Extending BasicPOS for Weak Memory Models

Roberto Gheda, Niyousha Najmaei, Yifan Song

Abstract

In this project, we extend the BasicPOS algorithm to weak memory programs by implementing it on top of C11Tester, the popular weak memory concurrency testing framework. We evaluate the BasicPOS for weak memory algorithm on C11Tester benchmarks and compare the performance with C11Tester and the PCT for weak memory algorithm. Our results show that BasicPOS for weak memory detects bugs more frequently than C11Tester. However, PCTWM outperforms it in the majority of the benchmarks. This motivates extending the POS algorithm to weak memory models, for which we present a draft of our suggested approach.

1 Introduction

The challenge of detecting concurrency bugs to avoid programs with unexpected behaviour has been addressed by the proposal of various verification techniques and testing tools. Although verification techniques guarantee soundness, they scale poorly [1]. This motivates the need for concurrency testing tools that guarantee a high probability of detecting bugs while scaling better.

C11Tester, a race detector for the C/C++ memory model [2], is an example of these testing tools. Despite the promising guarantees of C11Tester for detecting data races, [1] has shown that there is room for improvement in the C11Tester framework by implementing a more effective thread scheduling algorithm than random scheduling of threads. [1] implements the Probabilistic Concurrency Testing (PCT) algorithm, [3], on top of C11Tester and extends it to PCT for Weak Memory (PCTWM). Their results show that PCTWM can detect bugs more frequently than C11Tester.

In this project, we investigate implementing the Basic Partial Order Aware Concurrency Sampling (BasicPOS) algorithm on top of C11Tester. BasicPOS is a sampling-based algorithm with probabilistic theoretical guarantees designed to test concurrent and distributed programs under Sequential Consistency (SC) [4]. In order to extend this algorithm to weak memory models, we implemented it on top of the C11Tester framework and compared the performance in terms of bug-hitting rate to C11Tester and PCTWM. Moreover, we present some of our ideas and suggestions about extending the Partial Order Aware Concurrency Sampling (POS) algorithm, [4], for weak memory models.

The rest of the report is structured as follows. In Section 2, we briefly describe the C11Tester weak memory axiomatic model as well as the concurrency testing algorithms used in this project, namely BasicPOS, POS, PCT and PCTWM. Moreover, we compare the probabilities of detecting bugs with these algorithms while discussing two particular examples. In Section 3, we discuss our approach to extending BasicPOS to weak memory, and we introduce the BasicPOS for weak memory algorithm. Section 4 addresses our experimental evaluation results and compares the performance of BasicPOS for weak memory on C11Tester benchmarks to that of C11Tester and PCTWM. Moreover, we investigate the effect of the history parameter, h, on the algorithm's performance. Finally, we close with Section 5, where we provide a discussion of the results and our outlook on this project, as well as the limitations of the work and possible future work directions.

2 Background

We start by introducing some basic notions and explanations of algorithms that inspired the design of BasicPOSWM. In this section, we briefly introduce weak memory models, C11Tester axiomatic model and BasicPOS, POS, PCT and PCTWM algorithms.

2.1 Weak Memory Models

Under SC, threads must read from the latest write assigned to a variable, regardless of what thread did such write. Under weak memory, such a constraint doesn't hold; this means that a read event may read from a write event of a different thread which is not the latest write made in the program. This is why weak memory model generates several bugs that do not appear under SC. Figure 1 shows an example of a program that does not generate bugs under SC but does so under weak memory.

$$
\begin{array}{c}
X = Y = 0 \\
X = 1; \;||\; Y = 1; \\
a = Y; \;||\; b = X; \\
\texttt{assert}(a == 1 \;||\; b == 1)
\end{array}
$$

Fig. 1: An example of a program that doesn't generate bugs under sequential consistency but does under weak memory model.

2.1.1 C11 Axiomatic Model

For this project, we are following the C11 axiomatic model presented in depth in the C11Tester technical report [2].
Firstly, we start introducing some notation used from now on.

- We denote **E** to be the set of events of the program.
- We denote **SC** to be the total order over sequential consistency accesses.
- We define **rf** (reads-from) the relation of a write event with the read event that reads from it.
- We define **mo** (modification order) to be a strict total order over same-location write events.
- We define **fr** (from-read) relates a same-location read and write events. Meaning that if r reads from a write w and w' is mo-after w, then $\texttt{fr}(r, w' \setminus [E])$ holds. Formally: $\texttt{fr} \triangleq (\texttt{rf}^{-1}; \texttt{mo})$
- Eventually, we adopt the **sw** synchronize-with relation from RC20 [5]. And define the **hb** (happened-before) relation as follows:
  - $\texttt{sw} \triangleq [E_{\sqsupseteq REL}]; ([F], \texttt{po})^{?}; \texttt{rf}^{+}; (\texttt{po}; [F])^{?}; [E_{\sqsupseteq ACQ}])$
  - $\texttt{hb} \triangleq (\texttt{po} \cup \texttt{sw}^{+})$

The set of axioms defined by C11 are as follows:

- **Coherence**: The events accessing the same memory location are coherent. We categorize them in write and read coherence constraints. Meaning:
  - $\texttt{mo}; \texttt{rf}^{?}; \texttt{hb}^{?}$ `is irreflexive`
  - $\texttt{fr}; \texttt{rf}^{?}; \texttt{hb}$ `is irreflexive`
- **Atomicity**: The RMW accesses execute atomically. As a result, $(\texttt{fr}; \texttt{mo}) = \emptyset$ holds.
- **irrMOSC**: The mo and SC orders agree on same-location accesses, that, (mo ; SC) is irreflexive.
- **SC**: The SC accesses are globally ordered, meaning that $(\texttt{hb} \cup \texttt{rf} \cup \texttt{SC})$ is acyclic.

Therefore, we define an execution to sample from as a tuple:

$$X = \langle E, \texttt{po}, \texttt{rf}, \texttt{mo}, \texttt{SC} \rangle$$

2.2 BasicPOS and POS

2.2.1 Partial Order Sampling Notations and Definitions

In partial order sampling algorithms, we define: $\texttt{En}(t)$ to be the set of enabled events given a trace t. Moreover, we use notation $t \simeq \mathcal{E}$ to say a trace is following a partial order $\mathcal{E}$, meaning:

$$t \simeq \mathcal{E} \iff e_0 \sqsubset_t e_1 \Rightarrow e_0 \sqsubset_{\mathcal{E}} e_1 \ \forall e_0, e_1$$

Furthermore, we use notation $t \models \mathcal{E}$ to say a trace t is the partial order $\mathcal{E}$, meaning:

$$t \simeq \mathcal{E} \iff e_0 \sqsubset_t e_1 \iff e_0 \sqsubset_{\mathcal{E}} e_1 \ \forall e_0, e_1$$

Eventually, we use notation $e \bowtie_{\mathcal{E}} e' \coloneqq \exists t, t \simeq \mathcal{E} \wedge \{e, e'\} \subseteq \texttt{En}(t)$, meaning e and e' are simultaneously enabled under partial order $\mathcal{E}$. And

$$\texttt{PS}_{\mathcal{E}}(e) \coloneqq \{e' | e' \sqsubset_{\mathcal{E}} e \wedge e \bowtie_{\mathcal{E}} e'\}$$

Both BasicPOS and POS are strictly based on the lemma that, given a trace $t \simeq \mathcal{E}$ and $\texttt{En}(t) \neq \emptyset$, then it always exists an event e such that $t \bullet e \simeq \mathcal{E}$. Such lemma is properly proved in the original paper [4].

2.2.2 Algorithms

The main idea behind the POS algorithm is to give all possible partial orders the same probability of being sampled.

In BasicPOS, every event is assigned a random priority taken from a uniform distribution $\mathcal{U}(0, 1)$. The major weakness of BasicPOS is the constraint propagation of priorities. Meaning an event e with a large $\texttt{PS}(e)$ set may have a relatively low priority since its priority has to be lower than events in $\texttt{PS}(e)$. Thus, for any event $e' | e \bowtie_{\mathcal{E}} e'$ that has to be delayed, e''s priority has to be even smaller than e's one, which is unnecessarily hard to satisfy for a value taken from a random distribution. This probability can be as low as $\frac{1}{V!}$.

POS is a further expansion of BasicPOS that removes the constraint of priorities issue by regularly reassigning priority values. The idea behind POS is to reset the priority of e right after scheduling e' whenever the following holds:

$$t \simeq \mathcal{E} \wedge \{e, e'\} \subseteq \texttt{En}(t) \wedge t \bullet e \not\simeq \mathcal{E} \wedge t \bullet e' \bullet e \simeq \mathcal{E}$$

However, we don't know how to reset such values since $\mathcal{E}$ is unknown during the sampling. The extended version of the paper proves that we need to reset the priorities of all events racing with the one that was last scheduled [6].
The paper also gives a probabilistic guarantee of POS performance on general programs of:

$$p_{POS} = \frac{1}{\mathcal{P}^{\mathcal{N}}} \left(\frac{1}{1 - (1 - \frac{1}{\mathcal{P}}\alpha)} \right)^{\alpha \mathcal{N}/2}$$

Where $\mathcal{P}$ is the number of processes, $\mathcal{N}$ the number of events and $0 \leq \alpha \leq 1$ such that $\alpha\mathcal{N}$ is the number of non-racing events [4].

2.3 PCT and PCTWM

Probabilistic Concurrency Testing (PCT) is a randomized concurrency testing algorithm designed for SC programs that provide strong theoretical guarantees on the probability of detecting concurrency bugs [3].
The key notion behind the PCT algorithm is concurrency bug depth, defined as the number of ordering constraints between the concurrent events of a program. PCT samples an execution from the set of executions with d ordering constraints. It guarantees a lower bound on the probability of detecting a bug by a test execution with a probability of at least

$$p_{PCT} = \frac{1}{(tk^{d-1})}$$

where t is the number of threads, k is the number of program events, and d is the bug depth.

The PCTWM algorithm, which extends the PCT algorithm to weak memory, randomly generates a test execution with h-bounded d communication relations between the program events, where d is the minimum number of communication relations between the concurrent events in an execution that is sufficient to produce the bug, and h is the history depth. This means that h bounds the behaviour of a read event to the last h write actions on a specific object.
The paper then also gives a probability bound of detecting bugs of

$$p_{PCTWM} = \frac{1}{O((hk_{com})^d)}$$

which exponentially decreases with respect to d [1].

2.4 Examples

2.4.1 Example 1: POS performing better than PCT

```
global int x = y = z = w = 0;

    Thread A              Thread B
------------------- --------------------
                    local int a = b = 0;
A1: x++;            B1: x = 1;
A2: y++;            B2: a = x;
A3: signal(w);      B3: y = a;
A4: assert(z < 5);  B4: wait(w);
                    B5: b = y;
                    B6: z = a + b;
```

Fig. 2: POS hits the bug with probability $\frac{1}{48}$, while PCT does so with probability $\frac{1}{200}$ (Ref. [4]).

Figure 2 shows an example of a program for which POS outperforms PCT. In this example, which is taken from the POS paper [4], to fail assertion `A4` under SC, this specific trace needs to be executed:

$$\texttt{B1} \rightarrow \texttt{A1} \rightarrow \texttt{B2} \rightarrow \texttt{B3} \rightarrow \texttt{A2} \rightarrow \texttt{A3} \rightarrow \texttt{B4} \rightarrow \texttt{B5} \rightarrow \texttt{B6} \rightarrow \texttt{A4}$$

Random walk has to make the correct choice at every step. Among all ten steps, three only have a single option: A2 and A3 must be executed first to enable B4, and A4 is the only statement left at the last step. Thus, the probability of reaching the bug is $1/27 = 1/128$. As for PCT, we have

to insert two preemption points just before statements B2 and A2 among ten statements, thus the probability of hitting the bug $1/10 \times 1/10 \times 1/2 = 1/200$ for PCT, where the $1/2$ factor comes from the requirement that thread B has to be executed first.
In POS, this bug can be detected with a substantially high probability of $1/48$, much higher than other approaches. The formal guarantees of [4] ensure that any behaviour of this program can be covered with a probability of at least $1/60$.

2.4.2 Example 2: BasicPOS performing better than POS

$$\begin{array}{l} X = 1; \\ X = 2; \\ \ldots; \\ X = k \end{array} \Bigg\| \; \texttt{assert}(X! = k)$$

Fig. 3: In this example, BasicPOS hits the bug with probability $\frac{1}{(k+1)}$. Much better than the odds of POS, $\frac{1}{k!}$.

Figure 3 shows an example program for which BasicPOS may perform better than POS for some specific scenarios. In this example, we have two processes. The first executes k write events over the variable X, while the second executes only one read event on the variable. Clearly, the only way to trigger the assert is to execute the event of the second process as the last one. We now start deriving probabilities for Random Walk, BasicPOS and POS.
In Random Walk, the probability of hitting such a bug is $\frac{1}{2^k}$ since we have to decide between scheduling from the two threads for k times. With BasicPOS, on the other hand, the probability of running this execution is only $\frac{1}{(k+1)}$, which is a significant improvement from the Random Walk performance. POS eventually degenerates into the Random Walk behaviour. This happens since, for all events in the program, we access X; this means that all events are in a data race with each other, meaning we update priorities after every scheduling.

3 Implementation

In this project, we implemented the POS and BasicPOS scheduling algorithms instead of C11Tester's random thread scheduling for detecting weak memory bugs. Moreover, we extended the BasicPOS algorithm for weak memory programs for more effective detection of buggy executions in weak memory programs. The link to our GitHub repository is available in Appendix A.

The following section briefly describes our approach and the BasicPOS for weak memory algorithm. Moreover, we discuss the probability of hitting weak memory bugs with BasicPOSWM using an example program.

3.1 BasicPOSWM: BasicPOS for Weak Memory

We adopt the concepts of *communication sink* and *view* from PCTWM [1]. The weak memory algorithm relies on two key concepts; (i) communication relation between concurrent program events; and (ii) local thread view. Their formal definitions are as follows.

- *View*. A view is a map of a location to a list of maximal-mo events on that location, denoted as $view(x)$.
- *Combine Views*. We define the combination of two views $\bigsqcup_{mo}(view_1(x), view_2(x))$ as $maximal(view_1(x), view_2(x), mo)$.

• *Communication Sinks.* A communication sink event is an SC, *read* or *acquire* event.

3.2 The BasicPOSWM Algorithm

Algorithm 1 BasicPOS for Weak Memory

Require: h
  function BasicPOSWM(h)
    for $e \in E$ **do**
      $\texttt{Pri}(e) \leftarrow \mathcal{U}(0,1)$
    end for
    $t \leftarrow []$
    while $\texttt{En}(t) \neq \emptyset$ **do**
      $e^* \leftarrow \arg\max_{e \in \texttt{En}(t)} \texttt{Pri}(e)$
      $r \leftarrow \texttt{ExecuteAndUpdate}(t, e, h)$
      $t \leftarrow t \bullet e^*$
    end while
    return t
  end function

Algorithm 2 Execute the Event and Update View

Require: h
  function ExecuteAndUpdate(t, h, e)
    $x \leftarrow e.loc$
    if $e \in Write$ **then**
      $t.view(x) \leftarrow e$
    end if
    if $e \in Read$ **then**
      $e_{rf} \leftarrow readGlobal(t, h, e)$
      if $isSync(e, e_{rf})$ **then**
        $t.view(x) \leftarrow \bigsqcup_{mo}(t.view, e_{rf}.bag)$
      else
        $t.view(x) \leftarrow \bigsqcup_{mo}(t.view(x), e_{rf}.bag(x))$
      end if
    end if
    if $e \in SC$ **then**
      $e_{source} \leftarrow getSC(t, h, e)$
      $t.view \leftarrow \bigsqcup_{mo}(t.view, e_{source}.bag)$
    end if
    if $e \in F_{ACQ}$ **then**
      $e_{source} \leftarrow getSW(t, h, e)$
      for $e' \in e_{source}$ **do**
        $t.view \leftarrow \bigsqcup_{mo}(t.view, e'.bag)$
      end for
    end if
    $e.bag \leftarrow t.view$
    $execute(e)$
  end function

Our BasicPOSWM algorithm extends the functionality of BasicPOS to weak memory and generates an execution with an $h-$bounded communication relation between the program events

across threads. The process can be divided into the scheduling and the execution processes, which are depicted in Algorithms 1 and 2, respectively.

Function BasicPOSWM. The scheduling part of our algorithm is the same as the scheduling algorithm introduced in BasicPOS. This algorithm initially assigns a random priority to all events from a uniform distribution $\mathcal{U}(0, 1)$. The scheduler, then, chooses the thread for which the pending event has the highest priority within the enabled set $\texttt{En}(t)$. Depending on the type of the pending event, the selected thread would be deeded into the execution portion for view updates.

Function ExecuteAndUpdate. To extend the BasicPOS algorithm to weak memory programs, we need to extend the execution process from the SC version of the algorithm, bounding the probability of detecting a bug in a fixed parameter tractable fashion. In the update view process, the scheduled event can read from any write-event from the past within a history depth bound h. Choosing the target write event happens in the **readGlobal** function. Similarly to what has been done for PCTWM, we define h to be a bound on readings so that a read-event cannot read from a write-event that has more than h successors in the modification order, which is a strict total order over same-location write events. We also need to update the thread view accordingly with the event's bag for other types of communication sink events.

3.3 Probabilistic Guarantees

As discussed in Section 2, under sequential consistency, the BasicPOS algorithm hits the bug in the program of Figure 3 with probability $\frac{1}{k+1}$. BasicPOS for weak memory hits this bug under weak memory with probability $\frac{1}{O(kh)}$, where h is the history parameter. In general, the probability of hitting the bug with BasicPOS under sequential consistency is degraded by a factor of $\frac{1}{h^{k_{comm}}}$ for the case of BasicPOS for weak memory, where k_{comm} is the number of communication events present in the program. This is because the search space of the program is increased by a factor of $h^{k_{comm}}$ in the case of the weak memory algorithm.

4 Evaluation and Results

In this section we present our research questions, and the experimental evaluation that we carried out to answer them. We ran our implementation of BasicPOS for weak memory on C11tester benchmarks, and compared the achieved bug-hitting rate to that of C11tester itself as well as PCTWM and our implementation of POS and BasicPOS.

4.1 Research Questions

Our research questions regarding the evaluation of BasicPOS for weak memory are as follows.

- Can BasicPOS be extended to detect concurrency bugs under weak memory model?
- How does the history, h, affect the performance?
- How does the bug-hitting rate of BasicPOS for weak memory compare to that of BasicPOS and POS?
- How does the bug-hitting rate of BasicPOS for weak memory compare to that of C11tester and PCTWM?

4.2 Experiments and Results

In order to answer our research questions, we ran POS, BasicPOS and BasicPOS for weak memory algorithms on seven C11Tester data structure benchmarks that were originally used to check CDSChecker [7]. We used the version of the tests that can be found in C11Tester's vagrant repository [1], which contain seeded weak memory bugs. [1] provides a description of these benchmarks in terms of lines of code (LOC), estimated number of event (k), number of communication events (k_{comm}) and bug-depth (d) which is included in this report, Figure 4, for the sake of completeness.

[1] C11Tester's vagrant repository is available at https://doi.org/10.1145/

Benchmark	LOC	k	k_{com}	d
dekker	50	20	14	0
msqueue	232	49	31	0
barrier	38	15	10	1
cldeque	122	86	56	1
mcslock	75	26	16	1
mpmcqueue	108	19	17	2
linuxrwlocks	90	20	19	2
rwlock	98	84	74	2
seqlock	50	20	18	3

Fig. 4: Description of C11Tester's data-structure benchmarks. (Ref. [1])

Similar to [2] and [1], we use the bug-hitting rate as a measure of the performance of algorithms, which is the percentage of executions that were detected to be buggy. Moreover, we investigated the effect of changing the history variable on the performance of BasicPOS for weak memory.

In the following, we present the results of the experiments.

4.2.1 Comparison to C11Tester and PCTWM

Table 1: Best bug-hitting rates achieved by POS, BasicPOS and BasicPOS for weak memory compared to results from C11Tester and best results achieved by PCTWM.

	C11Tester	POS	BasicPOS	PCTWM	BasicPOSWM
barrier	76,7	76	77	78,7	86
cldeque	94,7	88	77	100	99
dekker	21,6	18	13	100	29
linuxrwlocks	86,1	80	51	100	93
mcs-lock	89,4	51	46	100	95
mpmc-queue	59,4	57	55	100	66
ms-queue	100	100	100	100	100

Table 2: The values for history corresponding to the highest observed bug-hitting rates in BasicPOS for weak memory

	bug-hitting rate	history
barrier	86	2
cldeque	99	4
dekker	29	3
linuxrwlocks	87	5
mcs-lock	95	4
mpmc-queue	66	6
ms-queue	100	1

The results of running all three implemented algorithms, namely POS, BasicPOS and BasicPOS for weak memory as well as the results of C11Tester and PCTWM are summarised in Table 1. This table presents the highest observed bug-hitting rates for each of the algorithms. The corresponding values for history that produce these results are listed in Table 2. It should be noted that the

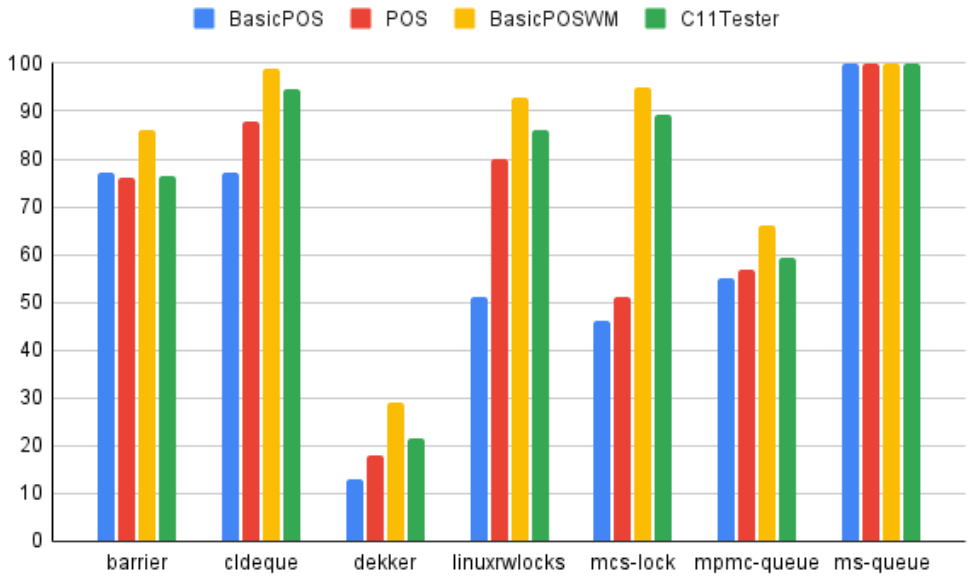

Fig. 5: The comparison of the highest observed bug-hitting rates in 500 executions for C11Tester and POS-based algorithms, POS, BasicPOS and BasicPOS for weak memory. The history values associated with the results of BasicPOS for weak memory are listed in 2.

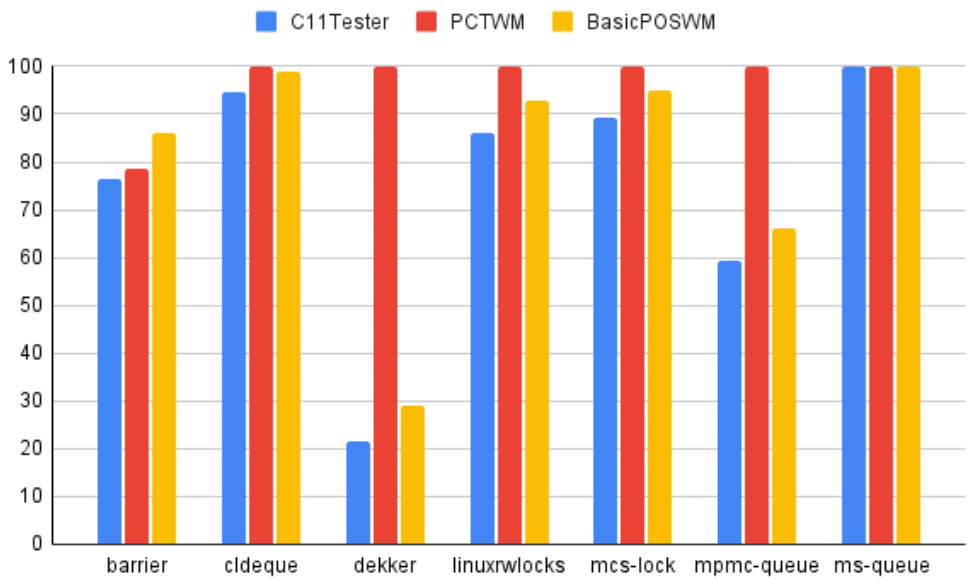

Fig. 6: The comparison of the highest observed bug-hitting rates in 500 executions for BasicPOS for weak memory, PCTWM and C11Tester. The history values associated with the results of BasicPOS for weak memory are listed in 2.

mentioned POS and BasicPOS algorithms are implemented on top of C11Tester by solely changing the scheduling algorithm. Thus, the remaining functionalities of C11Tester are unchanged and weak memory bugs can still be detected.

Figure 5 illustrates the comparison between the highest bug-hitting rates observed when running 500 executions of POS, BasicPOS, BasicPOS for weak memory and C11Tester. It is evident from this figure that BasicPOS for weak memory outperforms C11Tester's performance for all benchmarks. Moreover, it can be seen that POS performs better than BasicPOS, except on the barrier benchmark, where the difference in their performance is one per cent, which may be insignificant.

Figure 6 compares BasicPOS for weak memory, PCTWM and C11Tester. It can be seen from the figure that, even though BasicPOS for weak memory outperforms C11Tester in all the benchmarks, it does worse than PCTWM in all of them except for the barrier benchmark. Moreover, the bug-hitting rate achieved by BasicPOS for weak memory exceeds that of C11Tester by less than 10 per cent in all the benchmarks. This performance gain is relatively low compared to that of PCTWM.

4.2.2 The Effect of History

Table 3: Bug-hitting rates achieved by BasicPOS for weak memory for different values of the history variable

	$h = 1$	$h = 2$	$h = 3$	$h = 4$	$h = 5$	$h = 6$
barrier	75	86	75	73	74	72
cldeque	70	67	78	99	93	98
dekker	22	11	29	17	19	17
linuxrwlocks	56	56	51	84	87	80
mcs-lock	60	57	56	95	91	93
mpmc-queue	45	47	50	50	53	66
ms-queue	100	100	100	100	100	100

Table 3 shows the bug-hitting rates achieved by running BasicPOS for weak memory on the benchmarks for different values of history ranging from 1 to 6. Some of the benchmarks are affected more by variations in the history variable than others. For example, a significant increase in the observed bug-hitting rate can be seen in the mcs-lock, linuxrwlocks and cldeque benchmarks when history is increased to 4. In contrast, other benchmarks seem less affected by the change in the value of the history variable. As mentioned in [1], the effect of the history variable on the achieved bug-hitting rate heavily depends on the number of write events present in the program and how they are distributed over it. This could be a reason for our obtained results.

5 Discussion and Conclusion

5.1 Discussion of Results

As discussed in Section 4, BasicPOS for weak memory performs only slightly better than C11Tester. We believe this is because the probability of hitting bugs with the BasicPOS for weak memory algorithm degrades to that of random scheduling for programs with threads with approximately the same number of events. This is because the search space of BasicPOSWM for these types of programs grows to match that of random scheduling. Therefore, BasicPOS for weak memory is not performing significantly better than C11Tester.

Moreover, PCTWM outperforms BasicPOSWM in almost all benchmarks; this might be because of the propagation of priority constraints, which is an issue with the BasicPOS algorithm. This problem is addressed in the POS algorithm, where the priorities of the events in race with the last scheduled event are reassigned before the next scheduling. This motivates extending POS for weak memory models for possibly better performance results.

Regarding the comparison between BasicPOS for weak memory and BasicPOS and POS, the reason why BasicPOS for weak memory performs better than the other two algorithms is that in BasicPOS for weak memory, we have extended the execution of C11Tester for the read events to be able to read from a set of write events that are within the history bound. This is not the case

in our implementation of POS and BasicPOS. For those algorithms, we have only adapted the scheduling algorithm of C11Tester to that of POS and BasicPOS. However, C11Tester's execution algorithm allows hitting weak memory bugs, so some bugs can still be hit with our implementation of the POS and BasicPOS algorithms.

As our final outlook on this project, even though BasicPOS doesn't generally outperform PCTWM, it also comes with fewer tunable parameters. Therefore, BasicPOSWM may be a good compromise between performance and ease of use.

5.2 Limitations and Future Work

In this section, we discuss the limitations of this project and the future work they motivate.
First of all, BasicPOS for weak memory does not resolve the propagation of priority constraints, which is an issue inherited from BasicPOS [4]. Extending this algorithm to a novel POSWM algorithm will solve this issue, as POS doesn't have this problem; however, this requires a dependency definition suitable for weak memory models. Our draft proposal for such a definition is as follows. Given two events e_1 and e_2,

$$\begin{aligned} e_1 \perp e_2 \iff & \nexists e'.t \\ & \mathtt{rf}(e_1, e') \wedge e' \not\perp e_2 \vee \\ & \mathtt{rf}(e_2, e') \wedge e' \not\perp e_1 \end{aligned}$$

This definition is analogous to creating a directed dependency graph. Since we need to know when to update events' priorities, we need to define such a graph prior to scheduling, which is analogous to the problem of POS discussed in Section 2.

Another limitation of the algorithm is the bound on the search space given by h. Such a bound is required to avoid exploding the number of possible executions to search from. However, it prevents us from carrying out an exhaustive search of the space.

The third limitation of this project is that we only evaluated the performance of the algorithms based on the bug-hitting rate and did not consider execution time as a performance metric. As future work, we should compare the performance of BasicPOS for weak memory, C11Tester and PCTWM in terms of execution time.

In conclusion, we provided a novel algorithm that extends BasicPOS to weak memory models. This algorithm outperforms its sequential consistency counterparts and the standard C11Tester algorithm on C11Tester's data structure benchmarks.

Although BasicPOSWM has lower performance than PCTWM on most of the benchmarks, we must say that BasicPOSWM also comes with fewer tunable parameters. Therefore, BasicPOSWM may be a good compromise between performance and ease of use.

References

[1] Gao, M., Chakraborty, S., Ozkan, B.K.: Probabilistic concurrency testing for weak memory programs. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, pp. 603–616 (2023)

[2] Luo, W., Demsky, B.: C11tester: a race detector for c/c++ atomics. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 630–646 (2021)

[3] Burckhardt, S., Kothari, P., Musuvathi, M., Nagarakatte, S.: A randomized scheduler with probabilistic guarantees of finding bugs. ACM SIGARCH Computer Architecture News **38**(1), 167–178 (2010)

[4] Yuan, X., Yang, J., Gu, R.: Partial order aware concurrency sampling. In: Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic

Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30, pp. 317–335 (2018). Springer

[5] Margalit, R., Lahav, O.: Verifying observational robustness against a c11-style memory model. Proceedings of the ACM on Programming Languages **5**(POPL), 1–33 (2021)

[6] Yuan, X., Yang, J., Gu, R.: Partial order aware concurrency sampling. In: Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30, pp. 317–335 (2018). Springer

[7] Norris, B., Demsky, B.: CDSchecker. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. ACM, New York, NY, USA (2013)

A Code

Our code is available at the following GitHub repository.

https://github.com/r-gheda/c11tester

The BasicPOS for weak memory algorithm can be found on the basicposwm branch. The implementations of the BasicPOS and POS algorithms are available on basicpos and pos branches.