Accelerating by Idling: How Speculative Delays Improve Performance of Message-Oriented Systems

Aleksandar Prokopec[1]

Oracle Labs, Switzerland

Abstract. We propose a technique called *speculative lagging*, which improves performance by dynamically adding periods of idle execution into the message-oriented system. The speculation is guided by a statistical model, which predicts context switches that benefit from delays. We analytically derive the expected speedup, which, for a fixed confidence, allows identifying lagging opportunities in $O(1)$ time, without a performance overhead. We describe the corresponding speculation algorithm and use it to extend an existing scheduler. Comparison with other actor frameworks on standard benchmarks shows improvements of up to $2.1\times$.

1 Introduction

Consider a system with concurrent processes that communicate by exchanging messages. We call these processes *actors*. When a message arrives, it is placed on the message queue of the corresponding actor, and we say that it is *available*. The system is tasked with assigning CPUs to any number of actors with available messages. An actor cooperatively yields the CPU back to the system after emptying its message queue. Assigning and yielding is called *context switching* – this is *the period of time required to switch the CPU between two actors*.

The main question in this paper is the following: can the overall performance of a message-based system be improved by slowing down individual actors with periods of idle execution? Generally, adding extra execution cycles to a program slows it down, so the first reaction is to say no. Counter-intuitively, this paper shows that selectively adding periods of idle execution improves performance. The essential idea is that it can be less costly to wait for another message, than it is to undergo a context switch when there are no messages. The key difficulty addressed in the paper is to quickly detect (at runtime) that a program benefits from delays, apply those delays selectively to some actors, and do so without compromising the performance of programs that do not benefit from delays.

This paper brings forth the following contributions:

- A probabilistic model of *speculative lagging*, a new runtime technique that increases program performance by $O((1 + \delta - P)^{-1})$, where P is the probability that a relative delay δ is beneficial for a given program (Sect. 2), along with a decision criteria for applying speculative lagging (Sect. 2.2).

– A sampling strategy that, for some fixed confidence α, when speculation is beneficial, is expected to correctly decide in $O(1)$ time, and when delays are not beneficial, concludes this in $O(\varphi^{-1})$ time, where φ is the allowed performance overhead from sampling (Sect 2.1 and 2.3).
– An algorithm and an implementation of speculative lagging (Sect. 3).
– An evaluation on standard actor benchmarks [12], where we identify specific benchmarks on which speculative lagging achieves up to 2.1× speedups, without any noticeable performance overhead otherwise (Sect. 4).

2 A Model of Speculative Lagging

In this section, we construct a model of speculative lagging. The speculation is based on the bet that a context switch is expensive, and that another message will arrive between the start and the end of the context switch. We investigate how each actor determines the minimal number of messages to receive before making a speculation decision, how it decides whether to speculate or not, and how to minimize the time until making the decision.

2.1 Determining the sample size

To decide whether speculation is beneficial, an actor must have a sample – a set of delayed context switches. We start by estimating the necessary sample size.

Definition 1. *Consider an actor that, upon processing a message at some time t, waits for a fixed duration of time d before returning control to the scheduler. A* speculation hit *is the event in which at least one other message arrives in the time interval $\langle t, t+d \rangle$. A* speculation miss *is the event in which no message arrives in the interval $\langle t, t+d \rangle$.*

Definition 2 (Delay sampling). *Consider a sampling strategy in which an actor, before context switching, waits some fixed time d with a probability φ, and counts speculation hits. We call this process* speculation hit sampling*.*

Theorem 1 (Speculation hit estimate). *Consider a speculation hit sampling process that estimates the probability P of a speculation hit, which is independent between speculation hits. Let n be the sample size, and h_i a random variable, equal to* 1 *if there was a speculation hit in the i-th sampling iteration, and* 0 *otherwise. The sampled probability $\hat{p} = \overline{p} = \frac{1}{n}\Sigma_i h_i$ is a consistent estimate for P.*

An estimate $\hat{p}$ is consistent for the value P if $\hat{p} \to P$ when $n \to \infty$. We assumed that the probability P is independent between speculation hits, a method which is called *simple random sampling* [5]. It was shown in the related work [5] that $\hat{p}$ is in this case a consistent estimate for P.

Theorem 2 (Sample size). *Consider a speculation hit sampling process that estimates the speculation hit probability P with $\overline{p} = \frac{1}{n}\Sigma_i h_i$. Let α be the probability that a* ***normally distributed value*** *is within the $\pm z_{1-\alpha/2}$ range (α uniquely* *defines $z_{1-\alpha/2}$). The minimum sample size is at least $\frac{z^2_{1-\alpha/2}}{4 \cdot e^2}$, for the probability α that the estimated probability P is approximately within the range $\langle 0.1, 0.9 \rangle$.*

Proof. We are sampling speculation hits with replacement, so the number of observed speculation hits follows the binomial distribution. It was shown that in this case, the confidence interval e can be approximated with $z_{1-\alpha/2} \cdot \sqrt{\hat{p}(1-\hat{p})/n}$, where n is the sample size [5]. The term $\hat{p}(1-\hat{p})$ is maximized for $\hat{p} = 0.5$, which allows deriving the worst case n from e. $\square$

The result in Theorem 2 allows us to calculate the minimum number of samples required for deciding, with a specific confidence α, the interval of the possible values of the speculation probability P. As we show later, this allows deciding whether speculation improves program performance or not.

2.2 Estimating speculation benefits

Theorem 2 shows how to pick the sample size, but does not mention the delay time d. The probability P, and the estimated speculation hit probability $\hat{p}$, both depend on the chosen delay d. In this section, we investigate how to find the optimal value for the delay d. We first show that d is not unbounded – an actor only needs to search through a finite interval to find the optimal value d.

Definition 3. *The* setup cost c_s *is the time between the point when the scheduler assigns an actor to a processor, and the point when the actor starts processing the first pending message. The* teardown cost c_t *is the time between the point when the actor finishes processing the last pending message, and the point when the actor returns control to the scheduler. We define the* context-switch cost c *as the sum $c_s + c_t$ of the setup and teardown cost.*

Definition 4. *An actor* speculates with a delay d *if, after processing the last available message, it spends an additional time d waiting for the arrival of another message. The* speculation efficiency $\Psi_{spec} = T_{spec}/T_{base}$ *is the ratio between the total actor execution time with speculation T_{spec} and the time without speculation T_{base}, and its inverse value $S_{spec} = \Psi^{-1}_{spec}$ is the* speculation speedup.

Lemma 1 (Delay bound). *Consider an actor that speculates with a delay d. Let there always exist at least one inactive actor with an available message. Then for every $d > c$, the program execution time is not optimal.*

Proof. Assume that there is some time $d_0 > c$ for which the execution is optimal. Consider a specific actor R that, at some point in the execution schedule E_0, speculates with d_0. By assumption, when R starts speculating, there exists another actor Q with an available message. Consider now an alternative execution schedule E_1 in which the actor R does not speculate, but instead releases the processor. The scheduler can then assign the processor time to another actor Q, after a context switch with duration c. The execution schedule E_1 is at least $d_0 - c$ faster than E_0. By contradiction, E_0 is not optimal. $\square$

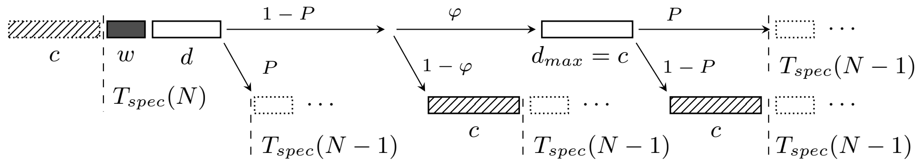


Fig. 1. Probability tree for the running time with speculative lagging

Lemma 1 states that there exists an upper bound on the benefits from speculatively delaying the context switch. Consequently, the sampling plan needs to focus only on the values of d in the interval $\langle 0, c\rangle$. We now investigate how to choose d from this interval to maximize the benefits.

Lemma 2 (Speculative running time). *Consider an actor that speculates with a delay d, and with probability $\varphi \ll 1$, prior to context switching, samples speculation hits. The time required to receive and process N messages is then $T_{spec}(N) = [(1-P)\cdot c + w + d]\cdot N$, where P is the speculation hit probability for the delay d, c is the context switch time, and w is the time to process a message.*

Proof. Consider the execution of an actor in Fig. 1, which has one message available at the beginning, and still has to process N messages. The time required to process these messages is $T_{spec}(N)$. The actor spends w time to process the first message, and d time speculating for the next message. During the time d, another message arrives with probability P, which brings the actor into a state with one available message, where the remaining execution time is recursively $T_{spec}(N-1)$. Alternatively, with a probability $1-P$, another message does not arrive during the time d. In this case, the actor may decide to return control to the scheduler with a probability $1-\varphi$ (i.e. not to sample speculation hits) and spend c time in a context switch. Alternatively, with a probability φ, the actor spends additional $d_{max} = c$ units of time sampling speculation hits. When the sampling ends, another message will appear with probability that is at least P, bringing the actor into the state with an available message – the remaining execution time is here $T_{spec}(N-1)$. Alternatively, with probability $1-P$, no message arrives, and the actor spends c time in a context switch.

Putting this together, we get the following execution time recurrence:

$$T_{spec}(N) = w + d + \Big((1-P)\cdot\varphi + (1-P)\cdot(1-\varphi) + (1-P)^2\cdot\varphi\Big)\cdot c + T_{spec}(N-1)$$

This recurrence has the following closed-form solution:

$$T_{spec}(N) = (w + d + (1-P)\cdot(1+\varphi-\varphi\cdot P)\cdot c)\cdot N \tag{1}$$

Under the assumption that sampling is done infrequently, that is, $\varphi \ll 1$, the term $1+\varphi-\varphi\cdot P$ becomes 1, and the claim follows. $\square$

Theorem 3 (Speculation efficiency and speedup). *Let c be the context switch duration, w the time required to process a message, and $\overline{p}$ be the sampled speculation hit rate. Define $\delta = d/c \in \langle 0,1\rangle$ and $\eta = w/c$ as ratios between the delay, the work and context switch time. Then, the expected efficiency of speculative lagging is $\overline{\Psi}_{spec} = 1 + \frac{\delta-\overline{p}}{1+\eta}$, and the expected speedup $\overline{S}_{spec} > \frac{1+\eta}{1+\eta+\delta-\overline{p}}$.*

Proof. The expected efficiency is defined as $E[\Psi_{spec}] = E[T_{spec}(N)/T_{base}(N)]$, where $T_{base}(N)$ is the time required to process N messages without speculation, and $T_{spec}(N)$ is the same time with speculation.

Without speculation, in the worst case, an actor always processes only a single message before returning control to the scheduler. The time without speculation is then $T_{base}(N) = (c + w) \cdot N$. By Lemma 2, expected time with speculation is $T_{spec}(N) = [(1 - P) \cdot c + w + d] \cdot N$. Since $T_{base}(N)$ does not depend on random variables, linearity of expectation gives us:

$$\overline{\Psi}_{spec} = E[\Psi_{spec}] = \frac{N \cdot [(1 - E[P]) \cdot c + w + d]}{N \cdot (c + w)} \tag{2}$$

Using $E[P] = \overline{p}$ from Theorem 1, this further simplifies to:

$$\overline{\Psi}_{spec} = 1 + \frac{\delta - \overline{p}}{1 + \eta} \tag{3}$$

By expressing the inverse Ψ_{spec}^{-1} of Ψ_{spec} with its Taylor series, it can be shown that $\overline{S}_{spec} = E[S_{spec}] = E[\Psi_{spec}^{-1}] > E[\Psi_{spec}]^{-1}$, as noted previously [9]. □

The result from Theorem 3 provides a way to decide whether speculative lagging improves performance. This is captured with the following corollary.

Corollary 1 (Speculation decision). *An actor that speculates with a relative delay $\delta = d/c \in \langle 0, 1 \rangle$ improves program performance when $\delta \leq \overline{p}$.*

Proof. Program performance is expected to improve when the expected speedup $\overline{S}_{spec} > 1$. Using the result from Theorem 3, we have:

$$\overline{S}_{spec} > \frac{1 + \eta}{1 + \eta + \delta - \overline{p}} \geq 1 \tag{4}$$

We can rewrite the second inequality as $1 + \eta \geq 1 + \eta + \delta - \overline{p}$, and the result follows irrespective of the work ratio η. □

The previous corollary states the necessary conditions to apply delays. Let's assume that we have a set of $(\delta_i, \overline{p}_i)$ pairs, and we need to pick the δ_i that maximizes speedup. The next corollary shows how to do this.

Corollary 2 (Speculation choice). *Given a set of $(\delta_i, \overline{p}_i)$ pairs, where δ_i is the speculated delay and $\overline{p}_i$ is the respective sampled speculation hit probability, speedup is maximal for the relative delay δ_i that has the minimum $\delta_i - \overline{p}_i$.*

Proof. Expression for $\overline{S}_{spec}$ from Theorem 3 is monotonic with respect to $\overline{p}$ and δ, and is maximized when $\overline{p} - \delta$ is minimized, irrespective of η. □

2.3 Better time-to-speculation with an adaptive sampling rate

We can use the Theorem 2 to estimate the sample size n. For example, for a $\alpha = 95\%$ confidence that speculation hit probability P lies within $e = 15\%$ of the true value, we need $n = 43$. Our analysis implicitly assumed that the sampling frequency φ is so low that it can be ignored. As an example, for $\varphi = 1\%$, execution overhead less than 1%, but the expected number of context switches that an actor must undergo before deciding on speculation is $N = n \cdot \varphi^{-1}$, which is 4300 for the confidence $\alpha = 95\%$ and the interval $e = 15\%$, assumed previously. This value is impractical for applications in which the actor lifetime is short.

To reduce the time until a speculation decision, we note that increasing the sampling frequency φ causes a slowdown only if speculation hits are unlikely. If speculation helps, it is likely that a message arrives during sampling. In what follows, we substantiate this intuition with an upper bound on the allowed sampling rate φ. We then compute an upper bound on the expected number of messages $\overline{N}$ that must be received before deciding on δ, when φ gets dynamically adapted.

Lemma 3 (Sampling rate bound). *Consider an actor that speculates with a relative delay $\delta = d/c$, and, before a context switch, samples speculation hits with the rate φ. Sampling does not decrease performance as long as $\varphi \leq \frac{P-\delta}{(1-P)^2}$.*

Proof. We are investigating the upper bound for φ, so the previous assumption that $\varphi \ll 1$ no longer holds, and we cannot use the result from Theorem 3. Instead, we rely on (1) from Lemma 2. We require that the speculative running time is less than equal than the baseline:

$$w + d + c \cdot (1 - P) \cdot (1 + \varphi - \varphi \cdot P) \leq w + c \tag{5}$$

By simplifying and substituting $\delta = d/c$, we get the desired bound on φ. □

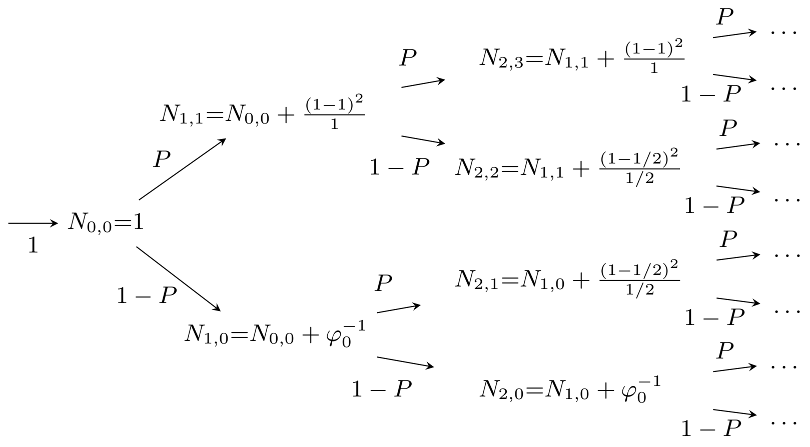
We could (prematurely) conclude that a newly created actor must set the sampling rate φ to the bound from Lemma 3. However, a new actor does not know the speculation hit probability P. At the end of each sampling iteration i, the actor only has an imprecise estimate p_i of P, which, as shown in Theorem 2, is only adequately accurate with the desired sample size $n \geq i$.

We now show that, somewhat surprisingly, the actor can indeed use its current estimate p_i as a proxy for P to increase the sampling rate φ, and reduce the expected number of messages $\overline{N}$ needed to reach the sample size n, without compromising performance. The bound on $\overline{N}$ will be proportional with the speculation miss probability $1 - P$, confirming the initial intuition.

Theorem 4 (Adaptive sampling). *Consider a newly created actor that starts with an initial sampling frequency φ_0, and then changes the sampling frequency to $\varphi = max(\varphi_0, min(1, \frac{\overline{p}_i}{(1-\overline{p}_i)^2}))$ after every sampling iteration i. The expected number of messages $\overline{N}$ that an actor will receive before gathering the sample of size n is bounded by $\overline{N} \leq 1 + \varphi_0^{-1} \cdot (1 + (2 + \varphi_0)^{-1}) \cdot (1 - P) \cdot O(n) + O(n)$. Expected speedup $\overline{S}$ between speculation and baseline is bounded by $\overline{S} \geq 1 - (1 + \varphi_0^{-1})^{-1}$.*

Proof. After each sampling event, the sampling probability φ is modified using the expression $\frac{P-\delta}{(1-P)^2}$ from Lemma 3, where δ is 0 because the newly created actor does not speculate yet, and P is replaced with the current value $\overline{p}$.

The probability tree in Fig. 2 shows a series of sampling events, and the number of messages received $N_{i,j}$, which are needed to reach the respective sampling iteration i and state j. $N_{0,0}$ is 1, since a newly created actor can immediately



1
$N_{0,0}{=}1$
P
$1-P$
$N_{1,1}{=}N_{0,0}+\frac{(1-1)^2}{1}$
$N_{1,0}{=}N_{0,0}+\varphi_0^{-1}$
$N_{2,3}{=}N_{1,1}+\frac{(1-1)^2}{1}$
$N_{2,2}{=}N_{1,1}+\frac{(1-1/2)^2}{1/2}$
$N_{2,1}{=}N_{1,0}+\frac{(1-1/2)^2}{1/2}$
$N_{2,0}{=}N_{1,0}+\varphi_0^{-1}$
$\cdots$

Fig. 2. Probability tree and message counts in adaptive sampling

sample once – the sampling cost is amortized by the actor creation costs. Sampling can result in a speculation hit with a probability P, or a speculation miss with $1-P$, where P is the true speculation hit probability. The expected number of messages between two sampling iterations is φ^{-1}, so we have:

$$N_{i,j} = N_{i-1,j\div 2} + min\Big(\varphi_0^{-1}, \frac{(1-\overline{p_i})^2}{\overline{p_i}}\Big) \tag{6}$$

Our task is compute the expected number of messages after n sampling iterations, in other words, to produce a sum of the messages received at the depth n in the tree, weighted by the respective probabilities. This is given with the expression $\overline{N} = \Sigma_j P_{n,j} \cdot N_{n,j}$, where $P_{n,j}$ is the probability of the outcome j after n sampling iterations. We now compute the upper bound for $\overline{N}$ by grouping the execution paths according to the number of speculation hits k, and choosing the longest path in each such group k. The execution is longest when the initial k sampling iterations are speculation misses, followed by $n-k$ speculation hits.

$$\overline{N} \le 1 + \sum_{k=0}^{n} \binom{n}{k} P^{n-k}(1-P)^k \Big(k \cdot f \cdot \varphi_0^{-1} + \sum_{i=k\cdot f+1}^{n} \frac{\left(1-\frac{i-k}{i}\right)^2}{\frac{i-k}{i}}\Big) \tag{7}$$

Note that in the min in (6), the second term does not outweigh φ_0^{-1} at $i = k+1$, but only after a few additional iterations. For this reason, we include the factor f in (7). It can be shown that the upper bound holds when $f = 1 + (2+\varphi_0^{-1})^{-1}$. Next, note that in the given range, the fraction in the last sum is always less than $i/(i-k)$. Therefore, we can use the following upper bound for the last sum:

$$\sum_{i=k+1}^{n} \frac{\left(1-\frac{i-k}{i}\right)^2}{\frac{i-k}{i}} \le \sum_{i=k+1}^{n} \frac{i}{i-k} = \sum_{j=0}^{n-k-1} \frac{j+k+1}{j+1} \le (k+1)\cdot H_n + n \tag{8}$$

Above, $H_n = \Sigma_{i=1}^{n} i^{-1}$ is the n-th harmonic number. By combining this with (7), and by applying the identities $\Sigma_{k=0}^{n}\binom{n}{k}P^{n-k}(1-P)^k k = n(1-P)$ and

$\Sigma_{k=0}^{n}\binom{n}{k}P^{n-k}(1-P)^k = 1$, we get the following upper bound for $\overline{N}$:

$$\overline{N} \leq 1 + \varphi_0^{-1} \cdot (1 + \frac{1}{2+\varphi_0^{-1}}) \cdot n \cdot (1-P) + n \cdot H_n \cdot (1-P) + n \qquad (9)$$

The bound in (9) is too conservative – in particular, the term $n \cdot H_n$ never exceeds $n \cdot \varphi_0^{-1}$ (in the worst case, the sampling frequency stays φ_0), so we can replace it with $n \cdot min(\varphi_0^{-1}, H_n)$. This proves the first part. To prove the second part, we find a lower bound for $\overline{N}$ – we consider the path in the probability tree that starts with $n-k$ hits (which set φ to 1), followed by k misses:

$$\overline{N} \geq 1 + \sum_{k=0}^{n}\binom{n}{k}P^{n-k}(1-P)^k\Big((n-k)\cdot 1 + \sum_{i=n-k+1}^{n} min(\varphi_0^{-1}, \frac{(1-\frac{n-k}{i})^2}{\frac{n-k}{i}})\Big) \qquad (10)$$

The term $n-k$ becomes $n \cdot P$ under the outer sum. The second term (the inner sum) consists of two parts, depending on which part under the *min* dominates. The part with φ_0 is alone greater than $k\varphi_0^{-1}$ when $P \to 0$. From this, it can be shown that $\overline{N}$ is lower bound by $n \cdot P + \varphi_0^{-1} \cdot n \cdot (1-P)$. When work tends to 0, the speedup $\overline{S} = \overline{T}_{base}/\overline{T}_{sampling}$ becomes the ratio between the time spent in context switching without sampling and the time spent with sampling. Note that sampling spends $n \cdot c$ extra time, but only in the $1-P$ cases that do not end in a speculation hit. By substituting the lower bound into the speedup:

$$\overline{S} \geq \frac{\overline{N} \cdot c}{\overline{N} \cdot c + n \cdot c \cdot (1-P)} \geq 1 - \frac{1-P}{1+\varphi_0^{-1} \cdot (1-P)} \qquad (11)$$

The last expression in (11) is minimal when $P = 0$, and the claim follows. $\square$

We interpret the Theorem 4 as follows. First, when $P \to 1$, the term with the initial frequency φ_0 disappears, and the expected number of messages $\overline{N}$ depends only on the sample size n. From (9) and (10), for $P = 0.9$, $n = 43$ and $\varphi_0 = 0.01$, the expected number of messages $\overline{N}$ is between 468 and 497, an $8\times$ improvement. Second, when $P \to 0$, the sampling overhead depends only on the initial sampling frequency φ_0. If we pick an unreasonably high value $\varphi_0 \to 1$ for the initial sampling frequency, the performance degrades by at most 50% – this is the case when we always sample after receiving a message, without benefiting from speculation hits, and effectively paying the context switch cost twice.

3 Algorithm and Implementation

We can summarize the results from Sect. 2 as follows. When the speculation hit probability P is greater than the relative delay δ, where $\delta = d/c$ is the ratio between the absolute delay $d \in \langle 0, c\rangle$ and the context switch time c, an actor must speculatively delay its context switches by the duration d. For a confidence level α, an actor must gather a sample of speculation hits of size $n = z_{1-\alpha/2}^2/(4e^2)$, where e is the confidence interval for the sampled speculation hit probability p. These n values are sampled with some probability φ, which is a small value φ_0 initially, but can be set to $\varphi = (\overline{p}_i - \delta)/(1-\overline{p}_i)^2$ after every sampling iteration.

```
 1 global $\varphi$ = $\varphi_0$
 2 global L = 32
 3 global counts = [1..L]
 4 global d_best = 0
 5 global sample_count = 0
 6
 7 has_more = poll()
 8 while has_more:
 9   has_more = false
10   if drain():
11     spins = d_best
12     while spins > 0:
13       spins -= 1
14       if spins%(C/L) == 0:
15         has_more = poll()
16       if has_more:
17         spins = 0
18 if random(0.0, 1.0) < $\varphi$:
19   spins = 0, i = 0
20   while spins < C:
21     spins += 1
22     if spins%(C/L) == 0:
23       has_more = poll()
24       i += 1
25     if has_more:
26       counts[j $\in$ i..L] += 1
27       spins = C
28   sample_count += 1
29   $\varphi$ = max($\varphi_0$, calc_p())
30 if sample_count == n:
31   sample_count = 0
32   k = argmin(counts[i]-C/L*i)
33   d_best = C/L*k
34   counts[i $\in$ 1..L] = 0
```

Fig. 3. Pseudocode for speculative lagging

The algorithm in Fig. 3 collects a set of sampled probabilities $\overline{p}_i$ for equidistant delays $d_i \in \langle 0, c\rangle$. An actor runs the algorithm immediately before each context switch. The algorithm maintains the current best delay `d_best`, initially zero, and the array `counts` of speculation hit counts for each d_i. It first checks for messages with `poll` in line 7, and handles them by calling `drain` in line 10. The `drain` method returns `false` only when the scheduler externally disallows further execution – in this case, the actor must immediately yield. Otherwise, the actor spins for `d_best` time units, and calls `poll` on the message queue every `C / L` time units, where `L` is the total number of delays d_i, and `C` is the context switch time. If the actor finds that the message is available, it calls `drain` again and this process is repeated. After the loop in line 8 ends, the actor samples the delays, with the probability φ, by finding the first delay d_i after which a method is available, and updates the speculation hit counts accordingly in line 26. The actor adapts the sampling frequency φ in line 29. Upon reaching the sample size `n`, the actor sets `d_best` to the d_i with the largest value $p_i - d_i$ in line 33, and then resets the speculation hit counts in line 34.

We implemented our algorithm in the Reactors framework [2] [19], as a modification of the pluggable scheduling system in Reactors [18]. Context switch in this framework consists of finding a worker thread, creating a task object, interacting with the work queue, and setting up actor-local state on the worker. The largest deviation from the analysis in Section 2 is that the number of messages a reactor can process is upper bound, and kept around 50 – this already amortizes the context switch times, but ensures fair scheduling (i.e. a bound on latency).

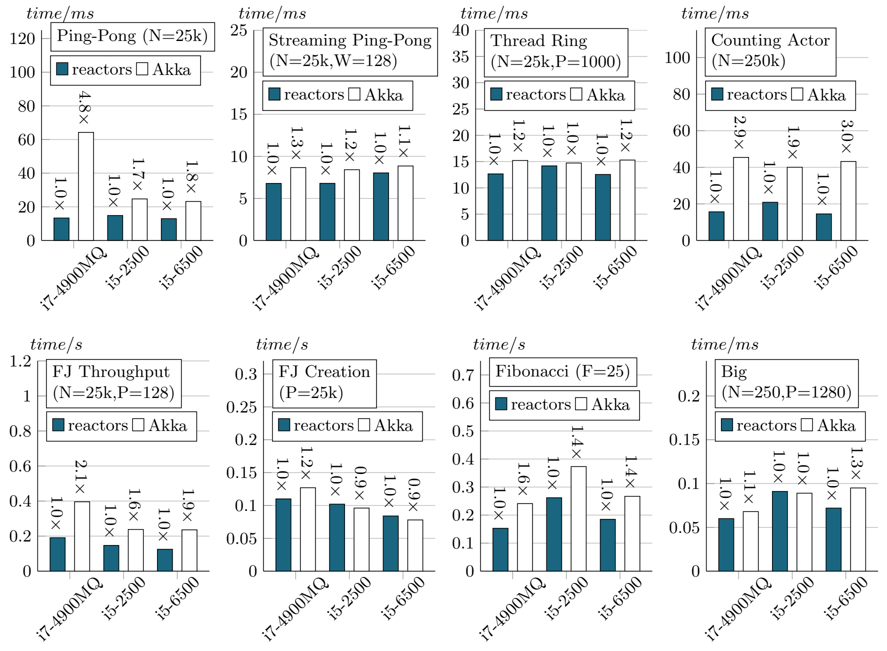


Fig. 4. Running time comparison between Reactors and Akka on the Savina benchmark suite (lower better; N – number of messages, P – number of actors, W – streaming window size, F – computed Fibonacci number)

4 Evaluation

In this section, we (1) show the running times of the benchmarks from the Savina actor suite [12], using three different processor models. We identify a subset of benchmarks on which speculative lagging improves performance, and use them to estimate the context switch time c. We then (2) study the performance of these benchmarks for a different number of actors in the system. Finally, to validate that speculative lagging does not degrade performance when it is not beneficial, we (3) identify a subset of benchmarks for which speculative lagging is potentially harmful, and compare the running time for different initial sampling frequencies φ_0. We use standard performance evaluation techniques [10].

We emphasize that we only expect to see a performance improvement on specific benchmarks, in which a majority of actors normally spend a lot of time context switching compared to doing useful work. On other benchmarks, where the actors' mailboxes are saturated and the context switches are rare compared to the amount of useful work, it is unreasonable to expect a speedup – here, eliminating the context switch is unnoticeable by definition. However, we require that the addition of our speculation algorithm *does not degrade performance on any benchmark* – in the worst case, the performance must stay the same.

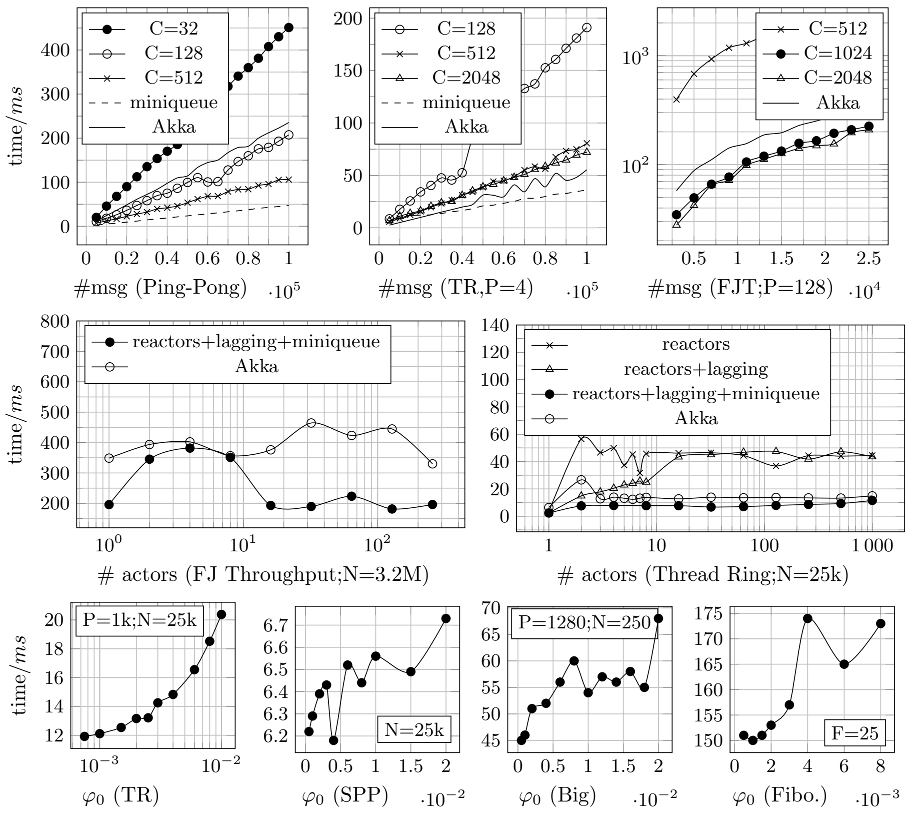


Fig. 5. First Row – Impact of Estimated Context Switch Time C (i7-4900MQ); Second Row – Impact of Parallelism Level P (i7-4900MQ; $C = 2048$); Third Row – Impact of Initial Frequency φ_0 (i7-4900MQ; $C = 2048$; note that smallest φ_0 is 0.1%)

In Fig. 4, we compare the running time between Reactors with speculative lagging and the Akka framework [1] on the Savina benchmark suite, using three different processors, quad-core 2.8 GHz i7-4900MQ with hyperthreading, quad-core 3.2 GHz i7-6500, and quad-core 3.3 GHz i5-2500. We distinguish between benchmarks for which speculative lagging is beneficial, namely, *Ping-Pong*, *Thread Ring* and *Fork-Join Throughput*, and the remaining benchmarks on which lagging does not improve performance. We note that Reactors is $2-3\times$ faster on the *Counting Actor* benchmark due to primitive type specialization optimization [7], which is only used in that particular benchmark.

In the first row of Fig. 5, we compare the running times for different values of the estimated context switch time C. Here, C is expressed as the number of spins in the algorithm from Fig. 3. In *Ping-Pong*, two actors repetitively exchange N messages, where N is shown on the x-axis. For $C = 128$, the running time is slightly below the Akka version, and converges after C reaches 512. In *Thread Ring*, a total of $P = 4$ actors form a ring, and send a message N/P times around, and C also converges after 512. In both these benchmarks, the running time is additionally improved by keeping an actor-local 1-element message *miniqueue*, which the actor can steal a message from to avoid context switching. The miniqueue optimization has no effect on *FJ Throughput*, where a single producer sends messages to $P = 128$ consumer actors in a roundabout manner. Here, speculative delays cause messages to pile up at non-active actors, which decreases the overall number of context switches, and improves performance by up to $10\times$ for $C > 1024$ (note the logarithmic y-axis).

In the second row in Fig. 5, we analyze the performance of *FJ Throughput* by keeping the total number of delivered messages fixed at 3.2 million, and changing the number of consumers from 1 to 256. The speculation benefits are less pronounced when there are 2 to 8 consumers, which coincides with the number of hardware threads in i7-4900MQ – when each consumer is assigned to a CPU core, speculative delays are just long enough to let the producer run a full cycle, but messages cannot pile up. The situation is reversed on *Thread Ring*, where performance is improved only when ring size is 8 or less. Since there is a single message passed around, speculative delays only help when each actor can be pinned to a CPU. For $P > 8$, Akka is noticeably faster when miniqueues are disabled in Reactors, since Akka's scheduler runs the next actor directly.

In *Thread Ring* (when $P \gg 8$), *Big* and *Fibonacci*, each actor receives only very few messages in total, so speculative delays can only slow down the program. We vary initial sampling frequency φ_0 in the third row of Fig. 5, and find that the optimal value is $\varphi_0 \leq 0.2\%$ (**we note that we used $\varphi_0 = 0.2\%$ and $C = 2048$ for the benchmarks in Fig. 4**). For comparison, actors in *Streaming Ping-Pong* (SPP) always have a large number of messages waiting in the mailbox, so that benchmark is insensitive to the sampling frequency.

5 Related Work

Speculation is in practice frequently used to improve execution performance. Many compiler optimizations speculate on program sections that execute less frequently [8], and optimistic concurrency control is based on the bet that synchronization can be omitted [13]. CPUs speculatively execute instructions out of order, and speculatively eliding locks improves performance in some cases [15]. In the context of cluster computing, delaying the start of a job can improve throughput [21]. To cope with straggler jobs, some cluster runtimes speculatively execute them in parallel [3].

In the context of concurrent computing, existing related work can be separated into two groups. The first group consists of various spin lock techniques, and was studied extensively [16]. When acquiring a spin lock, a process repetitively polls the availability of a critical section until another process releases the spin lock. Spin locks were used in a variety of applications, from general-purpose locking [6] to concurrent data structures [4]. Spin locks are similar to speculative lagging, proposed in this work, but differ in several crucial ways. First, in speculative lagging, the release of a computing resource is preceded by spinning, whereas with spin locks, a process spins prior to acquiring a lock. Second, after acquiring a spin lock, a process may attempt to acquire other locks, potentially entering a deadlock. In speculative lagging, the wait time is bound, and cannot lead to a deadlock. Third, spin locks are exposed as a programming primitive, while speculative lagging is performed transparently by a scheduler.

The second group deals with speculatively applying optimizations to concurrent programs. Optimistic concurrency techniques [13] speculate that it is more efficient to run a computation without synchronization, and pay the price only occasionally, compared to always synchronizing concurrent processes. Optimistic concurrency is used in databases and in software transactional memory [11]. Techniques for eliding locks in multithreaded programs were proposed in the past both as microarchitectural solutions [20] and as compiler techniques [15]. Some lock implementations speculatively assume that there are no concurrently executing modifications, and validate memory reads by only reading the lock state [17], which is cheaper compared to writes. Just-in-time compilation techniques make statistical assumption about the program behaviour, and deoptimize the program to a slower variant if an assumption is broken [8]. Common to all of these techniques is the idea of *speculation* – running a simpler, cheaper version of the execution, and potentially reverting to the more costly implementation if it turns out that the assumption is invalidated.

Speculation is often applied blindly, but was in some cases guided by a statistical model. For example, a statistical model was used in the past to predict which threads are more likely to acquire a lock next [14].

We are unaware of prior work on speculatively delaying context switches in actors and other message-based systems, and believe that this is the first contribution in this area.

6 Conclusion

We proposed a new technique for scheduling message-based programs, called *speculative lagging*. The speculation is based on the bet that waiting for another message is less costly than context-switching to another process. To correctly detect speculation opportunities at runtime, we proposed a statistical sampling model that predicts whether delaying a context switch is beneficial. The sampling is adaptive – when it believes that lagging helps, the algorithm increases the sampling rate, hence reaching the conclusion faster. We showed the bounds for the expected speedup, and derived their proofs. When applied to an existing actor system, our speculative lagging algorithm improves performance on benchmarks that spend considerable time in context switches. We experimentally identified the standard benchmarks in which speculative lagging improves performance, and we showed that performance is otherwise not degraded.

Our conclusion is that speculative lagging improves program performance by reducing the amount of context switching. The sampling overhead of detecting speculation opportunities at runtime can be made arbitrarily small with the correct choice of initial parameters. We found that the initial sampling frequency

$\varphi_0 = 0.2\%$ and the maximum spin count $C = 2048$ work well, but these numbers may need to be tuned on a per-system basis. Thus, speculation lagging *does not degrade* the performance of programs that cannot benefit from context switches, and it improves the overall throughput otherwise.

References

1. Akka documentation (2017), http://akka.io/docs/
2. Reactors.IO website (2017), http://reactors.io/
3. Ananthanarayanan, G., Hung, M.C.C., Ren, X., Stoica, I., Wierman, A., Yu, M.: Grass: Trimming stragglers in approximation analytics. NSDI 2014 (2014)
4. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. PPoPP '10, ACM, New York, NY, USA (2010)
5. Cochran, W.: Sampling Techniques. Wiley (1977)
6. Dice, D.: Biased Locking in HotSpot
7. Dragos, I., Odersky, M.: Compiling Generics Through User-Directed Type Specialization. ICOOOLPS '09 (2009)
8. Duboscq, G., Würthinger, T., Stadler, L., Wimmer, C., Simon, D., Mössenböck, H.: An intermediate representation for speculative optimizations in a dynamic compiler. VMIL '13 (2013)
9. Fleiss, J.L.: The Teacher's Corner: A Note on the Expectation of the Reciprocal of a Random Variable (1966)
10. Georges, A., Buytaert, D., Eeckhout, L.: Statistically Rigorous Java Performance Evaluation. SIGPLAN Not. (2007)
11. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. SIGARCH Comput. Archit. News 21(2) (May 1993)
12. Imam, S.M., Sarkar, V.: Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. AGERE! '14 (2014)
13. Kung, H.T., Robinson, J.T.: On Optimistic Methods for Concurrency Control. ACM Trans. Database Syst. (1981)
14. Lucia, B., Devietti, J., Bergan, T., Ceze, L., Grossman, D.: Lock prediction. In: Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism (2010)
15. Martínez, J.F., Torrellas, J.: Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. SIGOPS Oper. Syst. Rev. (2002)
16. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. 9(1) (Feb 1991)
17. Nakaike, T., Michael, M.M.: Lock elision for read-only critical sections in java. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '10, ACM, New York, NY, USA (2010)
18. Prokopec, A.: Pluggable Scheduling for the Reactor Programming Model. AGERE 2016 (2016)
19. Prokopec, A., Odersky, M.: Isolates, channels, and event streams for composable distributed programming. Onward! 2015 (2015)
20. Rajwar, R., Goodman, J.R.: Speculative lock elision: Enabling highly concurrent multithreaded execution. In: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture. MICRO 34, IEEE Computer Society, Washington, DC, USA (2001)
21. Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., Stoica, I.: Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. EuroSys '10 (2010)