shortly follow. A very interesting and relevant discussion revolved around this issue in the Linux kernel mailing list with the participation of Linus Torvalds, Alen Cox and others [27]. The discussion thread's title was quite appropriately: "a quest for a better scheduler". The main questions discussed were:

- whether applications that use several hundreds runnable task should be supported by Linux (which is optimized for a small box),

- if it makes sense to write a parallel application that use a number of runnable threads which is much bigger than the number of CPUs of the machine,

- and do such applications exist

The majority's opinion seemed to be: yes (to all questions). Some think its the application responsibility not to create a number of runnable tasks which is considerably bigger than the number of processors. The counter opinion is that this means every application that must manage a (possibly virtual) context of many tasks, must implement a scheduler by itself. This coincides with our opinion: if people write such applications, the scheduler should support them.

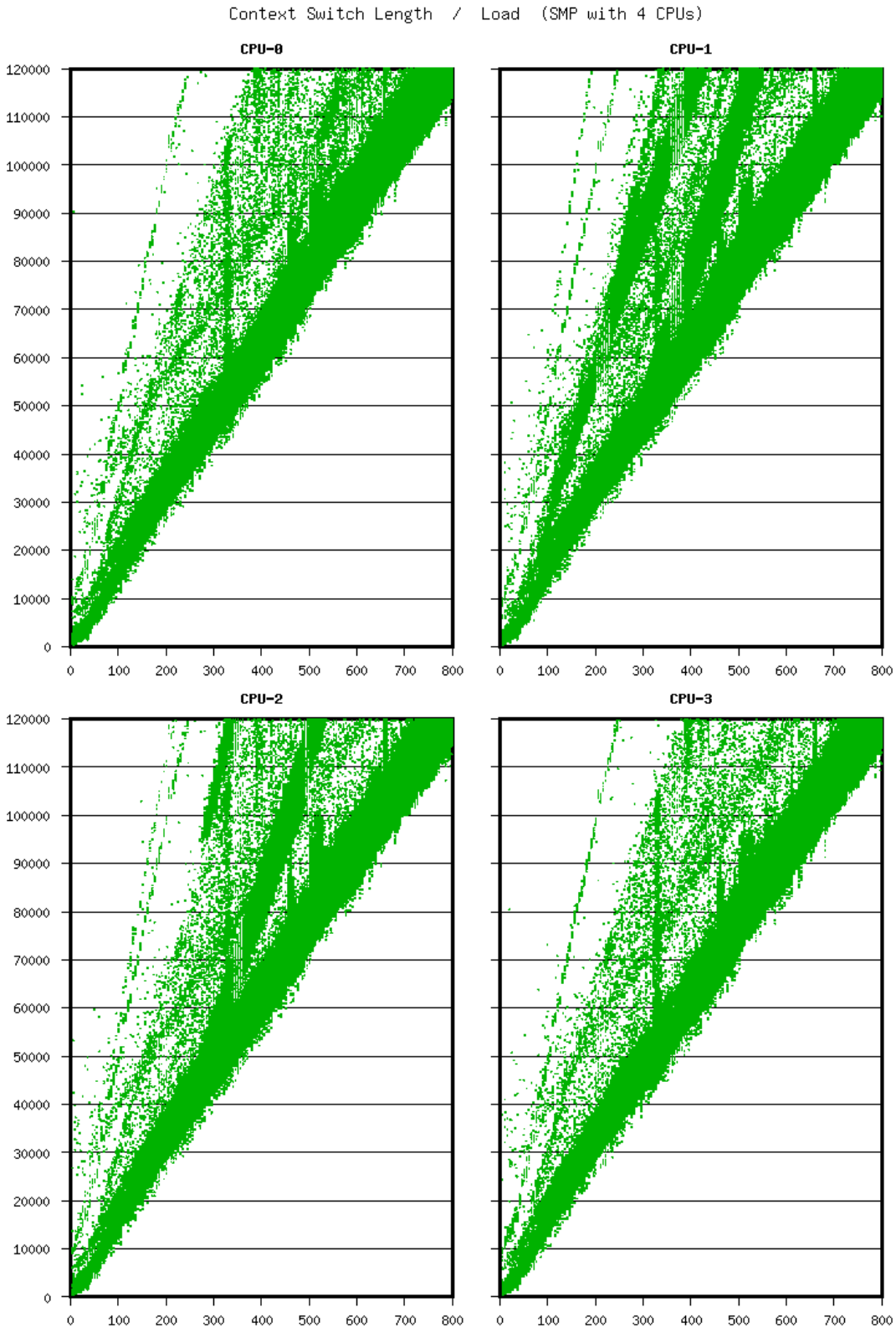

Figure 6.1: The X axis specifies the number of runnable processes in the system i.e. the load (5-800 tasks). The Y axis specifies the number of cycles consumed while context switching (only samples below 120,000 cycles are displayed). It's clear there's a linear dependency between the load and the lower/upper bounds of the context switch duration.

A concrete example was given for such an application: Running DB2 on an SMP system. In DB2 there is a processes/thread pool that is sized based on memory and the number of CPUs. The size of this pool is in the order of 100s for an 8-way system with reasonable sized database. A <maxagents> parameter determines the number of agents that can simultaneously execute an SQL statement. Requests are flying in for transactions. The agents are grabbed from the pool and concurrently fire the SQL transactions. Assuming that there is enough concurrency in the database, there is no reason to believe that the majority of those active agents is not effectively running. Of course limiting the number of agents would reduce concurrently running tasks, but would limit the responsiveness of the system.

Related work: There has been significant progress in achieving a working Linux scheduler which is (a) scalable, and (b) handles the case in which the number of runnable tasks is much bigger than the number of processors. However it hadn't found its way yet to the official Linux release. The effort is led by the IBM Linux Technology Center and a summary of this work (including the actual code of such working algorithms) may be found in [11].

Implications on next chapters' simulations: The maximal load generated by most simulations (in previous and following chapters) is usually a few hundreds (on a 32/64 CPU machine) which according to the above discussion is realistic. The linearity fault of the Linux scheduler is ignored because:

1. It is solvable [19] (while maintaining the existing scheduler behavior and semantics).
2. Even for large number of tasks, the maximal context switch duration we found, is still a very small fraction of quantum (according to our measurements worst case is:

$$\frac{max\,context\,switch\,measured}{speed\,of\,machine\ \times\ quantum\,duration} = \frac{566,584\,cycles}{550MHz\ \times\ 0.05sec} \approx 2\%$$

   of quantum) which we just use as an upper bound on the context switch overhead anyway.
3. Our main focus is on loads for which: $task\_num\ \leq\ 2 \times cpu\_num \pm$ (since on bigger loads spinning almost always fails).

Chapter 7

Synchronization Job in a Non-Synchronizing Environment Under the Linux Scheduler

7.1 Introduction

After introducing the Linux scheduler in the previous chapter, we will now follow the "round-robin path" and conduct a series of simulations with increasing complexity, the first one of which is of a single synchronizing job in a non-synchronizing environment. Before performing any simulations, section 7.2 will describe the minor changes made in the simulator in order to support the Linux SCHED_OTHER scheduling algorithm. Afterwards, section 7.3 and 7.4 will describe the first simulation and present its results. The remaining sections will be dedicated to analyzing and understanding these results and their implications.

7.2 Simulator Changes

Obviously, all the data structures and algorithm presented in chapter 6 had to be embedded into the simulator. Aside from that, the change in the simulator was minor: the decision of "which is the next thread to run" changed from "the first thread in the ready queue" to "the one determined by `schedule`". The structure of the simulator (see figure 2.1 page 16) has therefore remained almost the same. Recall that while executing, a thread may have one of the three SMP states:

ready The thread may run and is waiting in the ready queue to be allocated a CPU.

running The thread is currently running on some CPU. Recall that while it is being preempted (by the simulator-event with the duration associated with the parameter context-switch-out) or being scheduled (by the simulator-event with the duration associated with the parameter context-switch-in), a thread is considered to be in "running" state.

blocked The thread was preempted and is currently waiting for the other threads of its job to reach the next synchronization point.

In addition to the above states, while implementing the SCHED_OTHER we've added a fourth simulator state: *wait4cs* (stands for *waiting for context switch*). The reason it was needed is as follows:

- Let t_1 be an executing thread that has just "informed" the SMP it's yielding its processor *c* (either because its quantum is exhausted or because it finished its fixed spin period, failed, and is about to enter blocked mode).

- Now, in order to simulate `schedule`, the actions taken by the SMP are (**a**) to determine which will be the next thread — t_2 — that will run on *c* (assume $t_1 \neq t_2$), and (**b**) to push to the event-queue a context-switch-out-event — e_1 — on behalf of t_1. Recall that until e_1 expires, t_1 is considered to be the thread that is currently running on *c*.

- In the meantime (until e_1 will expire), t_2 is removed from the ready-queue so that it will not be chosen by other `schedule` invocations to be the next thread to run on some other processor different than *c*. It is then inserted to the container associated with the "wait4cs" mode.

- When e_1 finally expires, only then the SMP pushes a context-switch-in event — e_2 — on behalf of t_2 which is removed from "wait4cs" mode and is assigned to *c* (thus changing its state to "running"). We remark that the time consumed in order to execute e_1 and e_2 represents the duration of `schedule` (and the other operations performed by the kernel before switching back to t_2's user context).

The "wait4cs" state therefore fits in figure 2.1 as a circle between "ready" and "running", and the arrow that passes through it is the one associated with a thread being allocated a processor.

7.3 Simulation Description

As the first step, we conducted a simulation which is almost identical to the one conducted throughout chapter 3. Namely, a synchronizing job composed from 11 threads executing on a machine with 32 CPUs within an increasingly growing load of non-synchronizing threads. The difference of course is in the scheduling algorithm which was previously SCHED_RR and currently is SCHED_OTHER. We will skip the first few steps of gradually adding randomization and directly jump into the deep water of a fully randomized simulation (computation intervals are normally distributed and the ready queue is shuffled on startup). For starters, we will use the SMART wakeup scheme (the various schemes will be compared later in this chapter). The simulator parameters we use are therefore:

p	q	in	out	sync	nosync	barrier	spin	μ	σ	rand ord	wakeup scheme
32	100	3%	3%	11	0...200	50	6%	1% 10% 100%	90/15%	1	SMART

7.4 Results

The results of the first simulation are displayed in figure 7.1. After briefly investigating the simulation events in order to explain the many peaks and valleys displayed by the fine/medium grain curves, we've quickly reached the conclusion that similarly to the the round-robin-heterogeneous simulations, if we prolong the duration of the computation (i.e. increase the number of barriers), the resulting SSR curves stabilize and a clearer picture is received. This is demonstrated in figure 7.2 in which we present the results of the original simulation after the number of barriers was increased to 2000, 5000 and 10000. When analyzing the resulting graph we see that:

1. When we increase the barrier number, the curves associated with the various μ values become almost indistinguishable, which means that when we prolong the duration of the computation, the SSR of each simulation is converging to some value.

2. There is a peak in the SSR curves whenever the number of non-synchronizing threads is: $21 + 16n$ i.e. whenever the total load is $32 + 16n$ (when adding the 11 threads of the synchronizing job) .

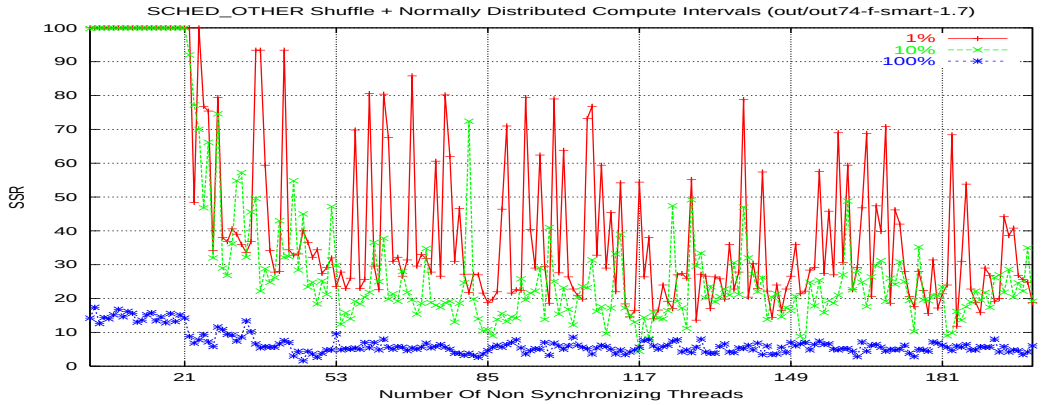

Figure 7.1: The SSR of an 11-sized synchronizing job within a non-synchronizing environment. Each curve is associated with a different μ value: fine grain (1% of quantum), medium grain (10%) and coarse grain (100%). The curves associated with the fine and medium grain job present many peaks and valleys.

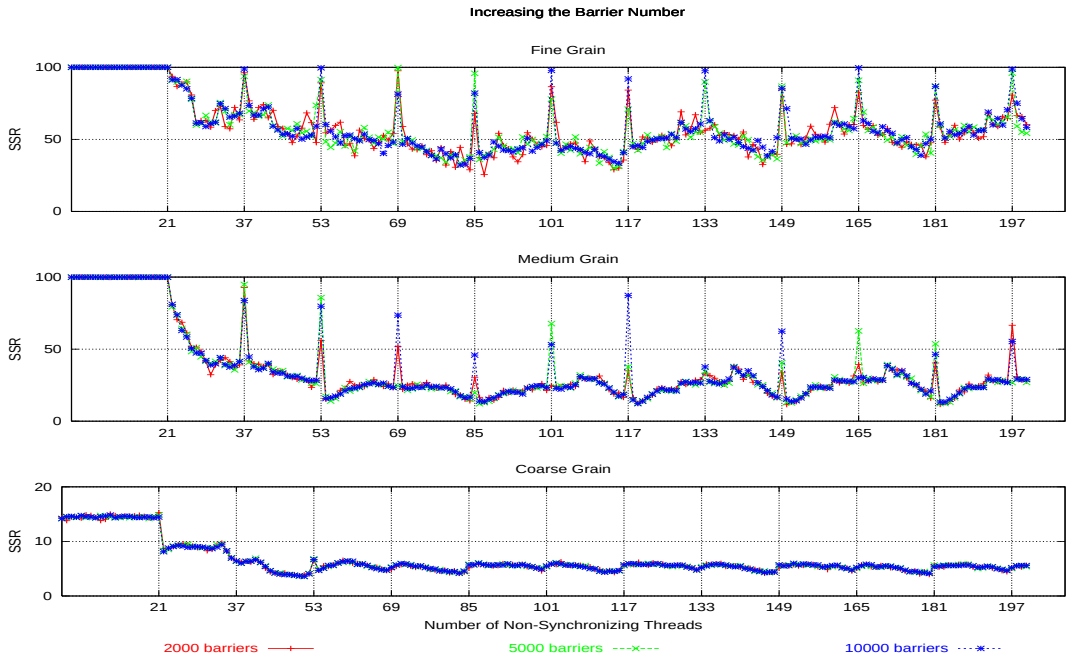

Figure 7.2: The SSR achieved by the original simulation when prolonging the duration of the computation by increasing the barrier number from 50 to 2000/5000/10000. The SSR curves are almost indistinguishable. There's always a peak when the total number of threads is a multiple of 16. Both medium and fine grain display an intermediate load with SSR > 50. Fine grain curves continue to do so on and off for very high loads.

3. Both medium and fine grain jobs display an SSR $> 50\%$ in the intermediate load (for the fine-grain job, this intermediate load spans up to a thread surplus of CPU#; for the medium grain job, the surplus is only 5-6 threads). As usual, the fine grain job achieves better results than the medium grain job while the coarse grain job fails in almost every spin, even when the machine is not fully utilized.

4. Aside from the peaks (when total load is a multiple of 16), fine grain simulations sometimes manage to achieve SSR $> 50\%$ on very high loads. The reader might think the fact that the SSR associated with the fine grain job gets a little higher as the number of non-synchronizing threads reaches 200 has significance. It doesn't. Different seeds produced different results while the large picture (as currently described) has remained the same.

5. The nature of the curves is quite similar to those displayed in the associated round robin simulation (that was conducted without randomization, figure 3.4, page 26).

7.5 Analysis

When trying to understand the results of the simulation, a good place to start seems to be in figuring out the reason for the fine grain peaks that reach up to almost 100% SSR when the number of participating threads (the synchronizing job included) equals $32 + 16n$. The conjecture that the simulation has a cycle (based on analysis of the similar round robin simulation) will be soon proved wrong.

7.5.1 The Transition Point

Let *J* be the synchronizing job. The first step in trying to understand the nature of the peaks was to use the tool developed in chapter 4 to monitor the distribution of *J*'s threads as it changes in time among the various SMP states. Figure 7.3 presents the results of this monitoring on fine grain jobs. Three rectangles are displayed: the first one is associated with *J*'s threads distribution within the simulation that included a total number of $80\ (= 16\times5)$ threads (including *J*), the middle with the simulation of the size $96\ (= 16\times6)$, and the last with the simulation of the size $112\ (= 16 \times 7)$. The X axis displays the time (in cycles). The Y axis displays the number of threads, and each different color represent one of the four SMP states (wait4cs as defined above included). The arrows at the top of each rectangle denote the time instance in which a new epoch was started (see section 6.4.1 and algorithm 4 lines 19...22 in page 61). When we examined *J*'s threads distribution among the various SMP states over time, we noticed that in all of them there exists a *transition-point* which is defined be a time instance such that:

1. prior to it, *J*'s threads computed separately (as indicated by all the "staircases" that came before it), and

2. after it, *J*'s threads manage to group and compute together (as indicated by the green=running and red=ready continuities) until the simulation ends.

For each simulation, the X-axis time range displayed was chosen such that the transition point will be displayed. The transition point of the 80 sized simulation is somewhere between time=600...700, for the 96 sized simulation it's between time=21,500...22,000 , and for the 112 it's in the neighborhood of time=52,000.

7.5.2 *J*'s Point of View

Let's focus on the first rectangle associated with the 80 threads load and describe what's going on there form *J*'s point of view:

- On startup, only 3 of *J*'s threads "got lucky" and were allocated CPUs. These threads compute for a very short while (recall that this is a fine grain job i.e. it's μ=1% of quantum which is composed of 100 cycles), reach the first barrier, spin for a while, fail, and block.

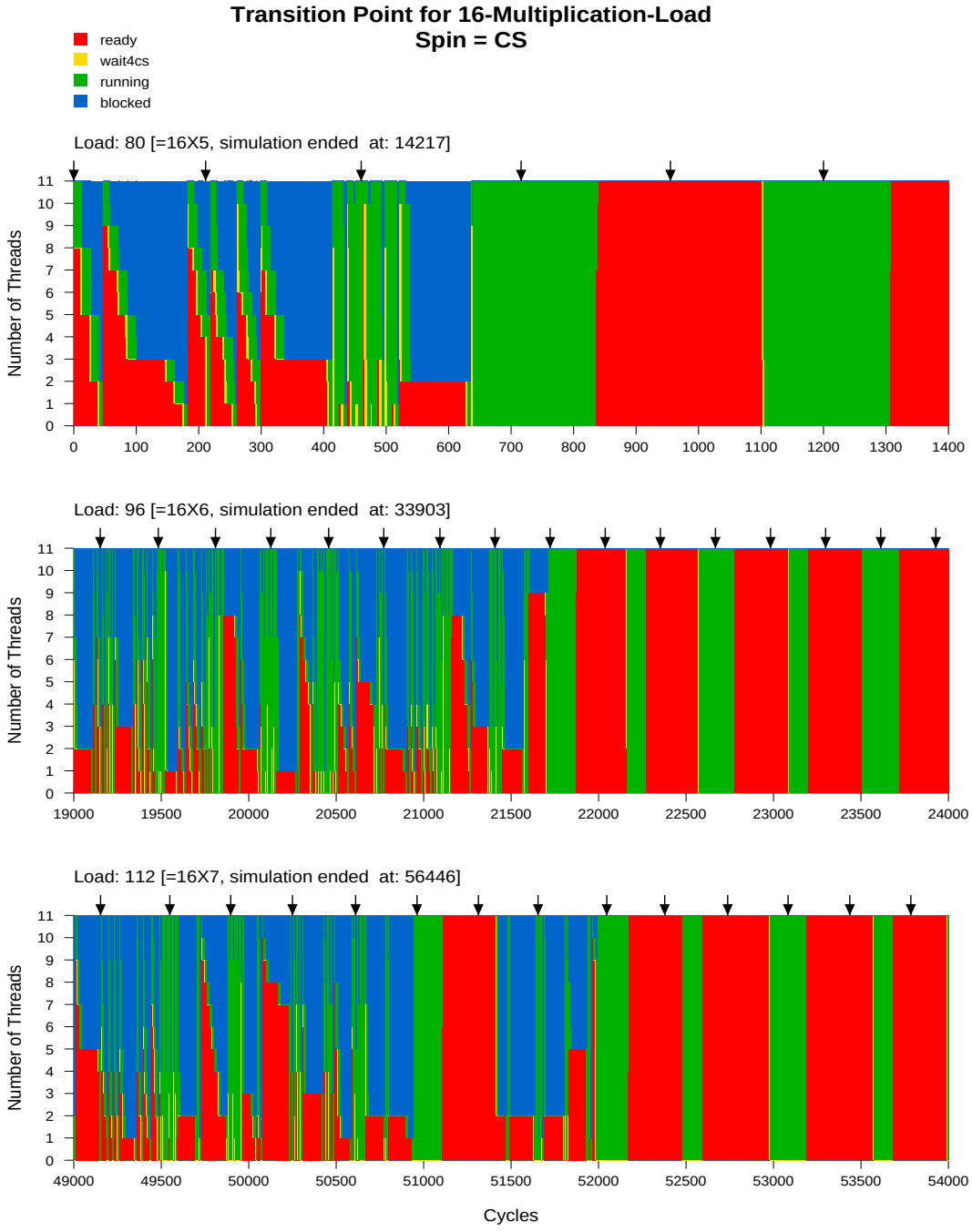

Figure 7.3: This figure displays the distribution of J's threads among the four SMP states as a function of time (cycles). The jobs were taken from simulations with $load = CPU\# + 16n$. The maximal spin duration performed by jobs that are displayed here equals the duration of a context switch (CS). The arrows at the top of each rectangle indicate when a new epoch was started. The X-range was chosen such that the transition point of each job will be displayed. After the transition point, threads are grouped and perform their computation together and thus all barriers are successful. Before the transition point, the jobs either perform a "tail chasing" alt synchronization or are confined to a small number of CPUs.

- As a result of this blocking, their CPUs are "up for grabs" and they are immediately assigned to 3 other ready threads. Evidently, these new three threads also belong to *J* (as indicated by the second green "step" in the first "staircase"). This wasn't just a low probability event that happened: the 3 original threads were substituted by threads from the same job even though currently there are 40 non-synchronizing and only 8 synchronizing threads in the ready queue, because of the `SAME_ADDRESS_SPACE_BONUS`. Since all the 40 non-synchronizing threads in the ready-queue don't share address space with the original three, while the other 8 do, then the goodness of the latter is higher than of the former which results in the choosing of another 3 of *J*'s threads.

- The second thread threesome also spin, fail and blocks only to be replaced by the third threesome of J's threads which soon enough also block. Now, as indicated by the size of the last green step in the first staircase, there are only 2 of *J*'s threads in the ready-queue (the other 9 are blocked) and therefore *J* "looses" one CPU in favor of some non-synchronizing thread.

- The last pair completes the first barrier (causing the other 9 threads of *J* to change state from blocked to ready), reach the second barrier, spin, fail, and block. For the same reason as stated before (`SAME_ADDRESS_SPACE_BONUS`) this pair is replaced by another pair of *J*'s threads ...

- This scenario repeats itself (in various forms) in the first 6 "staircases". The first epoch that ended somewhere between time=200...300 didn't change anything. However, towards the end of the second epoch just after time=400, the computation pattern changes and *J*'s threads effectively get hold of 11 CPUs at the same time (as indicated by the 11-threads-wide green=running lines). For some reason, *J* seems to "chase its tail", in what can only be described as (you've guessed it) alternating synchronization (this will be further elaborated later). The difference between the alt synchronization displayed here and in previous chapters, is the time spent by *J*'s threads in the ready queue between each two consecutive barriers. In previous chapters, when SCHED_RR was used, this period was long: After moving from blocked state to the end of the ready queue, in order to execute again, a thread had to wait until such time when it was the first thread in the ready-queue. This is not necessarily the case for SCHED_OTHER as indicated by the very thin (or simply non existent) red=ready lines between time=400...500. When a thread returns form blocked to ready, `reschedule_idle` is invoked, allowing threads with high enough goodness to preempt other threads and begin to run immediately.

- Towards the end of the third epoch, somewhere between time=600...700, something happens which allows all of *J*'s threads to get a hold of a CPU simultaneously and begin to compute together (this is the transition point). From that point onwards, this type of computation is maintained until the end of the simulation.

7.5.3 The Effective CPU Set

A clue as to why *J* behaves like this is given to us when we notice that this type of tail-chasing alt-synchronization happens *always* just before the starting of a new epoch (see also the other two rectangles). The reason becomes clear when we make the following two observations:

1. Towards the end of an epoch, the `counter` of most non-synchronizing threads is zero or close to it (otherwise we wouldn't have been close to the start of a new epoch). All of the non-synchronizing threads in the ready queue have zero `counter` (a necessary condition for a new epoch to start) and some of those that are currently running have a `counter` that is close to zero (since they have been running for a while).

2. This is not the case for synchronizing threads that spend most of their time in blocked mode. In fact, until the transition point, their `counter` is usually bigger than the default quantum duration i.e. up to 200 cycles (recall that "I/O bound" threads are favored by SCHED_OTHER due to the formula used in `schedule` to refresh `counter` fields at the beginning of a new epoch, which allows a thread to accumulate up to twice its default quantum).

Let ECS_J (the **E**ffective **C**PU **S**et of J) be defined as follows:

$$ECS_J \;=\; \{c \,:\, \exists t \in J \,,\, t.processor = c\}$$

It follows that towards the end of an epoch, the priority of J's threads is very high (in comparison to the priority of non-synchronizing threads), so high, it allows all of J's threads to overcome the `PROC_CHANGE_PENALTY` penalty and get hold of more and more CPUs until $|ECS_J| = |J|$. However, while J alt synchronizes ("enjoying" the fact that $|ECS_J| = |J|$ and that all its threads have much higher goodness values than the non-synchronizing threads that share with them the CPUs in ECS_J), a point is reached when `schedule` needs to find the next thread to run on some CPU and there are no threads in the ready queue with a positive goodness value. This is when a new epoch is started. All the `counter`s of the non-synchronizing threads are refreshed, causing these threads (like Popeye after eating a spinach can [24]) to instantly become "stronger". Indeed, J's threads' `counter`s are also refreshed, but this is done with a decaying factor. Meanwhile, J's threads are continuously alt synchronizing (as J "chases its tail") and therefore rapidly become "weaker" due to the unsuccessful spins. Soon enough, those non synchronizing threads that also periodically run on ECS_J become "strong" enough not to allow each of J's thread to preempt them whenever it returns from blocked to ready state. Unavoidably, ECS_J begins to shrink and the staircase-scenario described above reoccur.

This scenario has the nature of repeating itself over and over again. This fact is illustrated in figure 7.4 that displays the evolving of ECS_J in the 96 sized simulation over the first 10000 cycles.

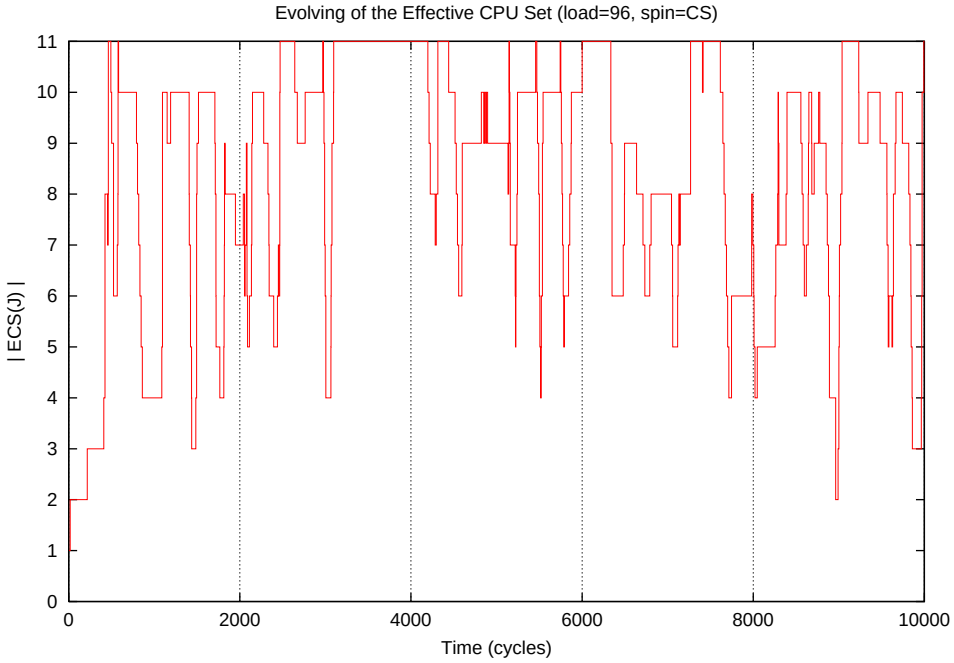

Figure 7.4: This figure displays the evolving of ECS_J of the 96-sized simulation as a function of time. The size of this job has the nature of expending (towards the a new epoch) and shrinking (shortly after).

7.5.4 Reason for Tail-Chasing Alt Synchronization

We therefore conclude that there's a time interval, spanning from shortly before the beginning of a new epoch till shortly after it, in which:

1. $|ECS_J| = |J|$, and
2. the goodness of J's thread is much higher than the goodness of each non-synchronizing thread — t — for which $t.processor \in ECS_J$

The first fundamental question that follows is why does *J* alt synchronize under this circumstances i.e. if

1. all of *J*'s threads have the opportunity to execute together (as if the machine was dedicated to them), and
2. those threads that first arrive to a barrier, wait while spinning for the duration it takes to perform a context switch, thus enabling the other threads to join in,

then what's stopping *J*'s threads from computing together without having to block after each barrier. The second question is what eventually breaks this pattern and allows *J*'s threads to compute simultaneously without alt synchronization (i.e. what is the cause of the transition point). The answer to these questions is a combination of two factors:

1. The serial nature of the simulator: Two simulator events with the same execution time are not executed simultaneously by the simulator because it's a serial program. Instead, they are serialized according to the id of the CPU on which the events are "executed". For example: if event e_1 is associated with a thread that executes on CPU $c1$, and event e_2 is associated with a thread that executes on CPU c_2, then e_1 will occur before e_2 if and only if $c_1 < c_2$ ($c_1 \neq c_2$ because two thread can't execute on the same CPU).
2. The spin interval is too small: Let *CS* (= **C**ontext **S**witch) denote the time it takes to conduct a full context switch (i.e the sum of the time consumed by context-switch-in and context-switch-out events). Since the fixed spin interval used by *J* is exactly CS, a waiting thread — t_1 — spins just enough for an awakened thread — t_2 — to join it. However, when t_2 finally begins to compute and is about to synchronize, t_1 "gives up" and blocks. In order for this spin to have succeeded, t_1 should have spun just a little bit more.

For example, We will now describe the exact chain of events that led to the "tail chasing" alt synchronization in the 80 sized simulation between time≈400...500. We use t_r^c to denote the thread of *J* with rank *r* for which the `processor` field value is currently *c*. We use b_i to denote the i-th barrier. Our description starts at time=414. At this time $A = \{t_2, t_4, t_9\}$ are executing and $B = \{t_0, t_1, t_3, t_5, t_6, t_7, t_8, t_{10}\}$ are in blocked mode after failing to synchronize on b_5.

Time=414 A's threads reach b_5 and finish it successfully. As a result, B's threads are awakened. All of them are able to preempt currently executing (low priority) non-synchronizing threads.

Time=415 A's threads finish the computation phase.

Time=416 A's threads reach b_6 and start to spin: they will do so until no later than time=422 since the maximal spin duration is set to be CS=6.

Time=417 The context-switch-out events of the low priority non synchronizing threads, that were triggered by the awakening *B* at time=414, have expired. Consequently, the context-switch-in events for *B*'s threads are pushed.

Time=420 The context-switch-in events associated with B's threads (triggered at time=417) have expired. B's threads start to compute.

Time=421 B's threads finish the computation phase.

Time=422 Two things happen "simultaneously": (1) B's threads reach b_6 for the first time, and (2) A's threads reached the maximal spin duration (since spinning began at time=416). As stated before, events are serialized according to the CPUs they are executed upon. Currently: $A = \left\{t_9^{13}, t_4^{14}, t_2^{30}\right\}$ and $B = \left\{t_0^{1}, t_1^{9}, t_3^{10}, t_7^{20}, t_5^{21}, t_{10}^{22}, t_6^{23}, t_8^{24}\right\}$. Therefore, when t_9^{13}'s last synchronization attempt is made, it fails, causing t_9^{13} to block (because the first synchronization events of B's threads with CPU ids bigger than 13 weren't executed yet).

Afterwards, the same scenario will repeat itself with the difference that now B's threads will wait for A's threads.

7.5.5 Necessary and Sufficient Condition for Transition Point

The previous subsection explained the reason for the "tail-chasing" alt synchronization. This one will explain the circumstances in which this computation pattern is broken and the transition point occurs:

- Let A_{min} be the minimal CPU of a thread in A ($A_{min} = 13$ in the above example).
- Let A_{max} be the maximal CPU of a thread in A ($A_{max} = 30$ in the above example).
- Let B_{min} and B_{max} be defined respectively.

By following the example given above we get that the necessary condition for the transition point to occur is:

$$or \begin{cases} A_{min} > B_{max} \\ B_{min} > A_{max} \end{cases}$$

because this will ensure the events will be serialized in the correct order. As the simulation evolves, A and B are constantly changing (the reason for this was explained in previous round-robin chapters). The transition point will occur only when the above condition is satisfied.

So far, we've established the reason why *J*'s threads manage to group together, but this is only half of the work. We now need to understand what's keeping them together i.e. what makes the scheduler continue to schedule *J*'s threads as a group until the computation ends. There are two reasons:

1. One of the characteristics of a fine grain job, is that the computation time done by its various threads is more or less the same at any given time instance. This is a direct product of doing a lot of (barrier) synchronization. In *J*'s case for example, it is very unlikely for t_1 to have computed for 1000 cycles if t_2 has only computed for 950 cycles. In general, the spin maximal duration serves as an approximation of the upper bound on the difference between the `counter` fields of each pair of *J*'s threads. This usually ensures us that after grouping, *J*'s threads will exhaust their quantum and be preempted together, i.e. will not split up again to two groups A and B in account of a quantum which is exhausted only for some, while the others continue to compute.

2. Finally, this is where the assumption that the total number of threads is a multiplication of 16 comes in. It has no mystic meaning, it is simply the smallest divisor of CPU# (=32) which is not smaller than $|J|$ (=11) and thus able to contain it: After *J*'s threads grouped and exhausted their quantum together, blocking and preemption events ceased to occur. This, along with the fact that threads have a tendency to run on their previous processor (because of `PROC_CHANGE_PENALTY`), allows the CPUs to simply be partitioned between unchanging groups of 16 threads: whenever a 16 sized group of threads (possibly containing *J*) has exhausted its quantum, it is simply replaced by another group of 16 threads.

We now have complete understanding of the behavior of the simulations displayed in figure 7.2:

- The peaks are caused due to the existence of a transition point.
- For loads that aren't a multiplication of 16, SSR is higher than similar round-robin simulations, revolving around 50%. The reason it's higher, is that *J*'s thread indeed manage from time to time to group and compute together beating the alt synchronization pattern for a while (at least until the quantum is exhausted). The reason the SSR is not as high as of simulations with load which is a multiplication of 16, is that the SMP cannot maintain constant groups of threads to be scheduled together, and soon enough *J* splits again to two alt synchronizing groups.

7.6 Bigger Maximal Spin Duration

The most important conclusion from the previous section is that a spin interval with a CS duration is not enough (or more accurately, it is the biggest spin interval which is not sufficient). We may safely assume that if the duration of the spin interval was "a little" longer, the transition point would have happened much

sooner (or more frequently for loads that aren't a multiple of a CPU#-divisor which is bigger than or equal to $|J|$), thus greatly improving the SSR achieved. Let *CS+* denote this spin interval duration. In real world systems, "a little" means enough time such that all the awakening threads would succeed to synchronize (i.e. overcome contention problems etc). However, within our simulator it is enough to define CS+ as simply: CS+1.

Figure 7.5 displays the transition point of a simulation similar to the one conducted in the previous section with the sole difference of using CS+ instead of CS as the maximal spin duration.

The following is a comparison between figure 7.3 (CS) and figure 7.5 (CS+):

Load	Transition Point Happened Before		Completion Time	
	CS	CS+	CS	CS+
$80 = 16 \times 5$	700	9000	14,217	20,877
$96 = 16 \times 6$	22,000	3000	33,903	19,707
$112 = 16 \times 7$	52,000	11,000	56,446	26,656

There are two "disturbing" aspects in these results:

1. It is clear that as expected the transition points happened much sooner for loads 96 and 112 when CS+ was used as a maximal spin duration. This resulted in a speedup of approximately factor of 2. However, for load 80, the transition point happened much later and as a result there has been a considerable slowdown (with factor of approximately 1.5). We will address this problem shortly.

2. Even though the results of the CS+ simulations with load 96/112 showed improvement with respect to the their CS counterparts, it still takes these simulations a considerable amount of time to reach the transition point. The question that follows is why.

The explanation for the second point raised above is simply that the maximal spin duration still doesn't suffice. Indeed, threads from A (as defined above) are spinning enough time to allow any blocked thread — t — with high enough priority to join them. But what if t is only about to block, i.e. it has just yielded its CPU after an unsuccessful spin and therefore `schedule` was invoked (and is just in the beginning of the process of choosing the next CPU to run). Since `schedule` cannot be "stopped" while it's executing, a new thread will be chosen by it to run on t.`processor` and t will have to wait until `reschedule_idle` "reassigns" it its processor. Note that t is considered to be running until line 26 in algorithm 4 (page 61) is executed.

The following is an illustration of a scenario in which CS+ maximal spinning time is not enough. It is taken from the CS+ simulation associated with load=80:

Time=858 t_8^{22} fails on b_{20} and a context-switch-out event is pushed on its behalf. The simulator-function which is the equivalent of `schedule`, decides which will be the next thread to run on CPU_{22}. Let this thread be denoted as u. In order to faithfully simulate the original `schedule`, this decision will take effect only after CS cycles will pass.

Time=860 t_4^1 is the last of *J*'s threads to reach b_{20} and therefore b_{20} is complete.

Time=861 The context-switch-out event of t_8^{22} has expired. Even though it is now known that t_8^{22} may (and will) continue to compute because the event it has been waiting for has occurred, nothing may be done at this stage because the original `schedule` cannot be interrupted while it is executing. A context-switch-in event on behalf of u is therefore pushed. In addition, t_4^1 finishes its current computation phase.

Time=862 t_4^1 reaches b_{21} and starts to spin. It will continue to do so for not longer than CS+ cycles i.e. not later than time=869.

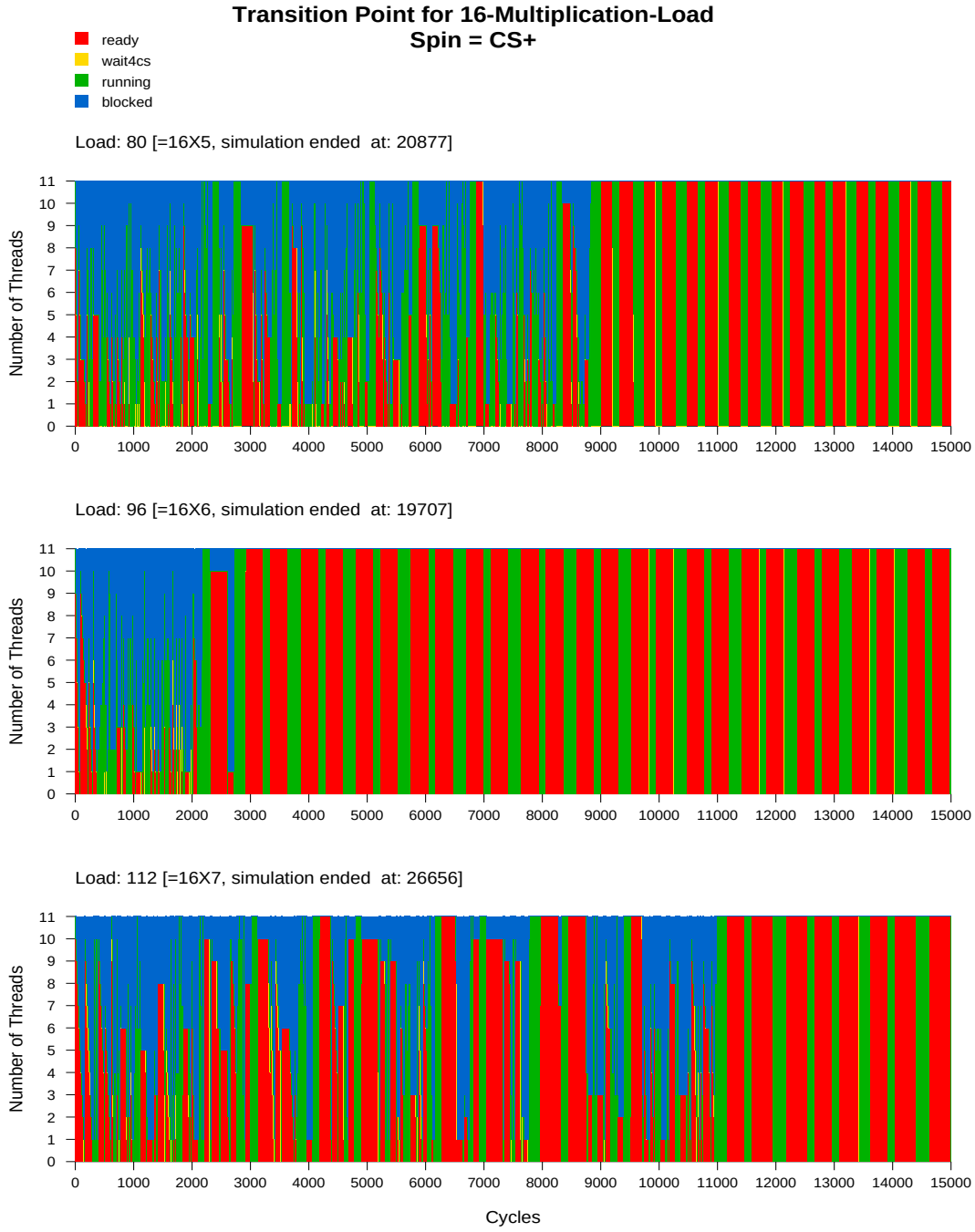

Figure 7.5: This figure is similar to figure 7.3 with the difference that the jobs displayed here used CS+ maximal spin duration. We can see that for loads 96 and 112 the transition point occurred earlier than for the CS simulations. However, for the 80 sized simulations the result is reversed.

Time=864 The context-switch-in event of u has expired. This is equivalent to the point where `schedule` is after switching to u's context (the completion of line 26 in algorithm 4).

Time=865 However, since t_8^{22} didn't block after all, and is a runnable task, `reschedule_idle`(t_8^{22}) is now invoked (line 27 in algorithm 4). t_8^{22} has the highest goodness value on CPU_{22} and it is high enough in order to preempt u. Consequently, u.`need_resched` flag is set. As the "kernel" is about to return to (u's) user mode, it checks the u.`need_resched` flag (which is set) and therefore invokes `schedule`, which in turn chooses t_8^{22} to be the next thread to run. As explained before, this `schedule` decision will take effect only after CS cycles will pass. Consequently, a context-switch-out event is pushed on behalf of u.

Time=868 The context-switch-out event of u has expired. A context-switch-in event has been pushed on behalf of t_8^{22}.

Time=869 t_4^1 has unsuccessfully finished spinning on b_{21} and yields its CPU.

Time=871 The context-switch-in event of t_8^{22} has expired. It begins to compute.

Time=872 t_8^{22} finishes the computation phase.

Time=873 t_8^{22} reaches b_{21}, but by now it is too late for t_4^1 which has already yielded its CPU at time=869.

We therefore conclude that in order for a fine grain job to have a maximal chance to reach the transition point as fast as possible, its maximal spin duration should actually be in the order of *2CS+*. This will supply other threads from the job, that yielded their CPUs in close proximity to the barrier completion, with enough time to join their waiting counterparts.

The explanation for the first point that was raised above, regarding the fact that the CS simulation (with load=80) achieved better results than of the CS+ simulation, is also simple: sheer luck. The division of J to two alt synchronizing groups A and B (as defined in the previous section) was such that the condition stated in subsection 7.5.5 was satisfied immediately when the simulation began. In order to prove this was just a lucky event, we've conducted the same simulation (load=80, μ=1%) using maximal spin durations of CS/CS+/2CS+ with 100 different randomly chosen seeds. The result is displayed in figure 7.6.

The figure shows that generally, simulations that used 2CS+ as the maximal spin duration, reached the transition point more quickly than CS+ simulations, which in turn reached the transition point much more quickly than the CS simulations. The average time it took the various simulations to reach the transition point, as well as their average SSR and completion time is as follows:

maximal spin duration	transition point	SSR	completion time
CS	17,016	81.1	26,194
CS+	4,986	96.6	17,644
2CS+	2,282	99.2	17,013

These results coincide with our findings so far. Note that for these kinds of loads which are a multiplication of a CPU#-divisor bigger than $|J|$, when making the computation duration go to infinity, the difference between the averages achieved by simulations with various maximal spin durations, is expected to go to zero. This is true because once the transition point is reached, all the barriers are successful all the time and the maximal spin duration is never used to its full. In this case, the difference between the various simulations is found only in their early stages, which have lesser relative weight the longer we make the computation. However, different maximal spin duration will play a much more important role for loads different than the above.

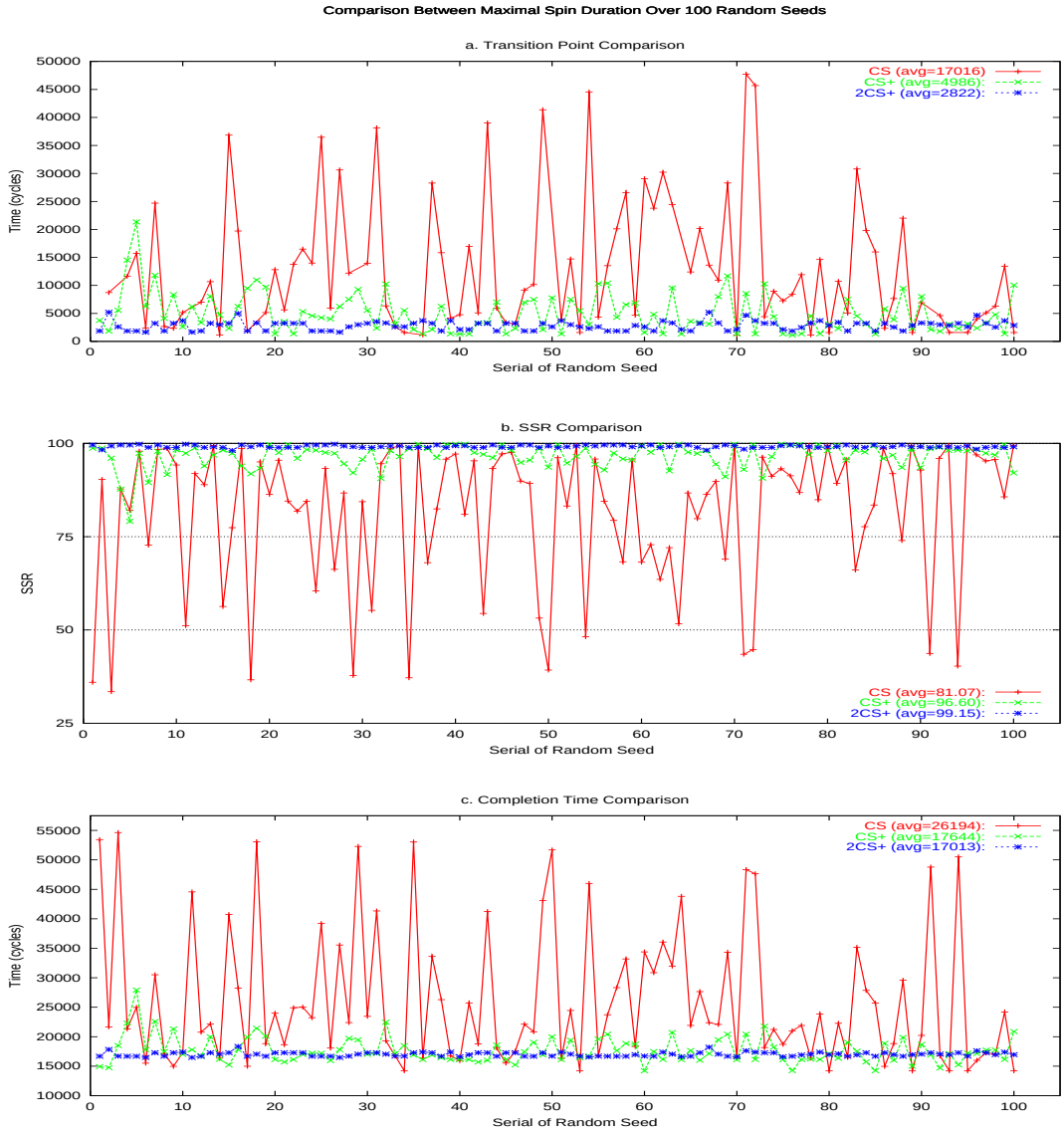

Figure 7.6: Comparison between the fine-grain 80-sized simulations that use CS/CS+/2CS+ as their maximal spin duration. Each such simulation was executed 100 times using 100 different random seeds. The X-axis displays the serial of the random seeds. Figure (**a**) displays the transition point comparison. The seed with serial 100 is the one used in the simulations we have analyzed in figures 7.3 and 7.5. For this seed, we can see that the transition point of the CS simulation indeed occurred much before than of the CS+ simulation. Evidently, this is an exception. Figure (**b**) displays the SSR comparison. The sooner the transition point occurs, the higher the SSR is. This is the reason why the highest SSR is achieved by the 2CS+ simulations and the lowest SSR is achieved by the CS simulations. Figure (**c**) displays the completion time comparison. It turns out that the higher the SSR is the faster the simulations end, even at the cost of longer spin durations.

7.7 Comparison Between Wakeup Schemes and Spin Durations

In the previous sections, we have analyzed the behavior and implications of a single synchronizing job executing within a non-synchronizing environment on a system that uses the Linux SCHED_OTHER scheduling algorithm. Our focus was on loads that generated SSR peaks (namely a multiple of a CPU#-divisor which is bigger than or equal to $|J|$), but we were able to generalize our understandings and apply them to other loads. We have concluded that in order for a barrier algorithm to be successful, a CS maximal spin duration is not sufficient and that better SSR will probably be achieved when prolonging it to CS+ or better yet to 2CS+. This conclusion was demonstrated for peak-loads. However, it was not demonstrated for the other loads. In addition, we would now like to affirm the implied connection between better SSR and faster executions within all possible loads. We would also like to compare between the various wakeup schemes and examine what are their effects. This section will present a comparison between simulations that are similar to the one we discussed up till now with:

- three maximal spin duration of: CS/CS+/2CS+, and
- three wakeup schemes: SMART/AIP/SILLY

7.7.1 SSR Comparison

The SSR comparison of fine and medium grain jobs is displayed in figures 7.7 and 7.8 respectively. Here is the average SSR and its absolute deviation (defined to be: $\frac{1}{n}\sum_{i=1}^{n}|measurment_i - \mu|$) across all the different loads that are bigger than CPU#:

wakeup scheme	fine grain			medium grain		
	CS	CS+	2CS+	CS	CS+	2CS+
SMART	$54_{\pm 10.6}$	$83.1_{\pm 7}$	$94.2_{\pm 2.6}$	$28.1_{\pm 7.8}$	$41.6_{\pm 13.5}$	$76.7_{\pm 8.8}$
AIP	$48.9_{\pm 10.3}$	$79_{\pm 8.1}$	$93.9_{\pm 3}$	$24.7_{\pm 6.8}$	$37.5_{\pm 12.5}$	$75.2_{\pm 9.2}$
SILLY	$38.5_{\pm 8.9}$	$69.9_{\pm 10.4}$	$91.9_{\pm 4.5}$	$23_{\pm 5.8}$	$34.7_{\pm 11.5}$	$73.8_{\pm 9.5}$

Here is the analysis of the above results:

- Regardless of the wakeup scheme used, we can see a dramatic improvement in the SSR as the maximal spin duration is enlarged: The difference between the SSR achieved by CS and 2CS+ simulations is 40-50% in favor of 2CS+. The difference between CS+ and 2CS+ simulations is 10-20% for fine grain jobs and 30-40% for medium grain jobs in favor of 2CS+.
- Notice how SSR achieved by fine grain jobs in the 2CS+ simulations is almost always above 90% ! The meaning of this is that by fine tuning the maximal spin duration, we have managed to transform all loads to "peak loads" i.e. if previously, only jobs that executed within a specific load enjoyed a SSR close to 100% (load which is a multiplication of a CPU#-divisor not smaller than $|J|$), now J enjoys similar SSR regardless of the load.
- Also notice that while fine grain CS+ simulations manage to achieve a decent SSR when AIP and SMART are used ($\approx 80\%$), for medium grain jobs, the only practical spin duration is 2CS+ (otherwise the SSR is below 50%). The reason for this is implied from our analysis above: A spinning thread must wait for an awakened thread enough time for it to (**a**) preempt another thread (usually a CS duration but occasionally up to 2CS duration) and (**b**) to finish the next computation phase before reaching the next barrier (an expected μ duration). For fine grain jobs (b) is much smaller than (a) thus allowing CS+ to achieve reasonable SSR. However, for medium grain jobs, the relative weight of the duration of (b) increases. This fact is partially embodied in 2CS+ but not at all in CS+. As a result, a CS+ maximal spin duration is usually not enough causing most spins to fail while 2CS+ is big enough to allow a considerable amount of spins to succeed.

- When comparing the various wakeup schemes, we can see that the difference in the average SSR (achieved by simulations using the same maximal spin duration) is quite small: For medium grain jobs, the differences are in the order of 2-7% (7% being the difference between SMART and SILLY when CS+ is used). For fine grain jobs the differences are 2-15%. In any case, when 2CS+ is used, the difference between the various schemes is only 2-3%.
  Although the SILLY scheme introduces a bug (CPUs might "get lost") and the idle pitfall (a group of threads might be assigned to the same "oldest" idle CPU), it seems that wakeup schemes do not play a crucial role in the context of this work. The main reasons for this are:

  1. The probability of idle CPUs is nonnegligible only in the intermediate load, when the number of non-synchronizing threads is smaller than CPU# while the total threads number is bigger. For this load, the combination of the following two factors "saves" SILLY from assigning all the awakened threads to the same (oldest) idle CPU:
     (a) each thread has a strong association with its previous CPU (because of PROC_CHANGE_PENALTY and algorithm 2 lines 4-5), and
     (b) $|ECS_J| \rightarrow |J|$ towards the starting of a new epoch (when the threads actually have a chance to group together) i.e. the previous CPUs of J's threads are pairwise disjoint.
  2. For bigger loads, idle CPUs simply do not exist and the problem is avoided altogether.

7.7.2 Maximal Spin Duration Speedup Comparison

A comparison between the conducted simulations according to their maximal spin duration, is displayed in figures 7.9 and 7.10 for fine and medium grain jobs respectively. In order to compare a pair of simulation $\langle X, Y \rangle$ according to their maximal spin duration (where X is considered to be the simulation that uses a "better" maximal spin duration in the sense that 2CS+ is "better" than CS) we've used the formula:

$$speedup \;=\; \frac{completionTime(Y) - completionTime(X)}{completionTime(Y)} \times 100$$

i.e. we display the speedup of X with respect to Y in percentage. The title "2CS+ vs CS" implies that the left term (2CS+) is associated with X and the right term (CS) with Y. Note that a positive speedup means that X is faster than Y and a negative speedup means the opposite. Also note that we use the term "speedup" in a non conventional way: in this work this term means "relative improvement". For example, 60% means: a reduction of 60% in the time, which is equivalent to a "conventional speedup" of 2.5 (i.e. 2.5 times faster).

Here is the average speedup and its absolute deviation across all the different loads that are bigger than CPU#:

maximal spin duration	fine grain			medium grain		
	SMART	AIP	SILLY	SMART	AIP	SILLY
2CS+ vs CS	$45.9_{\pm 7.5}$	$50.5_{\pm 7.7}$	$53.6_{\pm 9.6}$	$20.3_{\pm 7.5}$	$22.2_{\pm 8.4}$	$21.7_{\pm 10.3}$
CS+ vs CS	$36.8_{\pm 7.8}$	$35.1_{\pm 10.9}$	$32.9_{\pm 10.3}$	$5.4_{\pm 5.4}$	$5_{\pm 5.1}$	$4.3_{\pm 5.1}$
2CS+ vs CS+	$13.4_{\pm 11.1}$	$21.2_{\pm 13.4}$	$28.8_{\pm 15.8}$	$15.6_{\pm 6.7}$	$17.9_{\pm 7.6}$	$18_{\pm 9.2}$

The results displayed above coincide with our findings regarding the SSR:

- 2CS+ simulations are considerably faster than CS simulations: $\sim$50% faster for fine grain jobs and $\sim$20% for medium grain jobs.
- Fine grain jobs within CS+ simulations are $\sim$35% faster than within CS simulations (but for medium grain jobs the speedup is only $\sim$5%).

Figure 7.7: This figure compares the SSR achieved by fine grain jobs when different wakeup schemes and maximal spin durations are used. Generally, bigger spin duration resulted in better SSR. For 2CS+ most measurements are in 80-100% ($avg \approx 96\%$), for CS+ in 50-100% ($avg \approx 77\%$) and for CS in 25-75% ($avg \approx 47\%$). This means that when choosing a correct spin duration, a fine grain synchronizing job within a non-synchronizing environment may successfully compute regardless of the load. When examining the difference between the various wakeup schemes, we see that the difference is fairly small, and shrinks as we enlarge the spin duration (in 2CS+ the the curves are almost indistinguishable).

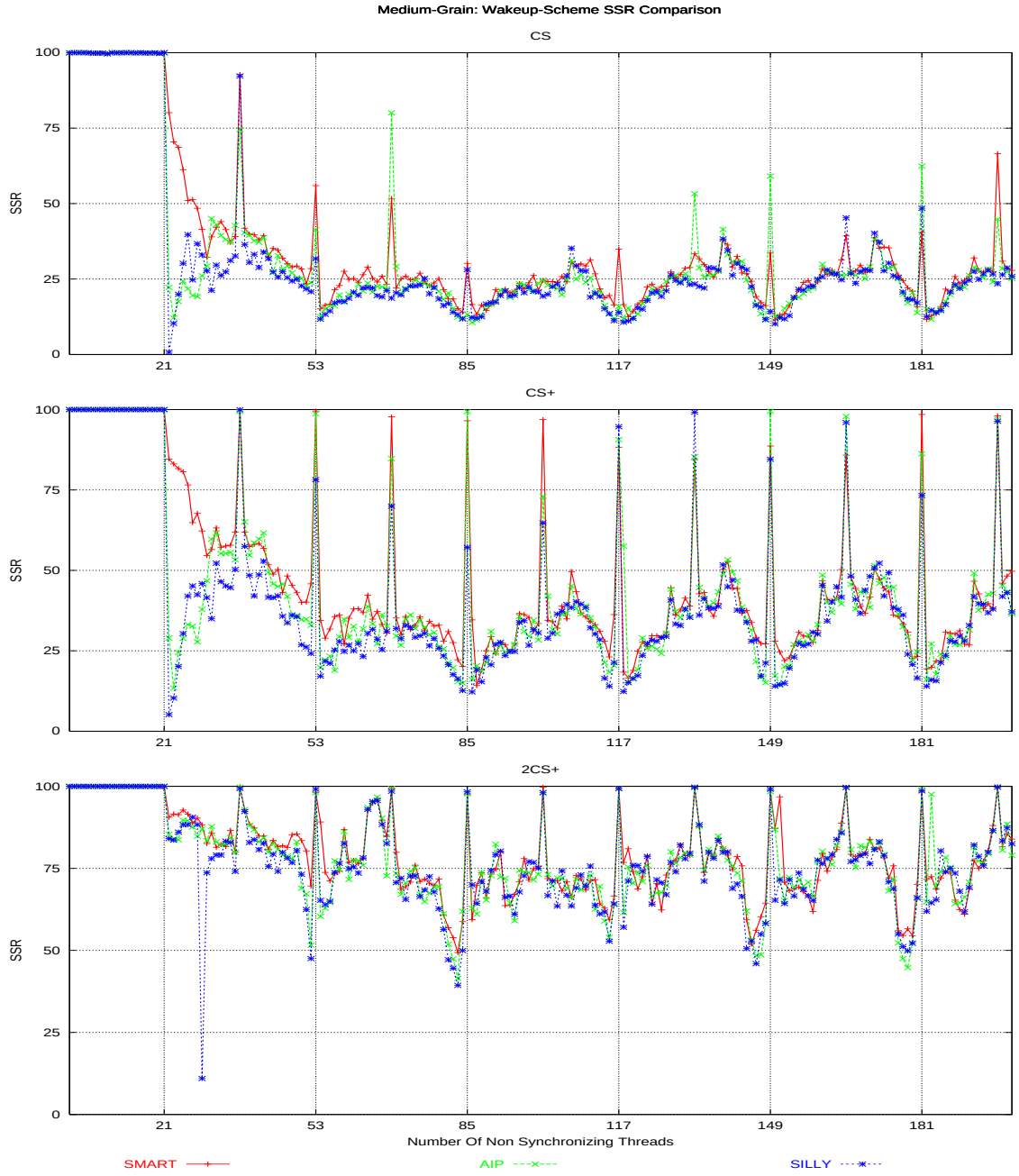

Figure 7.8: This figure compares the SSR achieved by medium grain jobs when different wakeup schemes and maximal spin durations are used. The results are similar to those obtained for fine grain jobs (figure 7.7) but here the SSR is much lower. The only practical spin duration for these kinds of jobs is 2CS+ (the others are below 50%).

Recall that the connection between better SSR and higher performance isn't a given since SSR in not a prefect metric: Indeed, when the SSR is low, we may safely say that it was preferable not to spin at all. However, the immediate drawback of achieving higher SSR by prolonging the spin duration is that synchronizing threads spend more time while spinning which theoretically might prolong the total computation time. The results above show that higher SSR indeed led to shorter execution time.

7.7.3 Wakeup Schemes Speedup Comparison

The wakeup schemes speedup comparison of fine and medium grain jobs is displayed in figures 7.11 and 7.12 respectively. The comparison is done in the same manner it was done in the previous subsection (i.e. each pair of schemes is compared and the speedup percentage is displayed). Here is the average speedup and its absolute deviation across all the different loads that are bigger than CPU#:

wakeup scheme	fine grain			medium grain		
	CS	CS+	2CS+	CS	CS+	2CS+
SMART vs SILLY	$22.6_{\pm 10.6}$	$25.3_{\pm 15.3}$	$8.4_{\pm 11}$	$7.1_{\pm 4.4}$	$7.9_{\pm 5.8}$	$4.1_{\pm 5.3}$
AIP vs SILLY	$13.7_{\pm 10.1}$	$15.3_{\pm 17.1}$	$6.1_{\pm 9.8}$	$2.3_{\pm 2.8}$	$3_{\pm 3.4}$	$1.7_{\pm 4.2}$
SMART vs AIP	$9.8_{\pm 10}$	$10.1_{\pm 12.7}$	$1.3_{\pm 9.2}$	$4.8_{\pm 3.6}$	$5_{\pm 4.7}$	$2.3_{\pm 4.3}$

The results here, also coincide with our analysis from section 7.7.1 that suggested only a minor improvement when 2CS+ maximal spin duration is used (SMART is $\sim 8\%$ faster than SILLY for fine grain jobs and less for medium grain jobs; the difference between AIP and SILLY is even smaller). However for smaller spin duration done by fine grain jobs, the improvement is more meaningful ($\sim 25\%$).

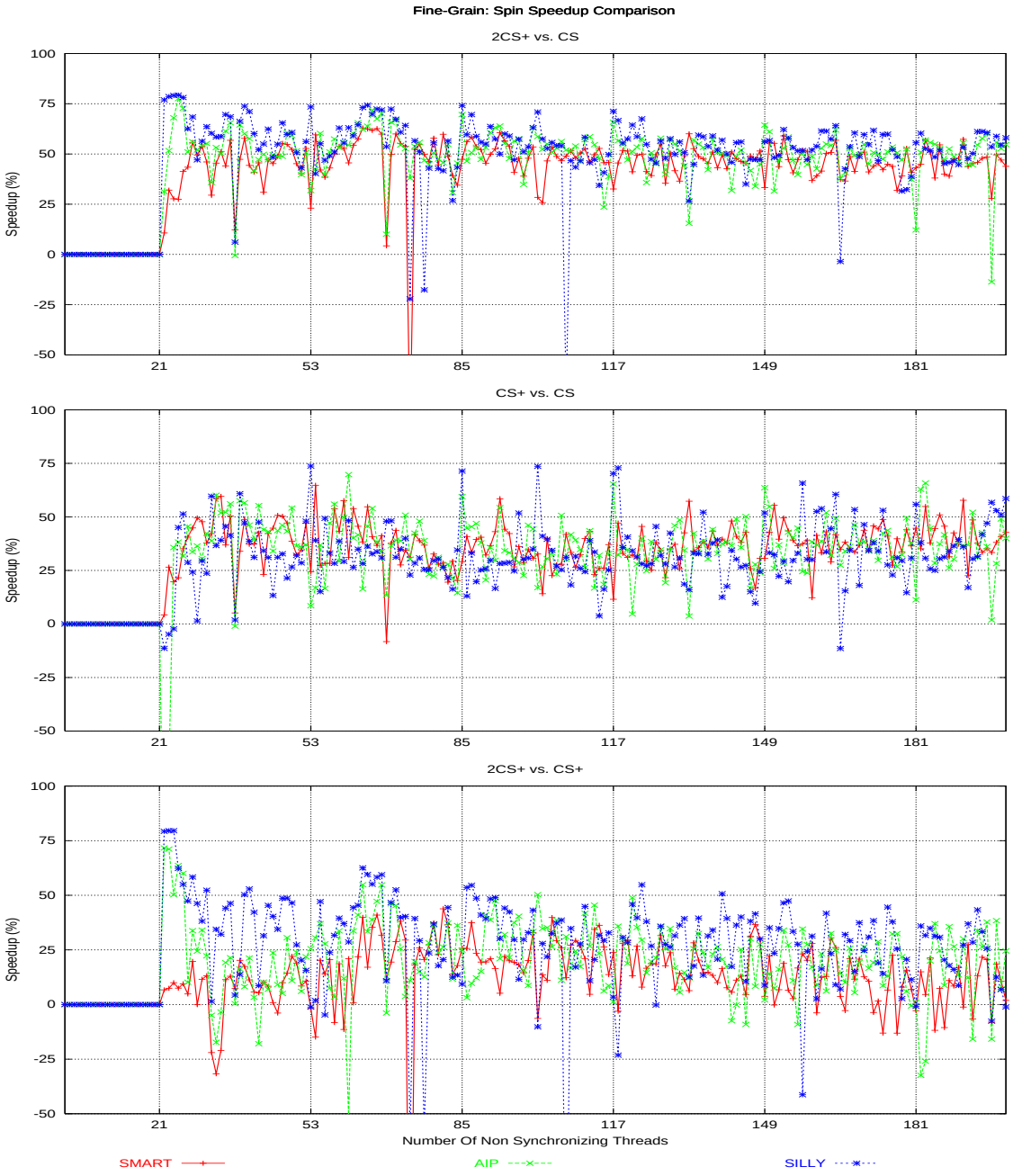

Figure 7.9: This figure displays the speedup achieved by a fine grain job using different maximal spin duration (each pair of spin duration was compared). The above results coincide with the SSR findings presented earlier, i.e. there's a strong association between the SSR and the overall performance: 2CS+ simulations are 25-75% faster than CS simulations ($avg \approx 50\%$); CS+ simulations are 25-50% faster than CS simulations ($avg \approx 35\%$).

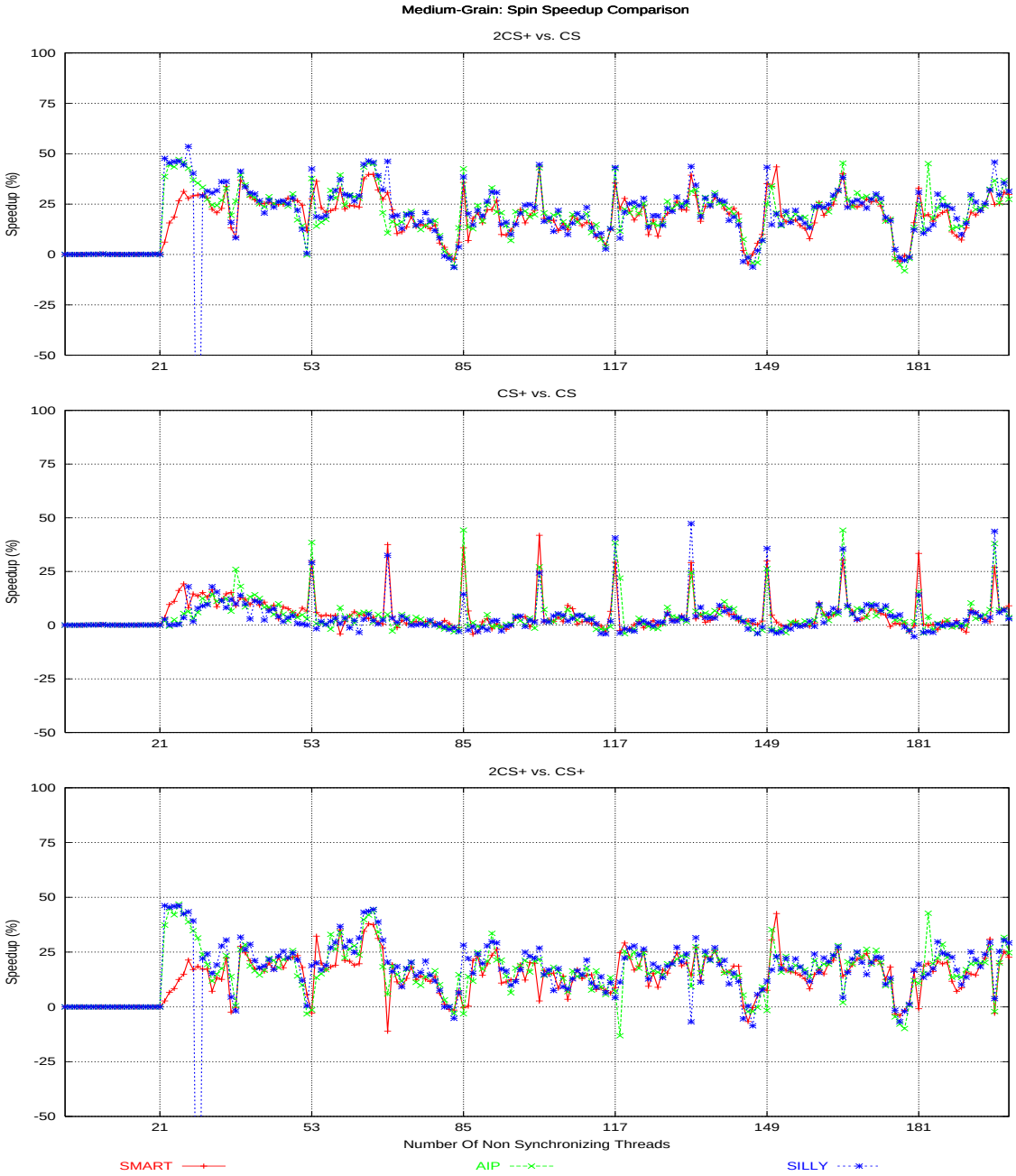

Figure 7.10: This figure displays the speedup achieved by a medium grain job using different maximal spin duration. The comparison is done similarly to the way it was done in figure 7.9. It is evident that a 2CS+ maximal spin duration is more suitable for medium grain jobs as there almost no difference between CS+ and CS simulation. 2CS+ simulation are $\sim$20% faster than the other simulations.

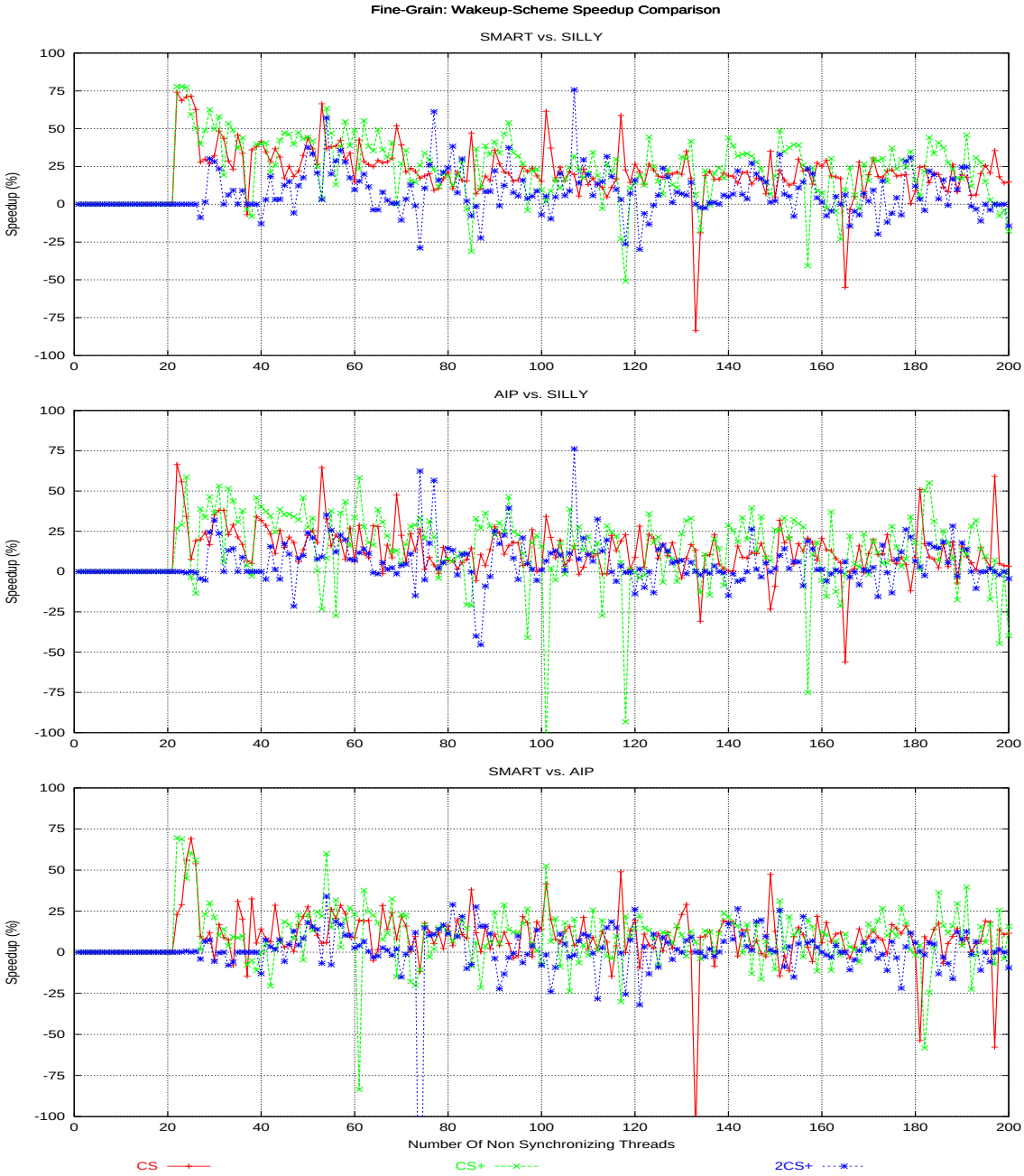

Figure 7.11: Comparison between the various wakeup spins for fine grain jobs. AIP and SMART are about 20-25% faster than SILLY on average when using CS/CS+ maximal spin duration. This speedup is reduced to an average of not more than 8% when 2CS+ is used.

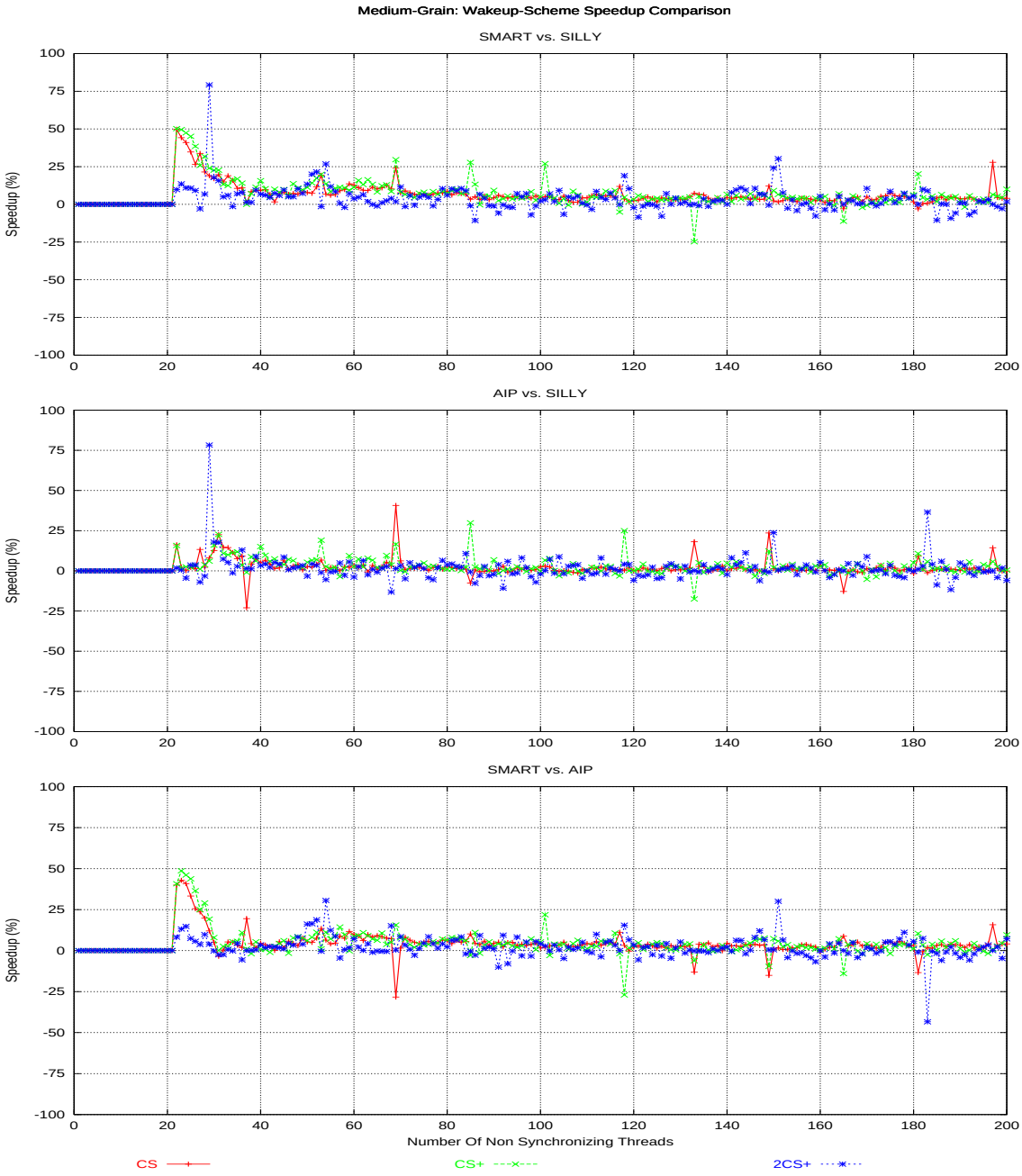

Figure 7.12: Comparison between the various wakeup spins for medium grain jobs. Curves are almost indistinguishable and are in the neighborhood of 0. The biggest average speedup is $\sim 8\%$ (CS+, SMART vs. SILLY).

Chapter 8

Homogeneous Collection of Synchronizing Jobs Under the Linux Scheduler

8.1 Introduction

In this chapter we will conduct a series of simulations identical to those conducted throughout chapter 4 (the associated round-robin chapter) with the sole difference of changing the scheduling algorithm to Linux SCHED_OTHER. In chapter 7 the Linux scheduler managed to allow all the threads of a single fine grain job to execute simultaneously even with extremely high loads, thus considerably shortening the elapsed execution time of this job. We believe that the job collections presented in this chapter, in which an increasing number of synchronizing jobs will compete on the system resources, will prevent this extraordinary success from re-occurring and establish the load as being a crucial parameter.

8.2 Description and Results

The simulations series is composed of an increasingly growing collection of jobs with an identical profile. The parameters shared by all the simulations we will conduct in this section are:

p	q	in	out	nosync	barrier	σ	randord
32	100	3%	3%	0	2000	90/15%	1

The simulation will differ in:

Size which may be one of $SIZ = \{2,\ 3,\ 4,\ 5,\ 10,\ 11,\ 15,\ 16,\ 22,\ 25,\ 32\}$

Maximal spin duration which may be one of $SPN = \{CS,\ CS+,\ 2CS+\}$, and

Wakeup scheme which may be one of $WSCM = \{SMART,\ AIP,\ SILLY\}$

Grain which may be one of $GRN = \{fine(1\%),\ medium(10\%),\ coarse(100\%)\}$

The total number of simulation sequences is therefore:

$$|SIZ| \times |SPN| \times |WSCM| \times |GRN| \ = \ 11 \times 3 \times 3 \times 3 \ = \ 297$$

Each simulation-sequence begins with a simulation that contains a single job, the next simulation in the sequence adds another job and thus composed from two jobs with an identical profile, and so on until

the total number of threads in the job collection exceeds 320 ($= CPU\# \times 10$). Figure 8.1 displays the results of the all the simulation sequences that used 2CS+ as a maximal spin duration and AIP as a wakeup scheme.

We have chosen to display only the results of AIP 2CS+ simulations because:

1. this seems to be the most practical choice: SILLY would probably be fixed (at least a minor fix like the one presented by AIP); SMART is probably too expensive; 2CS+ seems to be superior to CS and CS+,
2. we would need eight more pages to present the other scheme+spin pairs (because $|SPN| \times |WSCM| = 9$), and
3. most importantly, after comparing these pages, we can report here that they are quite similar and the differences between them are best presented here in the form of averages and deviations.

When examining the resulting graph, we notice:

- The difference between the results of the simulations conducted here and of those that were conducted in the previous chapter within a non-synchronizing environment: Our prediction that a collection of competing jobs will not allow a high SSR regardless of the load, proved to be true i.e. from a certain point, spinning doesn't pay off and is better avoided.
- For all the simulation sequences displayed, there exists an intermediate load range in which the SSR is bigger than 50%. Job collections composed from smaller jobs manage to achieve a bigger thread surplus (this will be further elaborated in the next section).
- The difference between the results of the fine grain simulations conducted here and of those that were conducted in the round-robin homogeneous chapter: High loads cause the SSR curves to go to zero rather than converging to some value in the south side of 50%. The reason for this is obvious: when using the Linux SCHED_OTHER scheduler, the order in the ready queue plays an insignificant role in choosing the next thread to run (simply a tie breaker i.e. the next thread to run form within a pair of threads with equal goodness is the one that is closest to the head of the ready queue). This is contrary to the round robin algorithm in which the fact that all the awakened threads are moved together to the ready queue and are contiguously ordered there, also ensures that they will be scheduled in close proximity to one another, which ensures in turn that the SSR will be in the neighborhood of 50% (as each group of threads fails and succeeds to synchronize alternately) .

8.3 Threads Surplus

This section will discuss the threads surplus for which $SSR > 50\%$ was achieved within the various simulations and its dependency on jobs size, maximal spin duration and wakeup scheme. Note that the surplus analysis is an important perspective which may be viewed as orthogonal to the speedup considerations (speedup considerations say: "when running simulation X, it ends faster when we use 2CS+ rather than CS+"; surplus consideration say: "spinning is worth while for load X when using 2CS+ but isn't worth while for the same load when using CS+"). Figures 8.2 and 8.3 display the threads surplus achieved by fine and medium grain simulations respectively. These figures differentiate between jobs sizes. The following however is the average number of threads surplus across all sizes:

wakeup scheme	fine grain			medium grain		
	CS	CS+	2CS+	CS	CS+	2CS+
SILLY	10.4	23.8	41.6	6.9	10.4	24.7
AIP	16.7	31.1	45.5	12	14.2	29.9
SMART	16.5	31.1	48.9	10.6	13.8	29.9

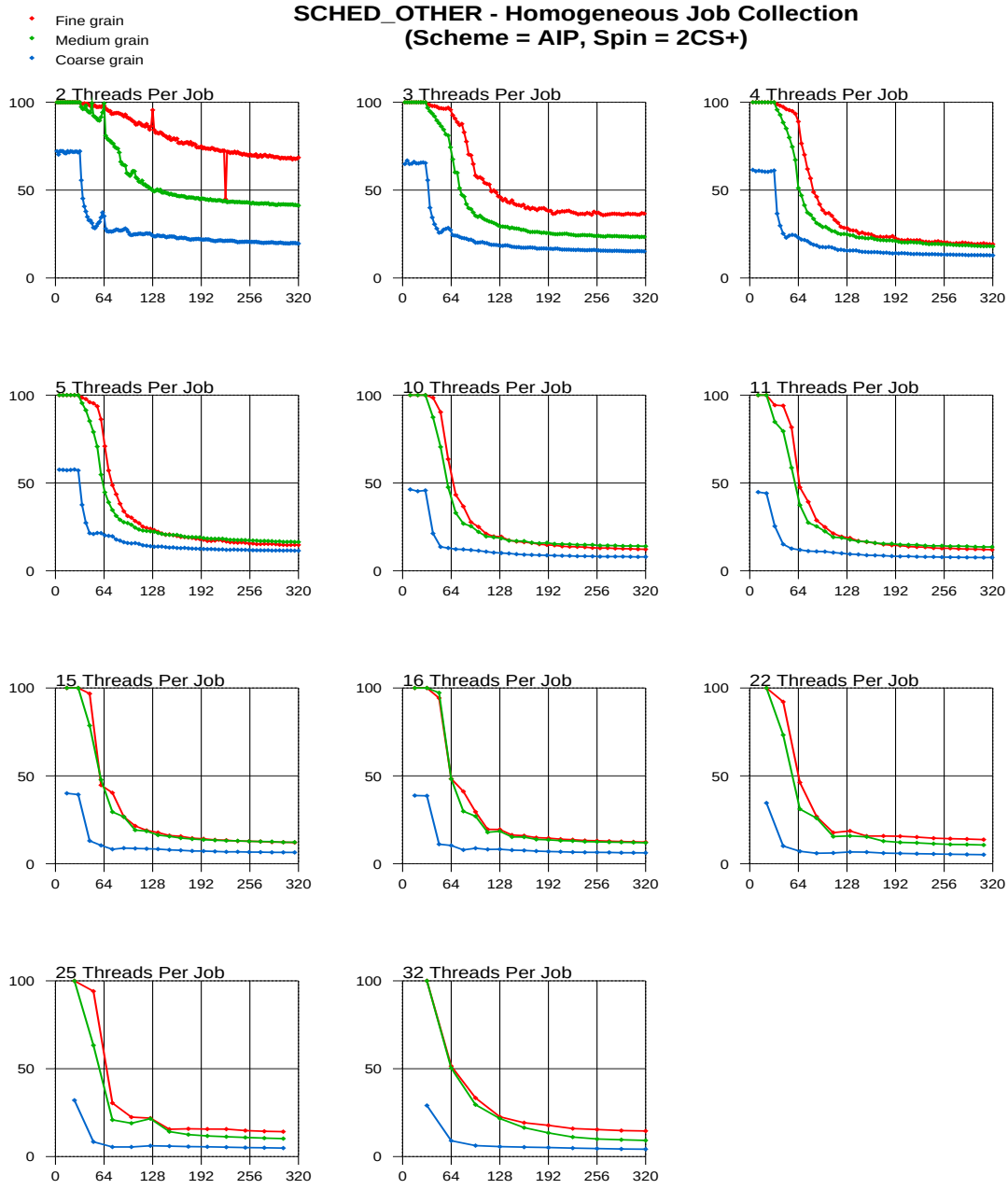

Figure 8.1: This figure displays the results of all the AIP 2CS+ simulations. The Y-axis is associated with the SSR. The X-axis is associated with the number of threads (not jobs) that participated in the simulation, and it ranges up to $CPU\# \times 10$. Curves associated with fine and medium grain jobs manage to sustain $SSR > 50\%$ in the intermediate load but eventually (contrary to the results displayed in the previous chapter) drop under the 50% threshold and converge to 0 as the load increases. Collections composed from bigger jobs achieve smaller thread surplus in which $SSR > 50\%$.

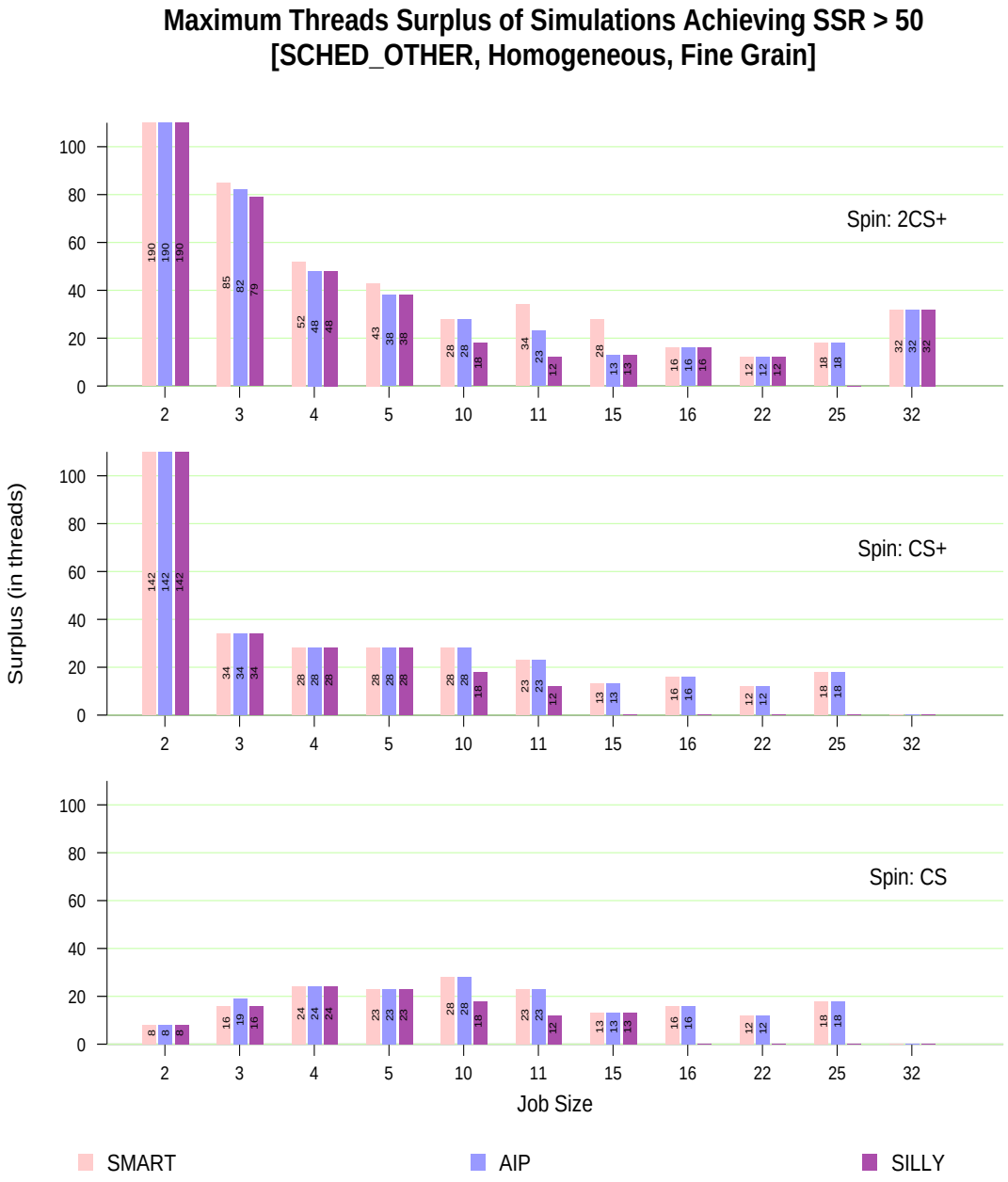

Figure 8.2: This figure displays the maximal threads surplus of fine grain simulations for which $SSR > 50\%$, as a function of the job sizes, the maximal spin duration, and the wakeup scheme used. Longer spin durations achieve bigger surplus. SILLY bars tend to disappear for bigger job sizes. With the exception of CS, collections composed from smaller jobs tend to produce bigger surplus.

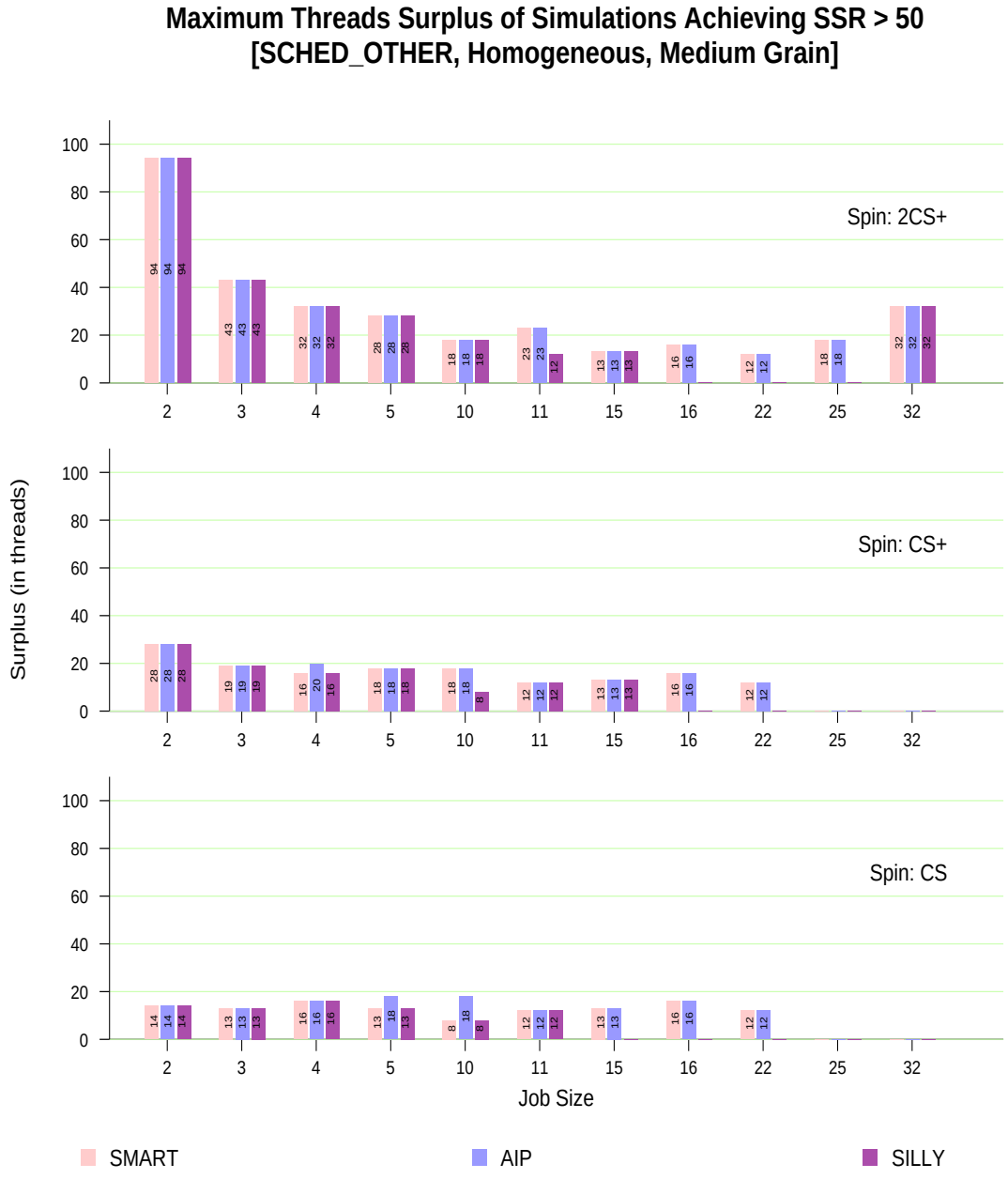

Figure 8.3: This figure is similar to figure 8.2 with the difference that the threads surplus displayed here is associated with medium grain jobs. The same observations that were made for the fine grain figure, also hold for this figure.

Our conclusions from the surplus figures and the average table are as follows:

- **Difference between sizes**: As mentioned earlier, the difference between the results of simulations with different job sizes was that smaller sizes seemed to result in a longer intermediate range with $SSR > 50\%$. This is evident in both figures associated with fine and medium grain jobs. However, this observation seems slightly incorrect when examining the surplus of jobs with $size \geq \frac{CPU\#}{2}$. The reason is that for these sizes, a job collection is "able" to contain only a single surplus job while sustaining the SSR above the 50% threshold. For example, in the 22 sized job collection the surplus is 12 threads because $22 \times 2 - 32 = 12$ (22×2 is the minimal collection of 22-sized-jobs that is bigger than the $CPU\#$). The reason that collections composed from smaller jobs manage to sustain $SSR > 50\%$ on bigger loads is prosaic: when the last thread — t_1 — of some job completes a barrier, than all the threads that were awakened as a result, must preempt currently executing threads in order to join t_1. In a competing environment, the probability that this will happen (i.e. that all the awakened threads will be able to preempt currently executing threads) gets smaller as the number of the awakened threads gets bigger (for 2-sized jobs, it's enough that 1 awakening thread will be able to preempt; for 3-sized jobs, 2 threads must be able to do that and so on).

- **Maximal spin comparison**: In general, when we examine the average surplus, we see that our conclusions regarding the maximal spin durations are also applicable to the threads surplus: For fine grain SMART/AIP simulations, 2CS+ achieve a surplus of $\sim 2.5 \times CPU\#$, while CS+ achieve a surplus of $\sim CPU\#$, and CS achieve only a $\sim \frac{CPU\#}{2}$surplus. We conclude that for fine/medium grain jobs, in addition to the fact that bigger spin durations result in a better SSR (and therefore better speedup) they also greatly effect the maximal load in which spinning is worth while.

- **Wakeup scheme comparison**: When comparing the averages of the various wakeup schemes, we see the difference between them is not as dramatic as when comparing spin durations. However, the fine grain jobs average specifies that SMART was able to sustain $SSR > 50\%$ for $\sim$7 ($\approx 20\%$ of CPU#) more surplus threads than SILLY. This difference in nonnegligible. AIP performs almost as well as SMART (and sometimes even better) while the surplus achieved by SILLY is always equal to or less than the surplus achieved by the other schemes.

- **Jobs of size 2**: Note that for jobs with: $2 < size < \frac{CPU\#}{2}$, about 50% of the associated bar clusters display a difference between the surplus achieved by the various wakeup schemes. Moreover, for each size in this range, there exists a $\langle grain, spin \rangle$ pair for which the associated wakeup-scheme bars present an unequal surplus. This is not the case for size=2, for which the surplus achieved is always equal among the different schemes. The reason is that when only one thread is awakened (which is always the case for jobs of the size 2), there is no difference between the various wakeup schemes. They differ only when a number of threads wakeup simultaneously i.e. for a single awakened thread t, the actions taken by all three schemes are:

  1. if *t*.`processor` is idle, assign t to *t*.`processor`, otherwise
  2. if there exists an idle CPU assign *t* to the one that is the least recently active, otherwise
  3. try to preempt some currently executing task in favor of *t*.

  had there been another awakening thread, SMART would have avoided assigning it to the same CPU while SILLY might have not done the same.

- **SILLY operating on jobs with** $size \geq \frac{CPU\#}{2}$: Notice that the bars associated with SILLY are usually non existent for these job sizes. The reason is that job collections that are composed from relatively big jobs constitute an environment in which SILLY's vulnerability expresses itself the most: Consider the case in which a fine grain job collection is composed from two jobs — J_1 and J_2 — of the size 25. In this case we would expect from the scheduling algorithm to gang schedule the jobs in an alternating fashion while virtually packing them into two intersecting CPU sets ([9]), such that:

– At the "first" time slice (quantum duration) the scheduler would allow J_1's threads to continuously execute on 25 out of the 32 CPUs while J_2's threads are round-robin-like alt synchronizing on the 7 remaining CPUs.
– At the following time slice, J_2's threads (with gradually increasing priority due to spending long periods in blocked mode) would become "powerful" enough to take over (via migration) 18 out of J_1's CPUs and thus replace J_1 as the currently executing job, leaving it to alt synchronizing on its 7 remaining CPUs.

The above gives a fairly good description of the execution when AIP or SMART are used. However, when SILLY is used, `_wake_up_common` is constantly placed in the position where it must assign a big number of awakening threads to an equally big number of CPUs, and most of these assignments involve migration. In this situation, `_wake_up_common` simply doesn't turn on enough `need_resched` flags (even though all the awakened threads are powerful enough to migrate) and therefore (a) prevents the powerful job from managing to group and compute together, and (b) deprives the "weaker" group at least one CPU thus splitting it to two alt synchronizing groups. Here is the SSR achieved by the 2CS+ simulation that involved two 25-sized jobs:

Grain	SMART	AIP	SILLY
fine	93.4	94.1	35.2
medium	72	63.3	33.6

These rates coincide with the above explanation.

8.4 Wakeup-Scheme Speedup Comparison

This section will describe the difference between the various wakeup schemes in terms of speedup. Figures 8.4 and 8.5 compare between each pair of wakeup schemes for fine and medium grain respectively.

Note that the displayed speedup is an average across all the job sizes that were discussed in the previous section. When examining this figures it is clear that the difference between the various schemes is found only in the intermediate load somewhere between CPU#...2CPU#. The difference between SMART and AIP for all spin durations is negligible. SMART/AIP are up to 50%(fine grain) / 20%(medium grain) faster than SILLY. However the average speedup is much more moderate. The following is the average speedup for loads between $CPU\# + 1 \ldots 2CPU\#$. Speedup is expressed in percentage by using the same formula that was used in the previous chapter:

Grain	SMART vs SILLY			AIP vs SILLY			SMART vs AIP		
	CS	CS+	2CS+	CS	CS+	2CS+	CS	CS+	2CS+
fine	4	10	6	5	9	6	-1	1	-1
medium	2	2	3	2	2	3	0	0	0

The same table for loads $2CPU\# + 1 \ldots 10CPU\#$ contains only zeros (i.e. since almost all the spins fail anyway in such loads, wakeup schemes have no effect).

When comparing the average completion time of SMART/AIP to SILLY simulations (all sizes included), we get only a minor speedup of 5-10%. However this average is somewhat misleading: When we computed the average, we gave each load an equal weight e.g. 3-sized job simulations have a much larger relative weight in this average than of 25-sized simulations, because the former are associated with loads=33,36,39,...,63 whereas the latter are only associated with load=50. Following the example given in the previous section, the table below contains the speedups associated with the job collection composed from 2 jobs of the size 25:

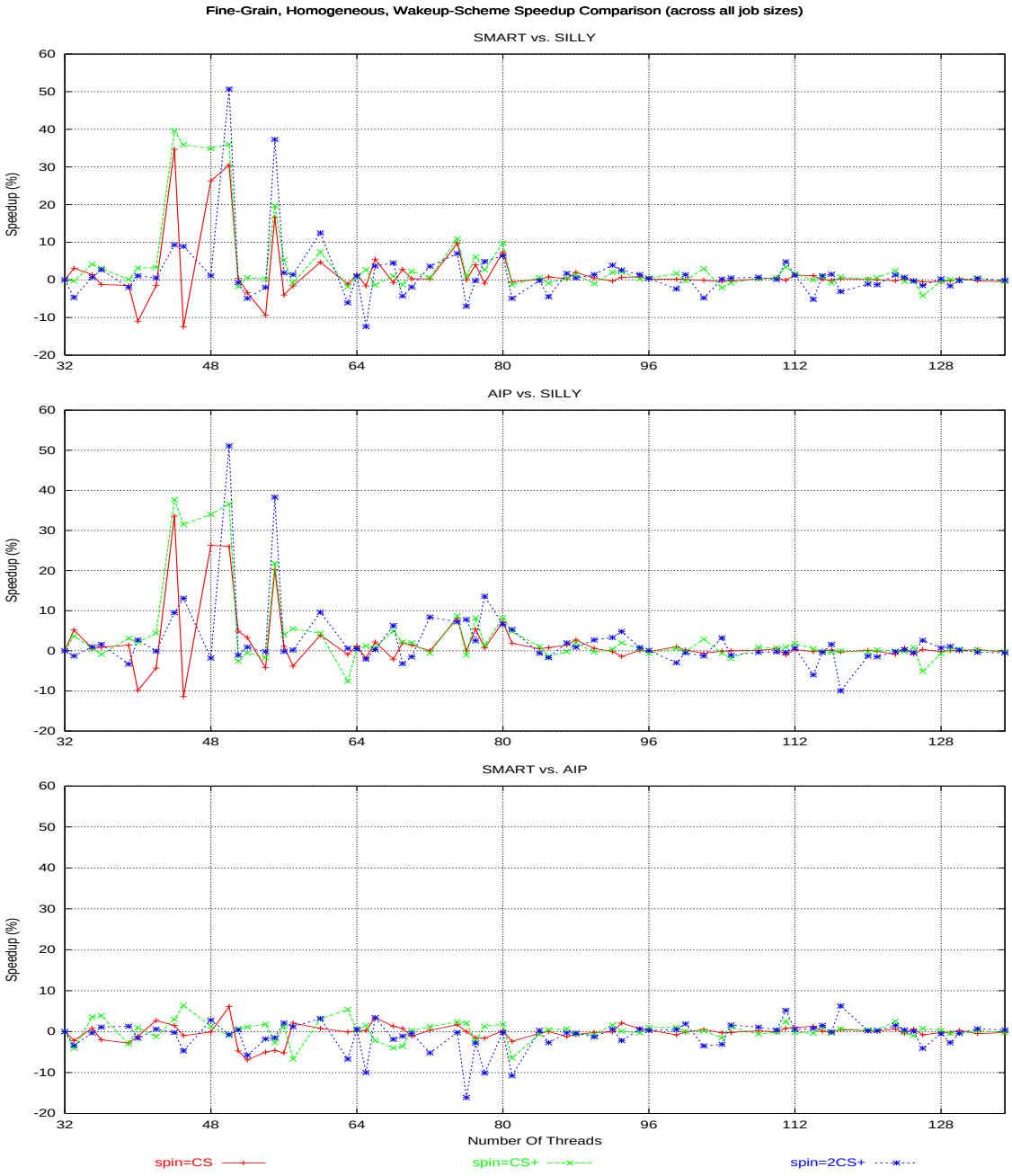

Figure 8.4: This figure displays a completion time comparison between each pair of wakeup schemes for fine grain jobs. The displayed data is an average across all job sizes (e.g. load=33 is an average between the results associated with a collection of 11 jobs of the size 3, and a collection of 3 jobs of the size 11). AIP and SMART are quite equivalent and are faster than SILLY, though only when the load is smaller than $2CPU\#$. For bigger loads, all the spins fail anyway and thus there's no difference between the various wakeup schemes.

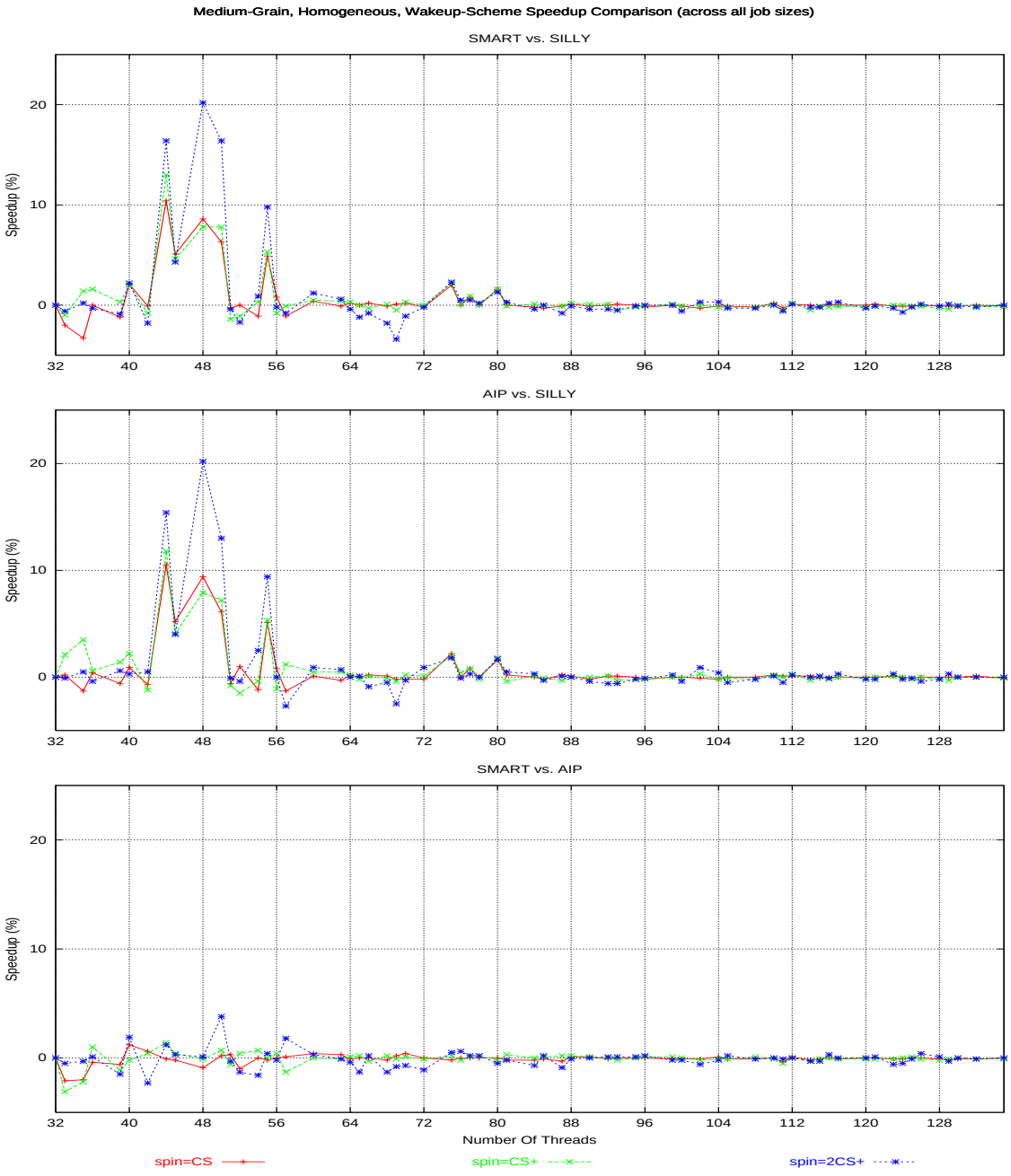

Figure 8.5: This figure is similar to figure 8.4 with the difference that the data displayed here is associated with medium grain jobs (rather than fine grain). The results in this figure are similar to those displayed in figure 8.4, but the difference between the various schemes is more moderate.

Grain	SMART vs SILLY			AIP vs SILLY			SMART vs AIP		
	CS	CS+	2CS+	CS	CS+	2CS+	CS	CS+	2CS+
fine	56	57	73	49	59	73	14	-4	-2
medium	7	10	31	5	8	24	1	2	9

Again, this shows that the speedup for these kinds of loads is substantially bigger than the average speedup that was computed across all sizes (up to factor of 1.7 for 2CS+ fine grain simulations and 1.2-1.3 for medium grain).

8.5 Maximal Spin Duration Speedup Comparison

The previous section compared between the various wakeup schemes for any given maximal spin duration (i.e. keep spin constant, change scheme). This section will compare (in terms of speedup) between the simulations completion time when using the various maximal spin durations, for any given wakeup scheme (i.e. keep scheme constant, change spin). Figures 8.6 and 8.7 compare between each pair of maximal spin durations for fine and medium grain respectively. The method of comparing is the same one used in the previous section.

When examining these figures, we can roughly divide the displayed load-range into three intervals:

1. $[\alpha] = CPU\# + 1 \ldots 2CPU\#$, where bigger maximal spin durations tend to be faster than smaller ones.
2. $[\beta] = 2CPU\# + 1 \ldots 3CPU\#$, an intermediate range in which there is no obvious "winner" i.e. sometimes bigger spin duration are faster but sometimes they are slower.
3. $[\gamma] = 3CPU\# + 1 \ldots 10CPU\#$, where the bigger the maximal spin duration is, the longer it takes a simulation to complete.

Recall that the displayed speedup is an average of all collections of jobs (which vary in size) and therefore graphs associated with different wakeup scheme look quite similar. However, when examining the speedup of jobs for which $size \geq \frac{CPU\#}{2}$ we were able to see a more meaningful difference. By again following the example given in previous sections, when comparing 2CS+ to CS, the speedup associated with the collection composed from 2 fine-grain jobs of the size 25 was of 23% when AIP was used, but of -47% (i.e. a considerable slowdown) when SILLY was used (because all spins failed, a smaller spin duration resulted in a faster completion). Having said that, we can now concentrate on the difference between the various maximal spin durations regardless of the wakeup scheme used. Therefore, we will now present the average speedup over the intervals defined above, only for AIP simulations (other wakeup schemes have similar average):

maximal spin duration	fine grain			medium grain		
	$[\alpha]$	$[\beta]$	$[\gamma]$	$[\alpha]$	$[\beta]$	$[\gamma]$
2CS+ vs CS	25	0	-34	4	-16	-24
CS+ vs CS	17	2	-4	0	-3	-3
2CS+ vs CS+	11	-4	-28	4	-13	-20

Fine grain jobs spinning for 2CS+ when load is in $[\alpha]$ are 25% faster than when spinning for CS. This coincides with all our findings so far: Since spinning may succeed within this intermediate range (as indicated by figure 8.1 page 93), then by choosing a suitable spin duration and thus actually allowing (more) spins to succeed, the system indeed reduces the overall time of execution even at the cost of longer spins. Although not effective as in the last chapter, there is no doubt spinning for 2CS+ is better than its counterparts for homogeneous job collections also (within the boundaries of $[\alpha]$). Even within the

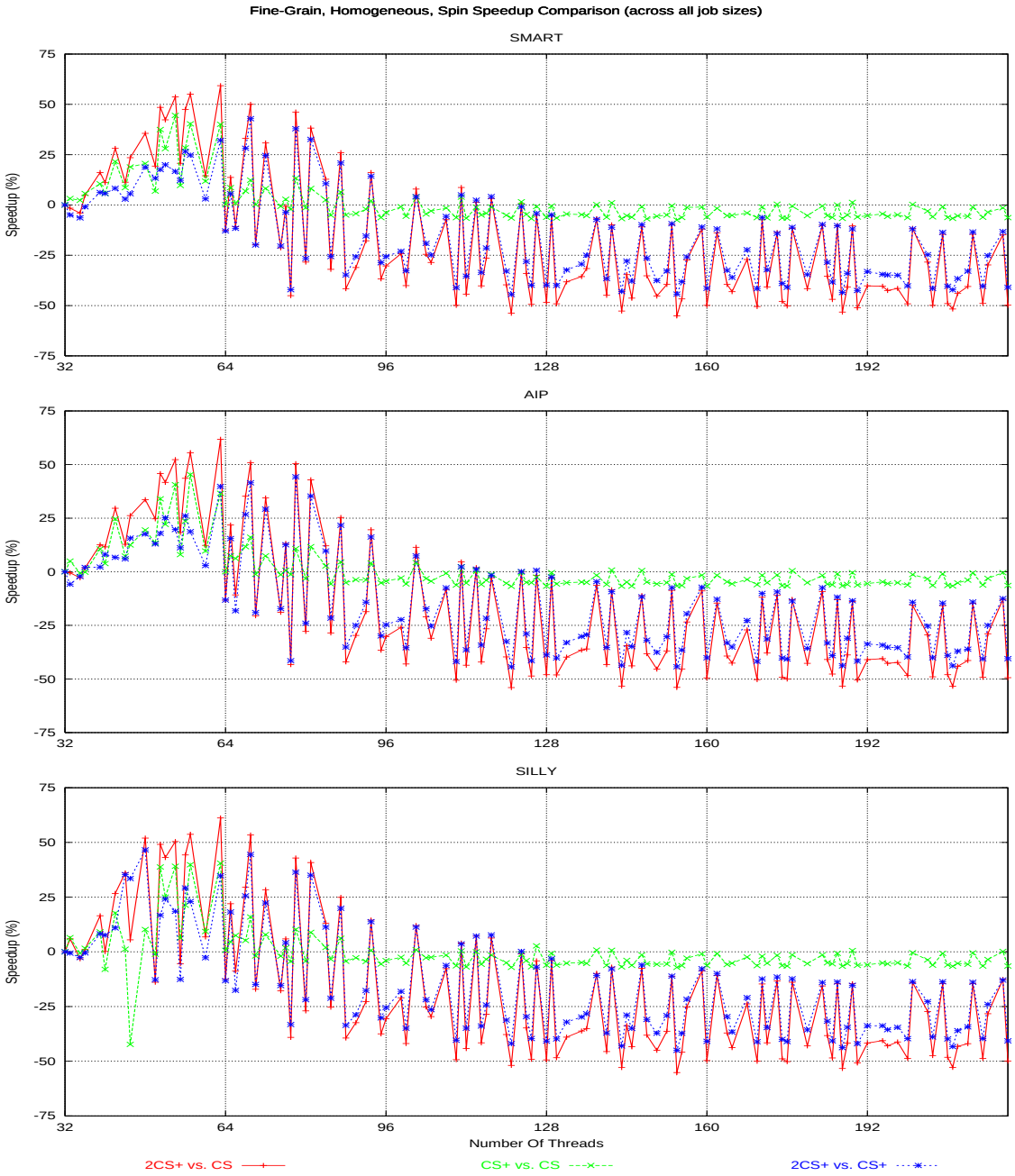

Figure 8.6: This figure displays a completion time comparison between each pair of maximal spin durations for fine grain jobs. Since the data displayed constitutes an average of all the job collections (which vary in size), the results associated with different wakeup schemes look very similar. We can roughly divide load into three intervals: (1) until $2CPU\#$ in which a bigger spin duration seems to result in a faster completion, (2) between $2CPU\# + 1$ until $3CPU\#$ in which it is not clear whether longer spinning is preferable than shorter spinning, and (3) from $3CPU\# + 1$ and onwards, in which shorter spinning results in a faster completion: The curves associated with CS+ vs CS are a bit below zero, because CS+ is bigger than CS in only one cycle; 2CS+ is much slower than its counterparts because it doubles the spinning duration.

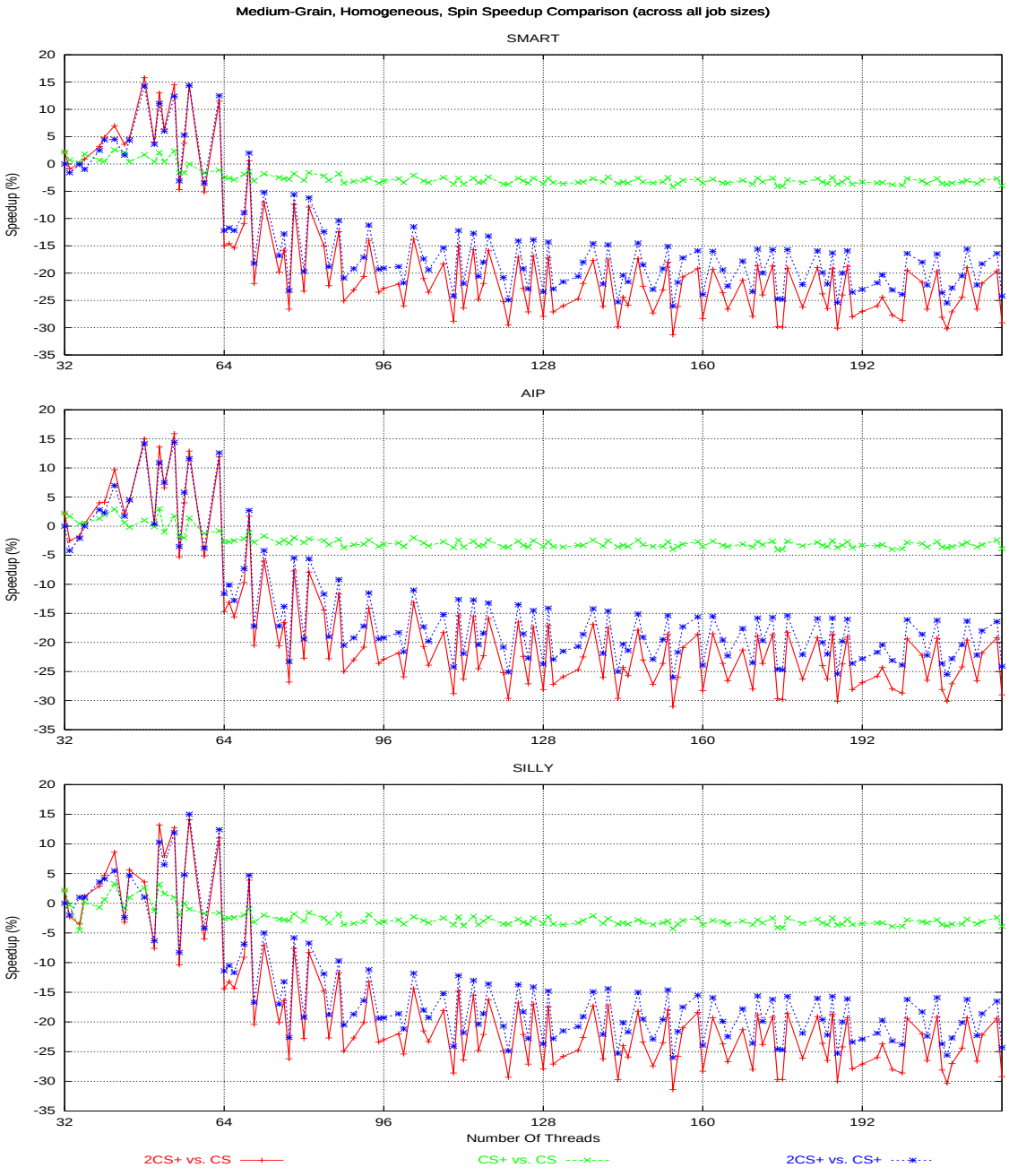

Figure 8.7: This figure is similar to figure 8.6 with the difference that the data displayed here is associated with medium grain jobs (rather than fine grain). Its analysis is quite similar, but here the load-range should be divided into two intervals (until $2CPU\#$, and from it) such that in the former spinning is probably preferable and in the latter it's not.

$[\beta]$ interval, spinning for longer durations sometimes manage to outperform the other options. However, this is balanced by simulations in which the situation was reversed, leading to a speedup/slowdown in the neighborhood of zero. Finally, in $[\gamma]$, almost all spins are unsuccessful and therefore the less a job spins, the better: The difference between CS+ and CS is very small (as indicated by the green curve that is just below 0 and by the above averages) because CS+ in these simulations is simply one cycle more than CS. Since it spins double the time, 2CS+ looses ($\approx 30\%$ on average) to both CS and CS+.

For medium grain jobs, the analysis is quite similar. The difference is that in $[\alpha]$ 2CS+ is only slightly faster than its counterparts, and $[\beta]$ is incorporated into $[\gamma]$.

8.6 Spin vs. Always-Block

In the previous section we have compared three maximal spin duration — 2CS+/CS+/CS — against each other and showed that for fine and medium grain jobs, within a load that does not exceed $2CPU\#$, spinning for 2CS+ achieves better results than the other options. However, this does not prove that spinning is a better option than not spinning at all. Theoretically, it is possible that the "always-block" policy would achieve even better results, because then no CPU time would be wasted on spinning at all. Figures 8.8 (associated with fine grain jobs) and 8.9 (associated with medium grain jobs) prove this to be wrong.

These figures compare each of the above maximal spin durations to the results achieved by a similar simulation in which the always-block policy was used. Similarly to the previous section, the load range seems to be naturally partitioned into three intervals:

1. $[\alpha] = 1 ... CPU\#$, where there are more CPUs than running threads, so there is no gain from blocking.
2. $[\beta] = CPU\# + 1 ... 2CPU\#$, the intermediate range in which it seems that spinning is still worth while. For medium grain jobs, this interval should be further divided by splitting it two sub intervals: $[\beta_1]$ and $[\beta_2]$ at $1.5CPU\#$, before and after, respectively.
3. $[\gamma] = 3CPU\# + 1 ... 10CPU\#$, where most spins are unsuccessful.

As in previous sections, the difference between the various wakeup schemes is quite small so we will allow ourselves to display the average speedup of AIP only (we remark that SILLY's averages are smaller within $[\beta]$ in 5-10%):

load	fine grain			medium grain			
	$[\alpha]$	$[\beta]$	$[\gamma]$	$[\alpha]$	$[\beta_1]$	$[\beta_2]$	$[\gamma]$
2CS+ vs always block	60	41	-107	22	10	-9	-54
CS+ vs always block	60	34	-64	22	7	-13	-29
CS vs always block	60	18	-59	21	6	-13	-25

There is nothing to gain from blocking when the number of threads isn't bigger than number of CPUs and therefore it comes as no surprise that within $[\alpha]$, spinning is preferable leading to an average speedup of 1.6 for fine grain jobs and 1.2 for medium grain jobs (across all spin duration since only a small portion of them is used).

The interesting interval is of course $[\beta]$ ($[\beta_1]$ for medium grain jobs). In this interval all the maximal spin duration achieved a positive speedup which means that for this type of jobs, spinning is always preferable than blocking, even when the number of running threads is considerably bigger than the number of CPUs (up to $2CPU\#$ for fine grain and $1.5CPU\#$ for medium grain jobs). After we've established that spinning is preferable than always blocking within this interval (regardless of the spin duration) and based on the previous section (affirmed by the above averages), we conclude that 2CS+ is indeed the preferable choice for a maximal spin duration within this interval.

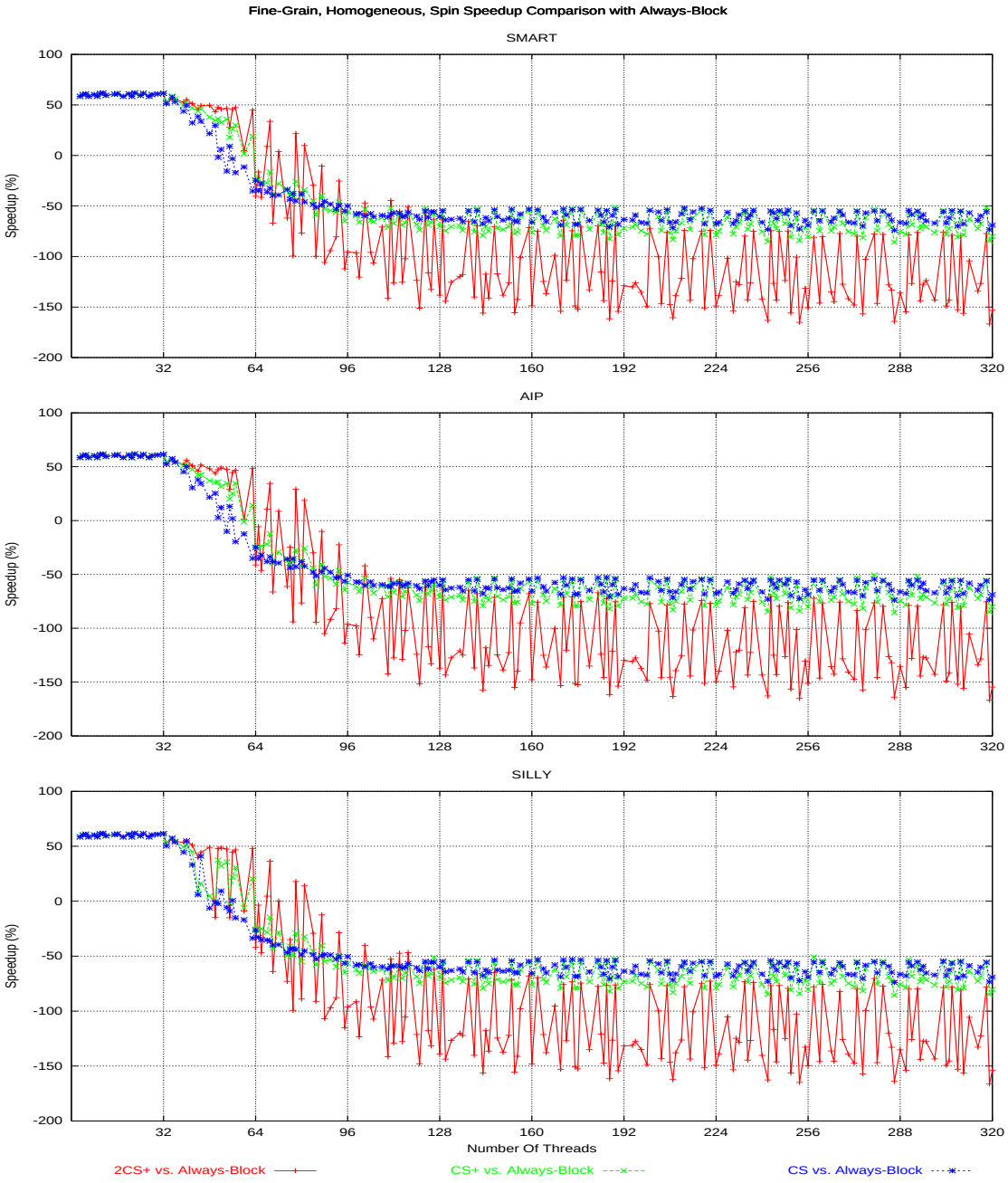

Figure 8.8: This figure compares between the maximal spin durations CS/CS+/2CS+ and the always-block policy (fine grain jobs). Obviously, when there are more CPUs than threads there is nothing to gain from blocking. In the intermediate load-range $CPU\# + 1 \ldots 2CPU\#$ it seems that 2CS+ usually maintains a speedup in the neighborhood of 50%. Afterwards, most spins are unsuccessful and it is better to avoid spinning all together.

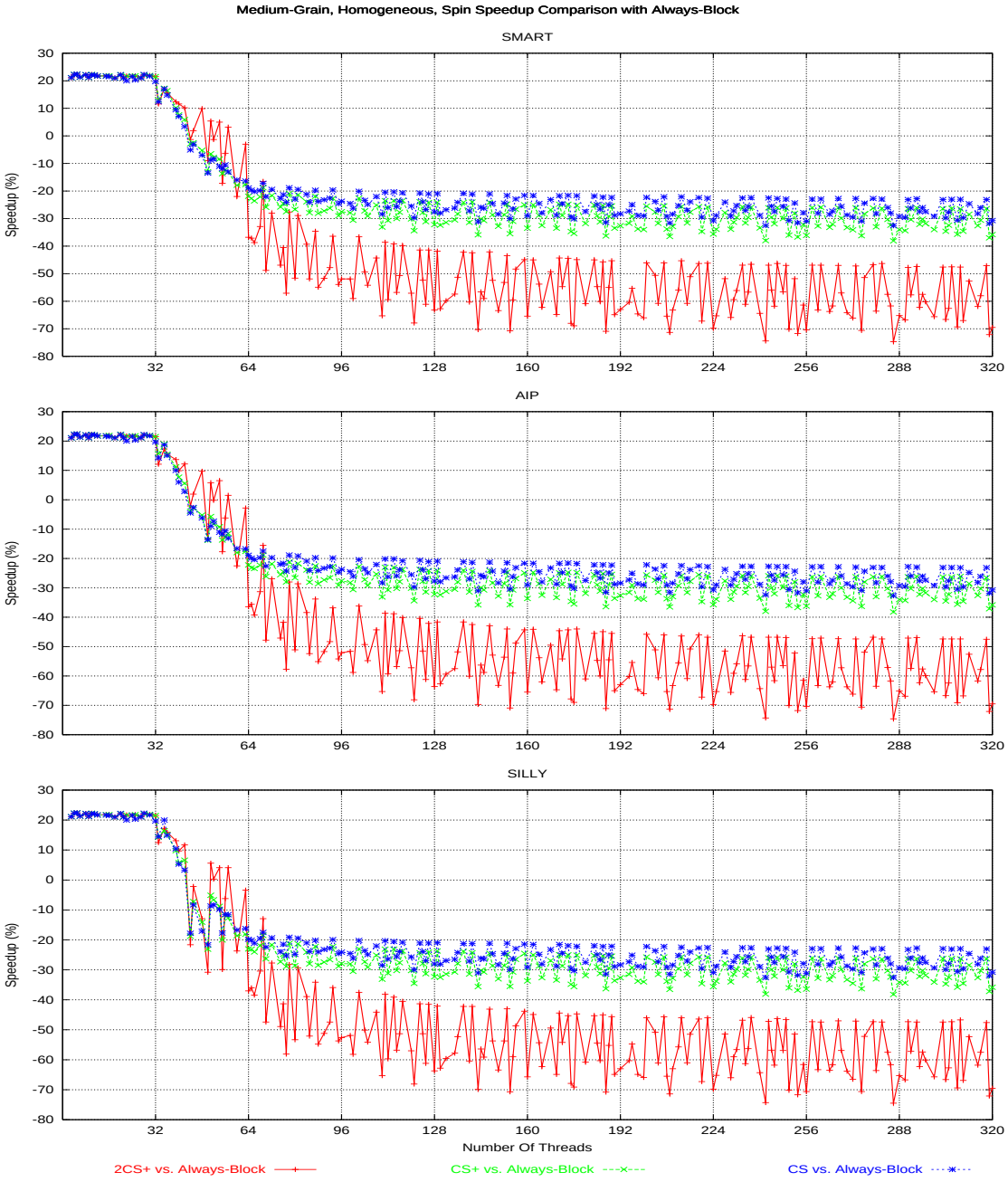

Figure 8.9: This figure is similar to figure 8.8 but is associated with medium grain jobs (rather than fine grain). The findings are the same but the speedup/slowdown is more moderate since spinning takes a smaller portion of the overall computation time.

Within $[\gamma]$, most of the spins are unsuccessful thus spinning for longer durations results in longer completion times: up to 100% slowdown for 2CS+ fine grain simulations and approximately half for CS and CS+ (slowdown is more moderate for medium grain jobs because the relative weight of spinning in respect to the total amount of computation is smaller).

8.7 Longer Spin Durations

The last issue (briefly) discussed in this chapter concerns bigger spin durations. Our experiments show that a maximal spin durations longer than 2CS+ usually lead to worse performance. Furthermore, for durations d_1 and d_2 for which $2CS+ < d_1 < d_2$, we found that on average d_1 yields shorter execution times than d_2. This result is demonstrated in figure 8.10. It seems that if an awaited (fine grain) thread didn't "show up" after a period of 2CS+ has passed, chances are slim it will show up in the "near" future.

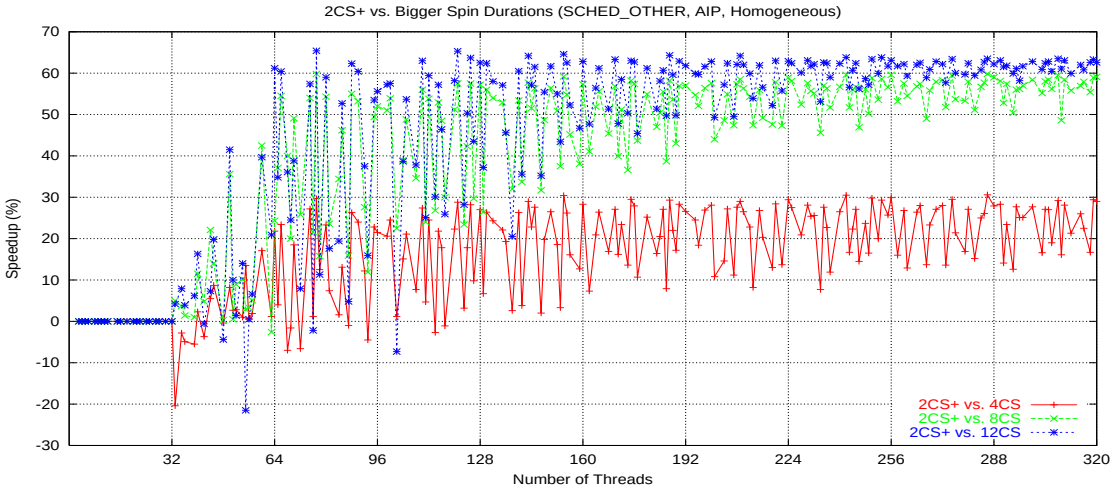

Figure 8.10: This figure displays how much faster are fine grain AIP 2CS+ simulations in comparison to similar simulations that use longer maximal spin durations of 4,8 and 12 CS. Evidently, longer spin duration leads to longer execution time.

Chapter 9

Heterogeneous Collection of Synchronizing Jons Under the Linux Scheduler

9.1 Introduction

In chapter 7 our discussion revolved around a job collection that contained a single fine grain job within a non-synchronizing environment. All the system resources and mechanisms that were designed to favor "I/O bound" processes (and were described in chapter 6) served this job and only this job, thus allowing its threads to execute simultaneously most of the time. When choosing the correct spin duration, this resulted in a close to 100% SSR regardless of the load. In other words, it is always preferable to spin rather than block within such a configuration (for a fine grain job). In chapter 8, we have presented the "opposite" scenario, namely a job collection in which all participating threads are competing on the system's resources and doing it in a similar manner (same job size, same grain). This has generated a growing burden on the system's resources until a point was reached when every act of spinning was doomed to fail. The domain found between these two extremes — non-synchronizing environment on one side and a homogeneous job collection on the other — contains all conceivable job collections. Obviously, we can't even begin to analyze every possible heterogeneous job collection. However, we can safely speculate that the results of such an analysis would be located somewhere between these two extremes.

Though we have conducted numerous simulations of various heterogeneous job collections in order to verify this speculation, it seems pointless to present their results here and analyze their every aspect (as we did in the previous chapter) because we would simply be repeating on arguments we have already stated almost word by word. Instead, we chose to give a fairly simple example of a heterogeneous job collection (constructed randomly using a large number of seeds), and through this example to demonstrate the above. We believe that such a job collection that contains a variety of jobs with different profiles, will have a positive effect on the scheduler and will allow it to be more successful in handling fine grain jobs within bigger loads. Constructing such a job collection is analogous to taking one step away from the homogeneous job collection and towards the non-synchronizing environment scenario.

9.2 Description

In order to simulate a heterogeneous job collection we used the simulator random permutation mode (as described in section 5.1.5 page 46). We will classify the threads according to their μ values. The only constraint on the chosen random job-permutations was that a representative of each job-class must appear in their "beginning" (i.e. if there are 3 job-classes, then we will find a job from each such class within the first 3 "places" of the permutation). The parameters shared by all the simulations which we will now analyze are the same as those that were used in the previous chapter (section 8.2 page 91).

The size of the jobs that will participate is expressed by the following distribution:

$$2-8\ :\ 2,\quad 9-16\ :\ 1$$

(this distribution syntax was defined in section 5.1.3 page 46). The reasons for choosing this distribution are:

1. We wanted to allow the total number of threads that participate in the simulations sequence, to gradually increase (instead of jumping for example from load 30 to load 60) so that we will be able to continuously monitor the change in the SSR. We therefore decided that only jobs with $size \leq \frac{CPU\#}{2}$ would participate in the example presented here.

2. The weights of the two intervals in the distribution were chosen such that approximately half of the threads in each simulation will belong to jobs for which $2 \leq size \leq \frac{CPU\#}{4}$, and the other half will belong to jobs for which: $\frac{CPU\#}{4} < size \leq \frac{CPU\#}{2}$. This is the reason the weight of the $'2-8'$ interval equals double the weight of the $'9-16'$ interval (i.e. for each one "big" job there are two "small" jobs). Note however that on average, there are a bit more threads belonging to "big" jobs because the expected size of such a job is 12.5 while the expectation of the size of a "small" job is 5 (which means the actual small:big thread ratio is 4/5).

The μ (computation interval expectation) of the jobs is given also in a distribution form:

$$1\%-2\%\ :\ 1,\quad 3\%-20\%\ :\ 1,\quad 21\%-90\%\ :\ 1$$

such that the first, second and third intervals are associated with fine, medium and coarse grain jobs respectively. Recall that in this simulation $\sigma = 90/15\%$ and $CS = 6\%$ and therefore 20%-of-quantum is the largest μ value for which 90% of the computation-intervals will be found within its "CS-neighborhood" (i.e. 90% of the computation intervals will be in $[\mu - 3\%,\ \mu + 3\%]$). Bigger μ values will result in a wider dispersal of computation intervals. Note that we chose to use our "traditional" value of σ (expressed as a percentage of μ) rather then to define it as a distribution. The implications of randomly choosing the σ (regardless of μ) were discussed in the analysis of the heterogeneous round robin simulations (sections 5.3 and 5.4). Our findings here regarding this issue are similar to those described in the associated round robin chapter, namely that a job's granularity is determined both by μ and by σ, if both of them are not small enough then the job should avoid spinning. Other then that, the effects of such jobs — with small μ and big σ or vice versa — on other job classes within the job collection were minor.

Much like in the previous chapter, we have compared between the various wakeup schemes and maximal spin durations. Each simulation was conducted 20 times using 20 different seeds.

9.3 Results

Within the context of this work, the SSR metric proved to be a good method for assessing whether spinning is worth while and a strong correlation between SSR and speedup was established. Therefore, we chose to present the SSR achieved by the various job classes in our example job collection. Figure 9.1 displays the SSR evolution of AIP 2CS+ simulations (only four out of the twenty seeds used are displayed).

Let's examine the graph associated with seed=0:

- The load in the last simulation that achieved $SSR > 50\%$ was 91 (surplus of 59 threads).
- The (randomly created) job collection was composed at that point from ten jobs:

job size	2	5	7	10	11	13	16
number of jobs	1	2	1	1	3	1	1

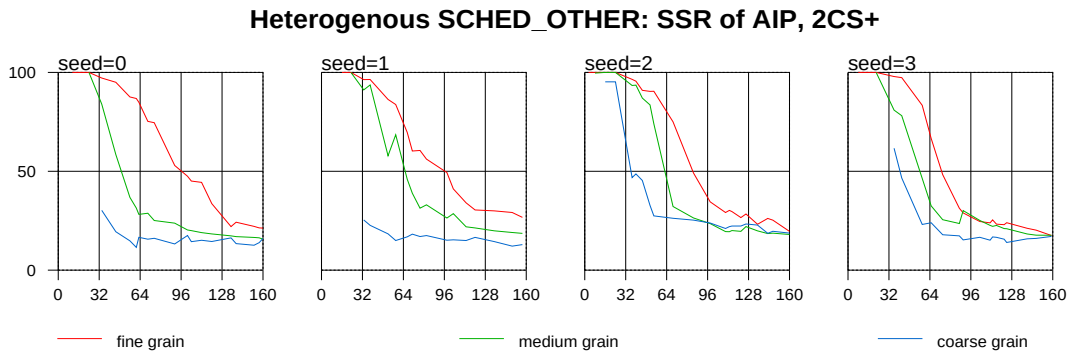

Figure 9.1: This figure displays the SSR evolving of AIP 2CS+ simulations (associated with 4 out of the 20 seeds we used).

- These jobs divided into job classes as follows:

grain	size	μ
fine	5,7,11	1
medium	5,11,13,16	7,14,17,20
coarse	2,10,11	68,74,88

It is therefore interesting to compare the fine grain curve of this graph to the homogeneous collection SSR graphs associated with sizes $5 \ldots 11$ (figure 8.1, page 93). Unsurprisingly, the former achieves better rates than those displayed by the latter.

Figure 9.2 presents the average SSR (across all seeds) achieved by fine and medium grain job classes. When comparing this surplus to the average surplus of the homogeneous simulations that was presented in the first table in section 8.3 (average over all sizes), we can see that the heterogeneous surplus is usually similar to or higher than of the homogeneous simulations (up to 14 threads difference). For example, homogeneous AIP 2CS+ fine grain simulations achieved an average surplus of 45.5 threads while the associated heterogeneous surplus is 52.2 threads. The exception to this rule is medium grain AIP/SMART 2CS+ simulations for which homogeneous simulations achieved better average than their heterogeneous counterparts. However this make sense when considering how the average surplus of the homogeneous simulations was computed: as explained earlier, smaller jobs had much higher relative weight than larger ones (recall that smaller jobs achieve better SSR). Within the heterogeneous average however, all jobs have equal weight. Actually, when considering this explanation, the difference between the heterogeneous and homogeneous fine grain surplus, seems more impressive.

Another interesting issue is the comparison between the completion time achieved by simulations using the various spin durations and the completion time achieved when always-block was used. It was hard to predict what would be the results of such a comparison because contrary to job collections in the previous chapter that only included jobs of a certain type (fine grain jobs for example), the heterogeneous collections include a variety of jobs. This means (for example) that for load $\leq 2CPU\#$, a 2CS+ maximal spin duration works in favor of the fine grain jobs within the collection. But, the collection also contains coarse grain jobs for which any spin period is a total waste of time that only delays the end of the computation. The same argument could be applied for example on the CS maximal spin duration (bad for fine grain, but better than 2CS+ for coarse grain jobs). Figure 9.3 displays this comparison. The result is quite nice: it shows that the above pros and cons were balanced in this particular heterogeneous job collection. When examining

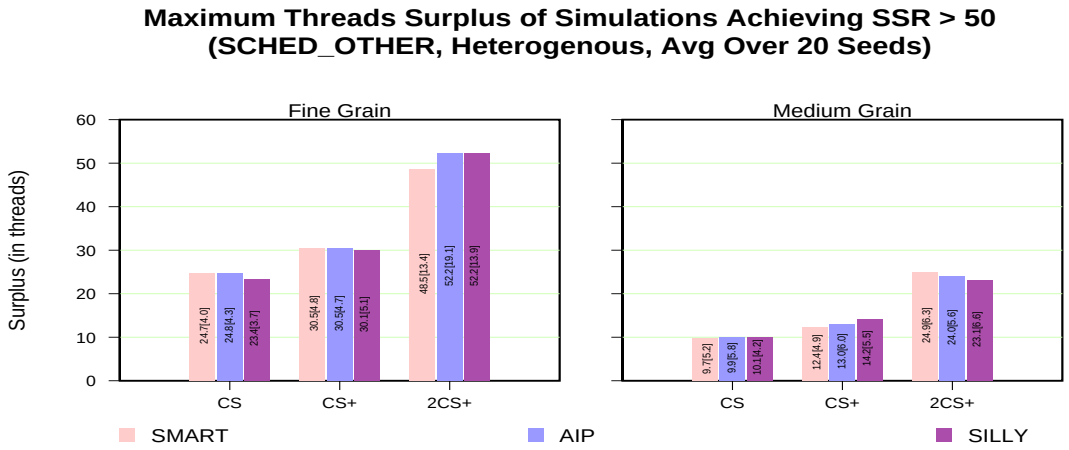

Figure 9.2: This figure displays the average surplus achieved by the heterogeneous simulations. The number within the brackets specify the absolute deviation of the surplus achieved. Usually, the surplus presented here is similar to or higher than the average surplus achieved by the homogeneous simulations. This is true even though smaller jobs had a much higher relative weight than larger ones within the homogeneous average surplus computation.

the associated homogeneous graph (the AIP section in figure 8.8) we can see that the 2CS+ curve usually presents a ~ 1.5 speedup all through the $CPU\# + 1 \ldots 2CPU\#$ intermediate range, while the CS curve presents a sharper decline and intersects with the zero axis shortly after $load = 1.5CPU\#$. In figure 9.3 this is not the case: The CS curve is much closer to the 2CS+ and both of these curves present a positive speedup until $load = 2CPU\#$. The reason for the first observation was already explained (the 2CS+ fine grain gain is balanced with its coarse grain loss). The reason for the second observation seems to be simply because of the positive effect a heterogeneous job collection has on the scheduler performance.

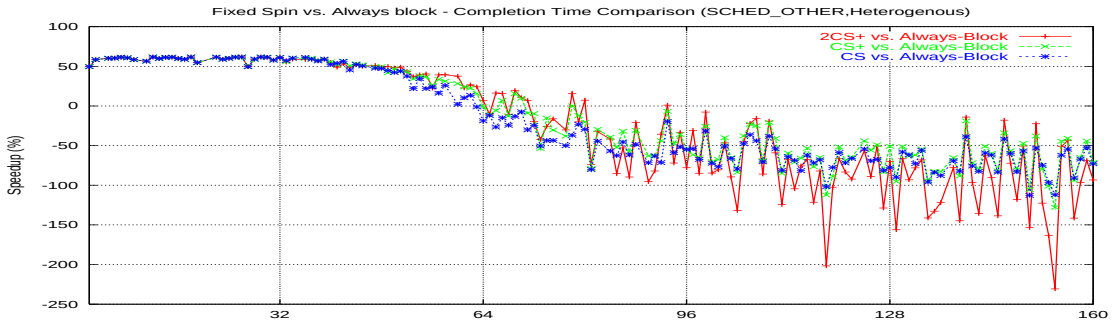

Figure 9.3: This figure displays the average speedup of the various fixed spin durations in comparison to the always-block policy. The average was computed over all seeds for AIP heterogeneous simulations. For $load \leq 2CPU\#$, it seems that CS simulations present a speedup which is much closer to the 2CS+ speedup in comparison to the results displayed in figure 8.8 for heterogeneous job collections. In addition, heterogeneous CS simulations manage to achieve a speedup for higher loads than those presented in figure 8.8.

The final point we will discuss is what happens when we increase the machine's CPU number. In order to examine this, we have conducted an additional number of simulations sequences which are similar to those described in section 9.2 with the following parameters change:

number of CPUs	size distribution
64	$2-16:2,\quad 17-32:1$
128	$2-32:2,\quad 33-64:1$
256	$2-64:2,\quad 65-128:1$

For these simulation sequences, figure 9.4 displays the comparison between the completion time achieved by 2CS+ and always-block AIP simulations (similarly to figure 9.3 but only for 2CS+ and with various machine sizes). As before, all the simulations were executed 20 times using 20 different seeds and the displayed results constitute the averages of all these executions. Evidently, as we enlarge the machine, the intermediate range in which it is preferable to spin, shrinks. Nevertheless, for larger machines in the magnitude of 128 and 256 CPUs, it is clear that while the load is smaller than $1.8CPU\#$ spinning will still achieve better performance than blocking.

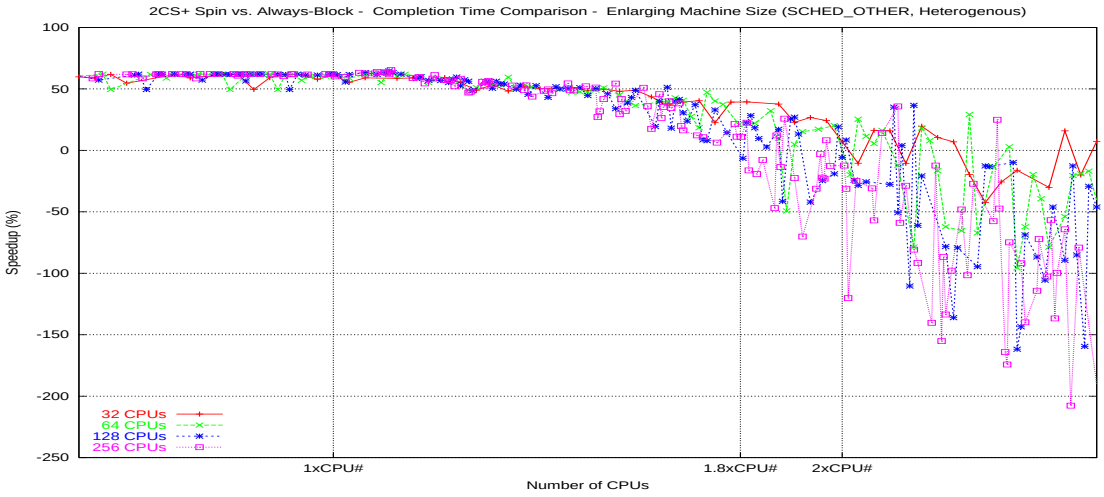

Figure 9.4: This figure displays the comparison between the 2CS+ and always block AIP simulations. Different curves are associated with machines with different number of CPUs. We can see that the effect of enlarging the machine is the shrinking of the intermediate range in which spinning is a better option than blocking.

Chapter 10

Discussion and Conclusions

Our goals in this research were to gain a better understanding of parallel-barrier based applications operating in a multitasking environment, and check the implications of high loads on such applications. We hope these understandings will serve designers and implementors of barrier algorithms. The following is a discussion and a summary of the central issues presented in this work.

Load Based Barrier Algorithm

A main contribution of this work is identifying that within the barrier synchronization context, load should be a dominant factor in the decision of whether to spin or block. Most of our empirical results have shown that when the total number of threads in the system exceeds $2CPU\#$, most spins will fail and therefore are better avoided. We have shown that any fixed spin algorithm is inferior to the always block algorithm for loads higher than $2CPU\#$. When comparing the performance of the fixed-spin (2CS+) and always-block algorithms under such loads, we have witnessed a slowdown of up to a factor of 4. We conclude that whenever there's a threads-surplus (i.e. the number of threads is bigger than the number of processors), a barrier algorithm should consider spinning only within the *intermediate load*, namely when this surplus is smaller than $CPU\#$.

We have pointed out one exception to this rule: a single synchronizing job executing within a non-synchronizing environment. For such a job collection, we have shown that by choosing the appropriate spin duration, the job's threads succeed to execute simultaneously most of the time regardless of the load (the transition point effect). We conjure this will also be the case when a small number of synchronizing jobs (with a total number of threads smaller than $CPU\#$) will execute within a non-synchronizing environment. We remark that such job collections are probably less common than the more diverse collections. The reason is that the only form of "I/O" that was done by threads in this work was barrier synchronization and therefore non-synchronizing threads played the role of CPU bound threads. This means that a "non-synchronizing environment" actually means "non-synchronizing environment of CPU bound threads". It is reasonable to assume that in real world heavily loaded systems, we will also find sequential I/O bound threads and threads that perform other types of synchronization. Nevertheless, this type of job collections present a real dilemma to the barrier algorithm described above: Although chances are that when the threads-surplus is bigger than $CPU\#$ it is better to block immediately rather than to spin, there's a positive probability that spinning is actually worth while. This dilemma will be further addressed towards the end of this chapter.

Alternating Synchronization

Another important contribution of this work is the identification of the alternating synchronization pattern: When jobs do not manage to synchronize, they tend to fall into this computation pattern, in which the job's threads form two contiguous groups in the ready queue. The first group reaches the barrier, spins, and blocks. When the second group runs, the barrier is completed, and all the processes in the first group are released again into the ready queue. Alt synchronization was found to be the dominant computation

pattern of barrier based applications executing within loaded systems. Almost all our findings are related and can be explained based on this phenomenon (e.g. the SSR asymptotes, the scenario before a transition point etc). A popular assumption among researchers is that the occurrence of synchronization events obeys some time invariant canonical probability distribution (Lim & Agarwal [21] assumed Poisson arrivals of synchronizing threads, and based on this assumptions have shown that uniform distribution is reasonable model of wait times for barrier synchronization). Alt synchronization refutes this assumption for barrier based applications (indeed, Lim & Agarwal's experimental results didn't support their theoretical model).

The SSR Metric

Within this work we have frequently used the Successful-Spin-Rate as a metric. This metric was defined to be the percentage of cases in which a thread succeeds to synchronize while spinning, excluding the last one to arrive. We have established a strong correlation between overall speedup and SSR, namely when the SSR was high (higher than 50%) it meant that spinning was worth while in terms of shortening the execution time. Generally, when evaluating a synchronization spinning strategy, the best way to do it is by measuring a suite of programs to see whether the performance is better than (for example) the traditional always-block approach. However this method's main drawback (as mentioned in [16]) is that many programs (such as operating systems components or window systems) cannot be measured by elapsed time because machine clocks' resolution might be too coarse and because the behaviors of the programs depend on many unpredictable, non-repeatable factors. In such cases the SSR metric is a reasonable option. We remark that the SSR metric must be used carefully because it can be easily abused, e.g. an always-spin algorithm will always achieve 100% SSR.

Barrier vs. Lock Synchronization

Within the context of (mutex) lock synchronization, Karlin et al. [16] have considered spinning as worth while only when the lock which a thread is attempting to acquire is held by another concurrently-running thread. We have shown that the barrier synchronization mechanism is fundamentally different than the lock mechanism in the sense that when a thread reaches a synchronization point, its very own arrival probably means that the awaited threads (in the consecutive synchronization point) are now being scheduled to run. The alt synchronization computation pattern implies that the practical meaning of following the policy suggested by Karlin et al. (in barrier context) would be to always block. This is contrary to our findings that within the intermediate load, always block is inferior to the fixed spinning policy.

Optimal Spin Duration

For our example priority based scheduling algorithm, we have shown that the very popular CS duration of fixed spinning is not enough for a fine grain parallel job attempting to complete a barrier. Indeed, a CS duration gives an awakened thread enough time to resume its execution. But, it denies from this thread the possibility to actually complete its short computation phase and therefore from reaching (in time) to the synchronization point in which its peer threads are waiting (while spinning). We have further shown that before all the threads of a job succeed to group together and execute simultaneously, the job usually performs what we referred to as "tail chasing" alt synchronization. This type of computation pattern mandates an even longer spinning period of 2CS+.

Some might be tempted to think that a 2CS+ period of busy waiting is too long. The obvious rational behind such an option is that this period is longer than the duration it takes to suspend and resume the waiting thread. However, we have shown that a spin duration of 2CS+ maximize the probability of the event in which all the threads of a job execute simultaneously. When such an event occurs then (a) frequent context switching (due to each barrier !) is avoided and (b) only a small portion of the maximal spin duration is actually used. The combination of these two factors leads to shorter overall execution time. 2CS+ has been proven to be superior to any shorter maximal spin duration (including always-block as mentioned before). Our experiments have also shown that longer maximal spin duration led to performance degradation.

We remark that within the context of communicating processes in a cluster of workstations, Arpaci-Dusseau et al. [2] have chosen 5CS to be the optimal maximal spin duration.

Job Granularity Classification

In general, we would like coarse grain jobs not to spin. The decision whether to spin or immediately block is trivial when the granularity of a job is known. This is a luxury we had in the context of this work, but a real world barrier algorithm will most likely be forced to somehow conclude or guess this information. In case this algorithm will produce a bad decision, a loaded system might suffer due to coarse grain jobs performing "hopeless" spinning. When 2CS+ is a very short period, then this issue should probably be ignored by a barrier algorithm [21]. However, this work suggests that for the Linux scheduler, such a duration might be in the order of thousands of cycles (when the system is loaded). It is therefore reasonable to consider some sort of a granularity classification mechanism. For the priority based algorithm, the usually unavoidable alt synchronization computation pattern before a job's threads succeed to execute simultaneously, suggests that guessing the granularity based on the near past spin successes/failures (such as the variable-competitive-algorithms presented in [16]) is not a good option. This is true because when a fine grain job is alt synchronizing, then most recent synchronization attempts have probably failed which will lead such an adaptive algorithm to decide to block rather than spin (even though the job has potential to soon reach a transition point). A possible alternative to these methods is for the barrier mechanism to maintain (for each thread) an average of the elapsed time between its few recent synchronization trials (within the same quantum !). The largest average constitutes a good approximation on the job's granularity for barrier synchronization purposes. On modern processors, measuring this elapsed time can be done very efficiently [8] (order of tens of cycles i.e. few nanoseconds).

Wakeup Schemes

When the last thread of a parallel job completes a barrier (i.e. it's the last to arrive to a synchronization point), the priority based scheduler checks whether consequently awakened threads (if they exist) can be immediately scheduled to execute. It is therefore faced with the problem of determining which awakened thread would be assigned to which processor. The question that follows is how much computational resources should a scheduler invest in this decision. This work has presented and evaluated three such wakeup schemes (with an increasing complexity):

1. SILLY, the is simplest wakeup scheme. It iterates through the awakened threads and tries to find the "best" processor for each such thread. The current iteration has no recollection of previous iterations' decisions.

2. AIP, which is a mildly improved version of the SILLY scheme. It performs exactly the same operations, but "remembers" its previous decisions and therefore (whenever possible) avoids assigning two awakened threads to the same processor.

3. SMART, the most sophisticated (and probably impractical) scheme. All the local considerations done by SILLY/AIP for each individual thread, are made global by SMART.

When not considering the actual cost of SMART, our findings have shown that usually SMART leads to faster results than AIP, which in turn leads to faster results than SILLY. However, they have also shown that (a) when choosing the proper maximal spin durations, and (b) when the job collection is diverse, then the difference between the various wakeup schemes is not more than 10% speedup. For job collections that were composed from fairly large jobs ($size > \frac{CPU\#}{2}$), SILLY was found to have a serious flaw and the other two schemes were found to be approximately 70% faster than it (within the intermediate load). The answer to the question of how much effort should the scheduler put in to the decision of which awakened thread to assign to which processor is therefore: not much, i.e. AIP suffices. Requiring that the wakeup scheme will not assign more than one thread to a given processor seems reasonable and easy to (efficiently) implement. In addition, AIP is much simpler than SMART, yet leads to almost identical performance.

Possible Barrier Algorithm and a Retrospect on Round-Robin

The analysis of the Round-Robin scheduler as the first phase of this work, helped in providing insights and intuition regarding the manner in which barrier based applications behave within high loads. This intuition proved to be valuable when we proceeded to the analysis of the more complex priority based Linux-2.4 scheduler. However, after completing the analysis of the latter, it too seems to shed some light on the former. The Round-Robin algorithm may be viewed as a simplified version of the more complex priority based scheduler in which all threads have equal priority all the time. When a thread reaches a synchronization point and triggers the awakening of currently blocked threads, *both* algorithms try to immediately schedule the awakened threads. The difference is found in the means at the disposal of each algorithm: a priority based algorithm may schedule an awakened thread (a) by preferably assigning the awakened thread to an idle processor or (b) by preempting another low priority thread in favor of the awakened thread. Round-Robin however, may only perform the former, i.e. if no processors are idle at the time in which a thread is awakened, then there is probably little chance for its peers to complete the next barrier successfully. This notion suggests the following barrier algorithm (in the context of Round-Robin): when a spinning thread has successfully completed a barrier *b* and the following two conditions hold:

1. as a result of the completion of *b*, other threads from its job have changed state from blocked to ready, and

2. currently there are no idle processors

the spinning thread should immediately release its processor rather than continuing for another iteration, in an effort to join the other threads in a single sequence in the ready queue, thus enhancing the chance of future synchrony. The virtues of jobs being contiguously ordered in the ready queue were demonstrated in this work when we identified and discussed the "grace period" in which the SSR is high, and spinning is generally worth while.

The above algorithm also seems to settle the conflict regarding what is the optimal maximal-spin-duration within a system that uses a Round-Robin scheduler. A thread *t* will spin if at least one of the above two condition doesn't hold: If (2) doesn't hold then there exists an idle processor and therefore *t* need not surrender its processor. On the other hand if (1) doesn't hold, then there's a strong possibility all the threads of *t*'s job are currently executing and therefore the widely accepted maximal spin duration of CS seems like a reasonable choice.

We remark that a similar algorithm may also work for priority based algorithms when changing the second condition from "currently there are no idle processors" to something like "currently the number of ready tasks is bigger than the processors number". Another idea of a fairly simple barrier algorithm (which is perhaps a generalization of the suggested algorithms, both for Round-Robin and for priority based schedulers) is for a spinning thread to release its processor if condition (1) holds, and if one of the awakened threads was not assigned a CPU by the wakeup scheme (this will often happen when the system is heavily loaded with competing jobs). Such an algorithm will solve the dilemma presented earlier in this chapter when the load based barrier algorithm was introduced.

These algorithms may overcome the CPU# surplus boundary we have presented in this work, and possibly allow successful synchronization even for diverse job collections executing within higher loads. The design and evaluation of such algorithms are left for future research.

Misfeatures of the Linux-2.4 Scheduler

Finally, a specific remark regarding the Linux scheduler. While analyzing the Linux-2.4.5 SCHED_OTHER scheduler as an example priority based scheduling algorithm, we came across three misfeatures:

1. Aside from its drawbacks which where mentioned above, the SILLY wakeup scheme introduces a race condition that might cause processors to "get lost" i.e. it's possible to have ready threads waiting for a processor, while some processors are idle. We have suggested a very simple and efficient solution to eliminate this race (AIP).

2. Changing the default quantum duration form 200ms (Linux-2.2) to 50ms (Linux-2.4) caused the value of the PROC_CHANGE_PENALTY parameter to be arbitrarily large. Its value in the current configuration precludes I/O-bound threads from preempting CPU-bound (low priority) threads when this preemption involves migration. The quantum duration change had a similar effect on both SAME_ADDRESS_SPACE_BONUS and PREEMPTION_THRESHOLD.

3. A 5 ticks quantum duration (as a result of only HZ=100 clock interrupts per second on all architectures aside from Alpha) seems to be too coarse. The current resolution of the Linux scheduler makes it impossible (for example) to define a PREEMPTION_THRESHOLD smaller than 20% of the maximal (default) priority. Recent studies [7] have shown that HZ=1000 (and therefore 50 ticks per quantum) seems to be feasible.

Bibliography

[1] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. H. Lim, G. Maa, and D. Nussbaum, "*The MIT Alewife machine: A large-scale distributed-memory multiprocessor*". In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*, Kluwer Academic, December 1991.

[2] A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring, "*Scheduling with implicit information in distributed systems*". In 1st *Measurement amd Modeling of Computer Systems*, vol. 26, pp. 233–243, SIGMETRICS, June 1998.

[3] T. Aviazian, *Linux Kernel 2.4 Internals*. http://www.linuxdoc.org/LDP/lki/index.html, August 2000. Online Document.

[4] D. P. Bovet and M. Cesati, *Understanding the Linux kernel*. O'Reilly & Associates Inc, first ed., January 2001.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, chap. 27, pp. 600–604. MIT Press, first ed., 1990.

[6] D. E. Culler and J. P. Singh, *Parallel Computer Architecture*. Morgan Kaufmann Publishers Inc., second ed., 1999.

[7] Y. Etsion, *How Should the UNIX Scheduler Handle Modern Interactive Processes*. Master's thesis, School of Computer Science and Engineering, the Hebrew University of Jerusalem, 2002. To be published.

[8] Y. Etsion and D. G. Feitelson, *Time Stamp Counters Library Measurements with Nano Seconds Resolution*. Technical Report, School of Computer Science and Engineering, the Hebrew University of Jerusalem, July 2001. http://www.cs.huji.ac.il/labs/parallel/tsclib.ps.

[9] D. G. Feitelson, "*Packing schemes for gang scheduling*". In *Job Scheduling Strategies for Parallel Processing*, vol. 1162, D. G. Feitelson and L. Rudolph (eds.), pp. 89–110, Springer-Verlag, April 1996.

[10] D. G. Feitelson and L. Rudolph, "*Gang scheduling performance benefits for fine grain synchronization*". *Journal of Parallel & Distributed Computation* **16(4)**, pp. 306–318, December 1992.

[11] H. Franke, M. Kravetz, and B. Hartner, *Scalable Scheduling for Linux: an Open Project*. http://lse.sourceforge.net/scheduling/, 2001. IBM Linux Technology Center, Online Document.

[12] B. O. Gallmeister, *POSIX.4: Programming for the Real World*. O'Reilly & Associates Inc, first ed., January 1995.

[13] A. Gottlieb and G. Almasi, *Highly Parallel Computing*. Benjamin/Cummings, first ed., 1989.

[14] D. Jiang and J. P. Singh, "*Scaling application performance on a cache-coherent multiprocessor*". In 2nd *Proceedings of the 26th annual International Symposium on Computer Architecture*, vol. 27, pp. 305–316, Atlanta, Georgia, May 1999.

[15] A. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki, "*Competitive randomized algorithms for non-uniform problems*". In *Proceedings of the first annual ACM-SIAM symposium on Discrete Algorithms*, pp. 301–309, San Francisco, CA USA, January 1990.

[16] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki, "*Empirical studies of competitive spinning for a shared-memory multiprocessor*". In *Proceedings of the 13th Annual ACM Symposium on Operating Systems Principles*, pp. 41–45, ACM, October 1991.

[17] S. W. Keckler, W. J. Dally, D. Maskit, N. P. Carter, A. Chang, and W. S. Lee, "*Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor*". In 3rd *Proceedings of the 25th annual International Symposium on Computer Architecture*, vol. 26, pp. 306–317, Barcelona, Spain, June 1998.

[18] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott, "*Scheduler-conscious synchronization*". *ACM Transactions on Computer Systems* **15(1)**, pp. 3–40, February 1997.

[19] M. Kravetz and H. Franke, *Implementation of a Multi-Queue Scheduler for Linux*. http://lse.sourceforge.net/scheduling/mq1.html, April 2001. IBM Linux Technology Center, Online Article.

[20] J. Laudon and D. Lenoski, "*The SGI origin: A ccNUMA highly scalable server*". In *Proceedings of the 24th annual International Symposium on Computer Architecture*, vol. 25, pp. 241–251, Denver, CO USA, June 1997.

[21] B-H. Lim and A. Agarwel, "*Waiting algorithms for synchronization in large-scale multiprocessors*". *ACM Transactions on Computer Systems* **11(3)**, pp. 253–294, August 1993.

[22] S. Maxwell, *Linux Core Kernel Commentary*. The Coriolis Group LLC, first ed., 1999.

[23] J. K. Ousterhout, "*Scheduling techniques for concurrent systems*". In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pp. 22–30, New York, October 1982.

[24] E. Segar, *Thimble Theatre, Popeye the Sailor Man*. King Features Syndicate, 1929.

[25] C. P. Thacker and L. C. Stewart, "*Firefly: A multiprocessor workstation*". *IEEE Transactions on Computers* **37(8)**, pp. 909–920, August 1988.

[26] L. Torvalds, *The Linux-2.4.5 Kernel Source Code*. http://www.kernel.org/pub/linux/kernel/v2.4/linux-2.4.5.tar.gz, 2001. Online File Archive.

[27] L. Torvalds, A. Cox, H. Franke, M. Kravetz, and I. Molnar, *A Quest for a Better Scheduler*. http://www.geocrawler.com/mail/thread.php3 ?subject=a+quest+for+a+better+scheduler&offset=0&list=35, April 2001. Thread from the Linux Kernal Mailing List.

[28] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "*SPLASH-2 programs: characterization and methodological considerations*". In *Proceedings of the 22th annual International Symposium on Computer Architecture*, vol. 23, pp. 24–36, Santa Margherita Ligure, Italy, June 1995.