Lecture 21: DAG-Consistent Distributed Shared Memory

In this lecture we will consider a dag-consistency model for shared memory. We will start with description of some properties we would like our shared memory model to have, and we will present an intuitive definition of one model, dag-consistency, which satisfies our constraints. Then we will describe an algorithm that maintains dag-consistency, the Backer coherence algorithm. We will consider a series of anomalies, and we will reconsider our definition of dag-consistency based on each of these anomalies. We will also prove the correctness of Backer algorithm, and discuss the limitations of dag-consistency.

1 Model

1.1 What do we want?

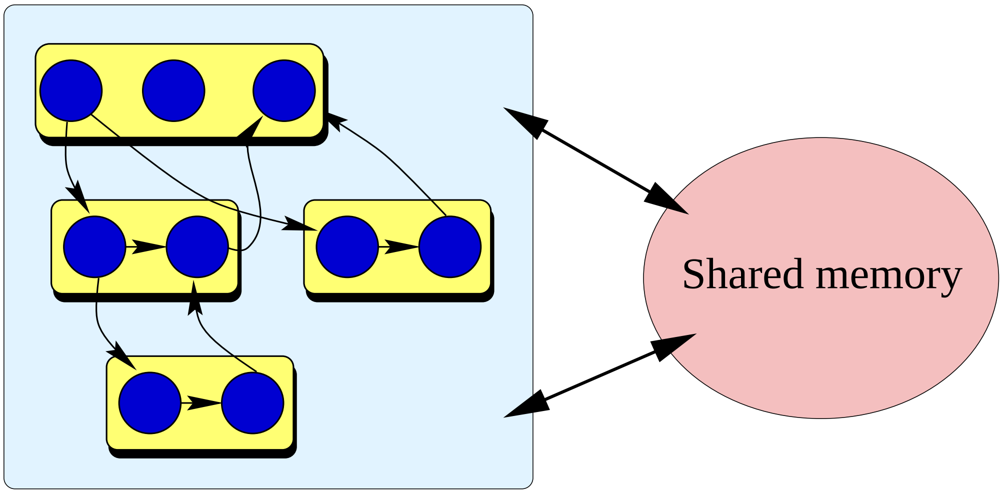


Figure 1: Cilk shared memory model

Cilk-1 and 2 had no shared memory, so all the programming was done in functional style. This was inefficient for many reasons, one of which was the need to copy an object every time it had to be used. Therefore, a need for shared memory became apparent, and the following criteria for the model to be used in Cilk were developed:

- we do not want to model processors explicitly,

- we do not want to pay for features we do not use (ala RISC),[1]
- we want the model to be easy to understand,
- we do not care about the interaction of things running in parallel (recall the matrix multiplication Divide and Conquer algorithm),
- we want to guarantee that all results are stored at sync.

1.2 DAG-consistency

We will now intuitively derive a model of consistency satisfying the above criteria. Consider the dag in Figure 2:

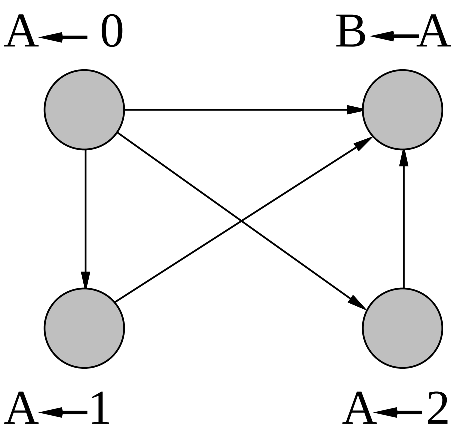


Figure 2: A case for examining the intuitive memory model for Dag consistency

Our intuition tells us that B can get the value of either 1 or 2, but not 0. This is because before B $\leftarrow$ A is executed, both A $\leftarrow$ 1 and A $\leftarrow$ 2 have to execute, but in either order. This intuition leads to the following definition of dag-consistency:

Definition 1: Memory is dag-consistent if, when a thread k reads a value, there is a sequential execution in which k reads the same value.

We can test our intuitive understanding of dag-consistency again on the dag in Figure 3. What value does B get? Here it is 0 or 1, but not 2. This is because A $\leftarrow$ 2 in a sequential execution would happen after B $\leftarrow$ A, which could happen before or after A $\leftarrow$ 1.

Notice that dag-consistency is fast. As indicated in Figure 4, on a CM5, the matrix-multiplication algorithm scales, and runs very close to the machine peak.

[1]The reason for this becomes apparent if we consider that in sparc read/write to same location costs 5 cycles, because there needs to be a processor observing this transaction. If we have no need for such an observer, we still pay for it under the strong models of consistency

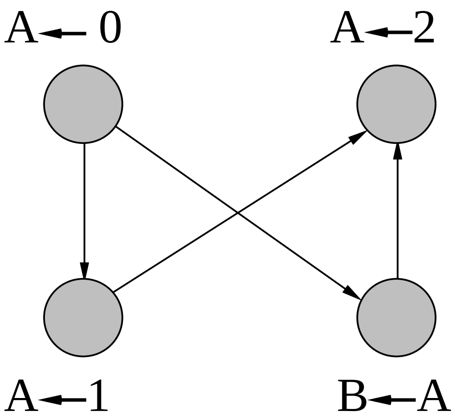


Figure 3: Which value does B get?

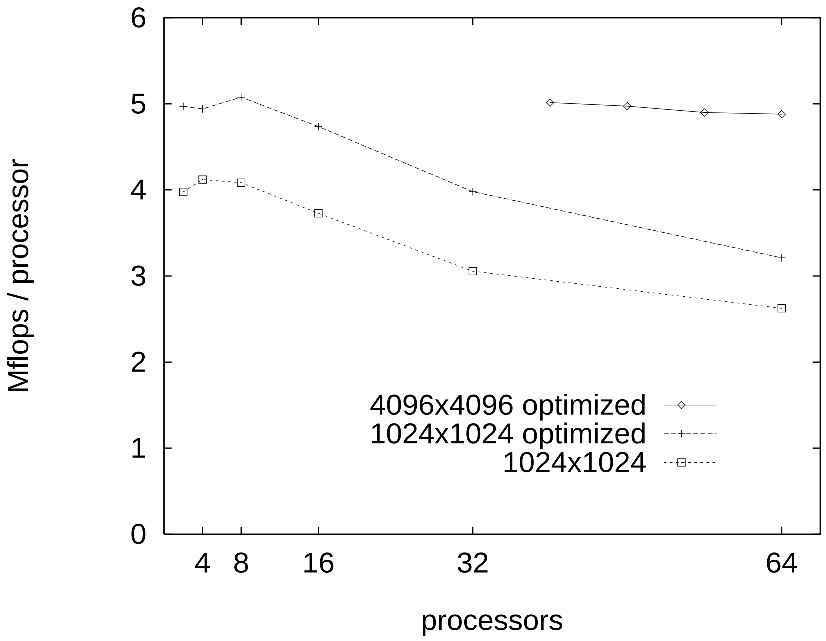


Figure 4: Dag consistency performance on matrix multiplication

2 Backer Coherence Algorithm

In this section we will consider the general model of distributed shared memory, as well as a particular algorithm, Backer coherence algorithm, which implements dag-consistency within the framework of distributed shared memory. We will describe the algorithm, present an example of how Backer works, describe a software implementation, and discuss the properties of Backer algorithm.

2.1 Distributed Shared Memory

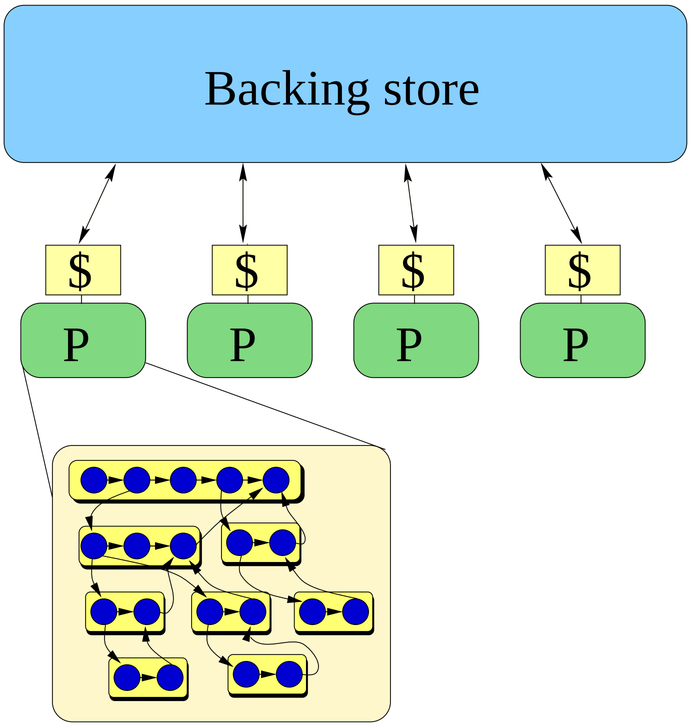


Figure 5: Distributed shared memory architecture

Some properties of distributed shared memory:

- The processors only operate on their cache,
- The cache maintains *dirty bits* to detect which objects have been modified (processor writes in these to indicate that it has modified an object),.
- When a processor needs an object not in the cache, the object is *fetched*,
- When a *dirty* (modified) object is kicked out of the cache, it is *reconciled* with the backing store by writing it back.

2.2 Coherence Algorithm– BACKER

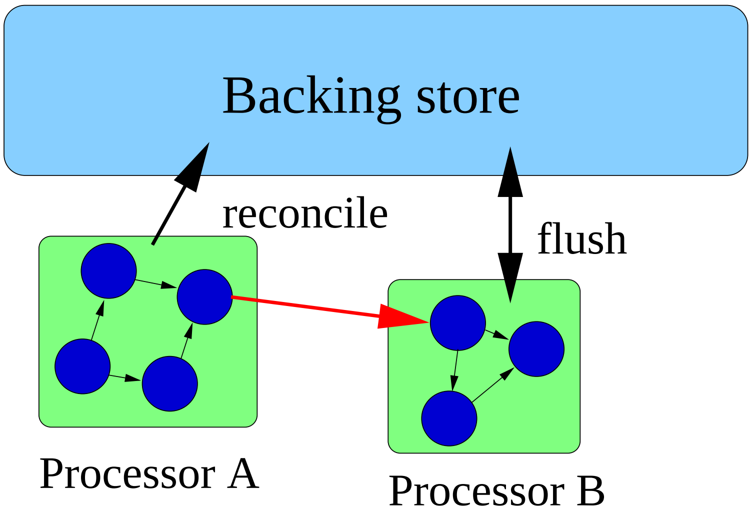


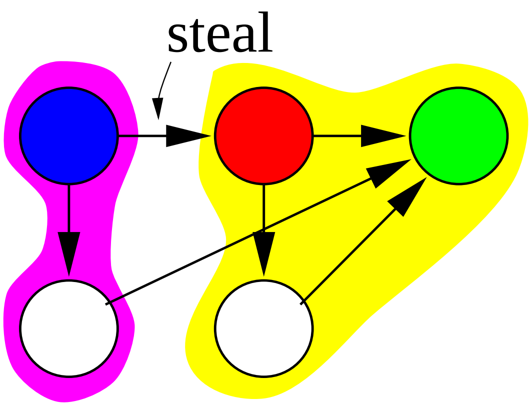
Figure 6: The Backer coherence algorithm

Principles of Backer:

- The scheduler partitions the dag between processors,
- Some dag edges connect nodes in different processors,
- Before processor A enables a thread on another processor B, it *reconciles* all its dirty objects with the backing store, and causes processor B to *reconcile* and *flush* its cache.

2.3 How Backer Works– Example

```
cilk mergesort(X, l, r)
{
     m = (l + r) / 2;
     spawn mergesort(X, l, m);
     spawn mergesort(X, m+1, r);
     sync;
     merge(X, l, m, r);
}
```



Processor A reconciles with the backing store all the work it did. Only then processor B, which stole work from processor A, is allowed to continue with the merge.

2.4 Backer in Practice– Software Implementation

Some challenges and solutions to them when designing a software implementation of Backer are:

- Communication is expensive

  ← can not afford to send a message to retrieve 32 bits

  → organize objects into pages (4kb) and move one page at a time. This is based on the idea of locality – if a processor needs a certain piece of data now, it is likely to need its close neighbors soon. So the hope is to save on communication costs by using pages.

- No hardware-maintained dirty bits

  ← keeping own dirty bits turns out to be expensive (for writes)

  → when fetching a page, create a twin. Use the twin to identify dirty objects (i.e. do an XOR on the page and its twin. This means we pay only at compare time, not at write time).

Figure 7 presents an illustration of how twins are used.

Action	Backing store	A'a cache	A's twin
initially	[0 1 2 3]	empty	empty
A fetches	[0 1 2 3]	[0 1 2 3]	[0 1 2 3]
A writes	[0 1 2 3]	[0 1 2 **4**]	[0 1 2 3]
A writes	[0 1 2 3]	[0 1 **5** 4]	[0 1 2 3]
B reconciles	[0 **6** **7** 3]	[0 1 5 4]	[0 1 2 3]
A reconciles	[0 6 **5** **4**]	[0 1 **5** **4**]	[0 1 **5** **4**]

Figure 7: The use of twins in a software implementation of Backer

Notice that at reconcile time we:

1. Compare the page with its twin, and send to the backing store the objects that are different.
2. Also update the twin.

It is interesting to observe that in Cilk there are only three reasons to reconcile:

1. When a steal occurs, victim needs to reconcile.
2. When a subcomputation is done, and the value needs to be passed to a node that is not on the same processor.
3. When a processor runs out of cache.

2.5 Properties of Backer

Some of the properties of Backer are:

- It is simple to understand and implement.
- It is empirically fast (recall CM5 implementation).
- It is provably efficient.
- No thrashing due to "false sharing" occurs.
- There is bilateral communication between processors and backing store.
- It does "The Right Thing^{TM}", even though Definition 1 is broken. In other words, *it is easy to reason about the multi-threaded program.*

Recall that dag consistency is a very weak model of consistency. For example, if two things are spawned, they must be completely independent (we can use locks to force otherwise, but it is "unnatural"). However, in this model we are not paying for the overhead we are not using.

Since Backer breaks Definition 1, but still does "The Right Thing^{TM}", we would like to know exactly what memory model DOES Backer maintain? We will answer that question in the next section

3 Refining the Definition of DAG-Consistency

3.1 Sequential Rewriting

Recall Definition 1:

Definition 1: Memory is dag-consistent if, when a thread k reads a value, there is a sequential execution in which k reads the same value.

Now consider the dag in Figure 8. It is easy to see that in no serial execution does E$\leftarrow$ A occur. However, it can occur in a parallel execution. Moreover, we expect that A=1 at the time of the execution of the last node. Notice that Definition 1 is OK for the 100% Nondeterminator-clean programs, i.e those programs that have no concurrent reads/writes from/to the same location.

In light of the above anomaly we derive a new definition of dag-consistency:

Definition 2: Memory is dag-consistent if, for every triple i, j, and k satisfying $i \prec j \prec k$, if i writes to m and j writes to m, then k does not see i's write.

This definition is somewhat tricky, since we say what a thread can not read, rather then what it can read. Another words, if there was a node a, not connected to k in the sequential execution, k could see a's write.

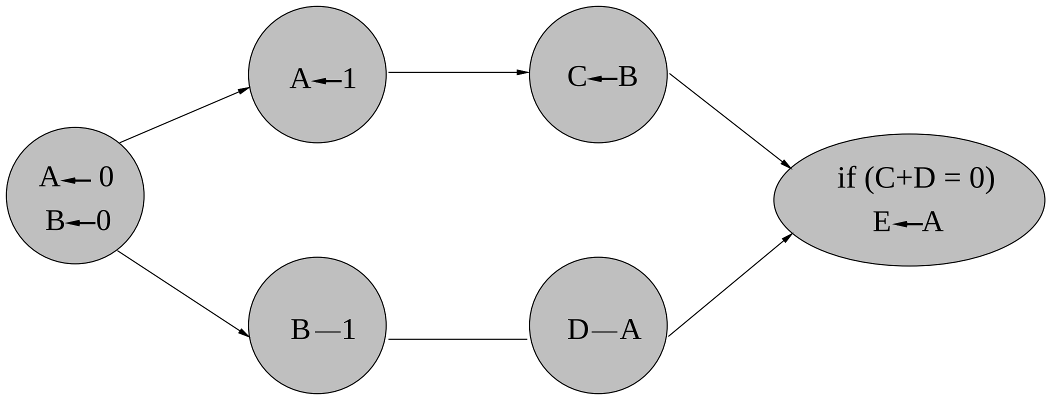


Figure 8: "Existential" anomaly

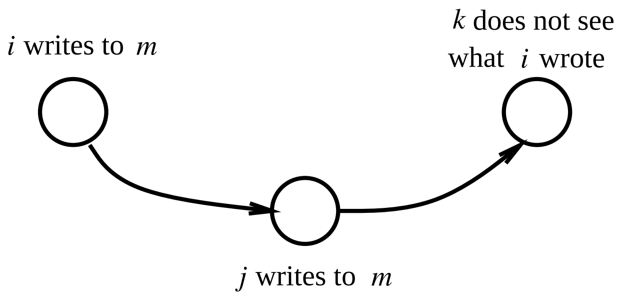


Figure 9: Thread j write of location m hides i's write of the same location from k.

3.2 "Hamlet" Anomaly and Its Solution

Consider the dag in Figure 10.

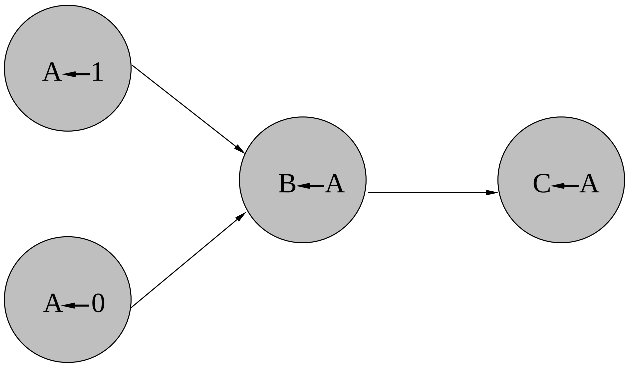


Figure 10: "Hamlet" anomaly

Can it be that B≠C? With dag-2 consistency– yes, it can, since the middle node does not write to A, and so does not prevent C from choosing a different value of A then the one B chose. Thus, this is termed "Hamlet" anomaly– decisions, decisions...

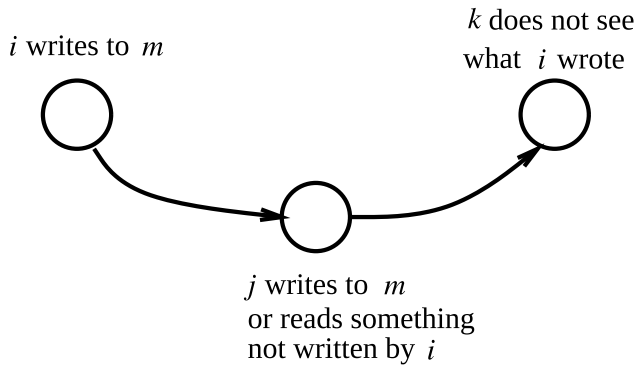


Figure 11: The action of the intermediate thread j determines the behavior of the following thread, k. By either writing a new value to m or reading a value that some other thread has written, j forces all the following threads to read that value, until m may be overwritten by another thread. (see Definition 3)

To handle the "Hamlet" anomaly we introduce a third definition.

Definition 3: Memory is *dag consistent* if, for every triple i, j and k satisfying $i \prec j \prec k$, if i writes and j writes, then k does not see i's write; moreover if i writes and j reads something not written by i, then k does not see i's write either. (see Figure 11)

3.3 'Quantum' anomaly and Observer Functions

Notice that the DAG in Figure 12 allows a sort of "quantum" anomaly. This example shows that observing the value alters the system! This does not seem right.

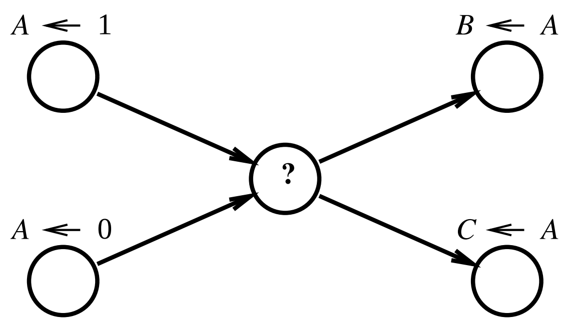


Figure 12: Quantum anomaly: In dag-3 consistency, if the thread marked with '?' reads A, then $B = C$. Otherwise, it may happen that $B \neq C$.

Observer functions fix all these problems. An observer function maps a thread's view of a memory location into the thread that wrote the value that the viewer will see.

Definition 4: Memory is *dag consistent* if, for every memory location m, there exists a function $f_m : V \mapsto V$ such that:

1. $f_m(i)$ writes, and $i \not\prec f_m(i)$.
2. If thread i reads, it reads the value written by $f_m(i)$.
3. If thread i writes, then $f_m(i) = i$.
4. If $i \prec j \prec k$ and $f_m(k) = i$, then $f_m(j) = i$. Equivalently, if $i \prec j \prec k$ and $f_m(j) \neq i$, then $f_m(k) \neq i$.

The observer function requires that all nodes have some viewpoint on the memory. Thus, even if a node does not perform memory operations, it still carries synchronization semantics.

3.4 Example of Definition 4

Dag-4 consistency confines nondeterminism. When multiple threads all flow into a single thread, when the DAG is constricted, then the nondeterminism prior to that single thread collapses. This is forced by our requirement that each thread has a viewpoint on memory.

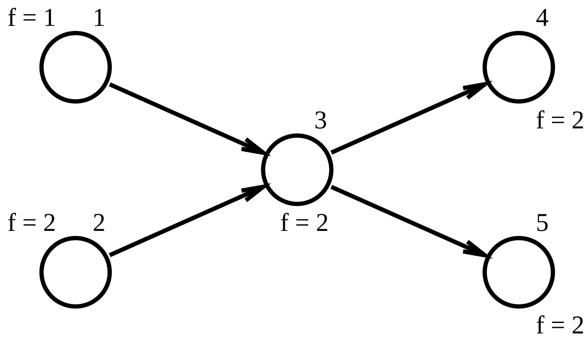


Figure 13: The definition forces thread 3 to have a viewpoint on the memory, which influences the successor threads. In the example, $f_m(3) = 2$.

4 Correctness of Backer

We can now prove that the Backeralgorithm is correct under definition 4 of dag consistency.

Construct and observer function $f_m(i)$ satisfying the four conditions of the dag-4 definition.

2. If thread i reads, it reads the value written by $f_m(i)$.
3. If thread i writes, then $f_m(i) = i$.

Also, If thread i does nothing, pretend it is reading.

We must prove:

1. $f_m(i)$ writes, and $i \not\prec f_m(i)$.
   - Obvious: computers cannot see into the future.
4. If $i \prec j \prec k$ and $f_m(k) = i$, then $f_m(j) = i$.
   - Prove the contrapositive: if $f_m(j) \neq i$, then $f_m(k) \neq i$. Because the way Backer works, a value that has been overwritten cannot appear again.

5 Limits of dag consistency

First of all, we note that deterministic programs work fine with dag consistency. We only run into problems with nondeterministic programs.

- Some nondeterministic programs do not work at all.
- Other nondeterministic programs work with DC, but could be faster with a stronger memory model.

5.1 A Nondeterministic Program That Doesn't Work

Consider Dekker's algorithm:

```
#include <cilk.h>

int firing[2];

cilk int main(void)
{
    firing[0] = 0;
    firing[1] = 0;
    spawn launcher0();
    spawn launcher1();
    sync;
}
```

```
cilk void launcher0(void)
{
    firing[0] = 1;
    if (firing[1] == 0)
        launch_rocket();
}

cilk void launcher1(void)
{
    firing[1] = 1;
    if (firing[0] == 0)
        launch_rocket();
}
```

Dekker does not work. Neither thread will see the update to `firing` made by the other, and so both may launch the rocket. It cannot even be fixed, because we do not have safety nets like the `membar` operation which would force updates to shared memory.

5.2 Slow Nondeterministic Programs

Consider a typical branch and bound algorithm.

- We are walking a tree, looking for a node of maximum value.
- At each node we can compute an upper bound to the maximum value in the subtree.
- If the upper bound is less than the best node so far, there is no point in exploring that subtree.

The algorithm works even if the "best so far" is out of date, but is slower. This is because we will end up doing extra work exploring subtrees which are sure to result in values lower than the actual "best so far." Since the threads are not able to share their findings of the maximum value, some threads may be working with sub-optimal estimates of the "best so far."

6 Summary

- Dag consistency is easy [sic] to understand.
- Dag consistency is defined in terms of the computation dag only.

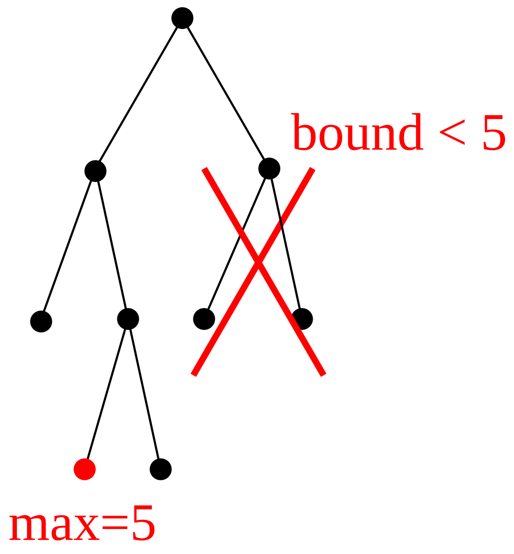


Figure 14: If we have found a maximum value of 5 so far, and we know that the right subtree cannot have a value greater or equal to 5, then we do not need to search the right subtree.

- There are no locks in the model.
- DC is really relaxed, but still sufficient in many cases.
- There is a simple algorithm (BACKER) for maintaining dag consistency.
- We have a fast implementation.
- The performance bounds of Cilk are maintained when BACKER is used.