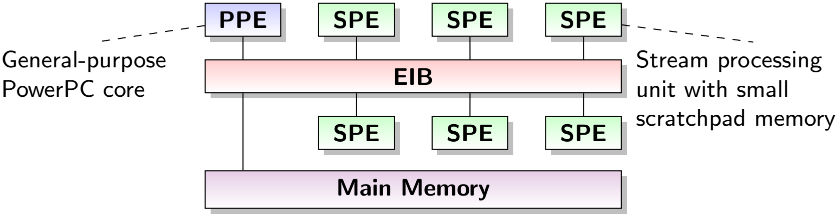
Strengthening Induction-Based Race Checking with Lightweight Static Analysis

A. Donaldson L. Haller D. Kroening

Oxford University Computing Laboratory

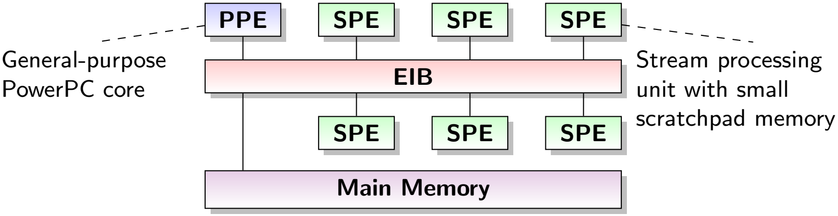
VMCAI 2011

Cell BE processor



SPE cores have small (kb) and very fast scratchpad memory, to which they have exclusive access.

Cell BE processor



SPE cores have small (kb) and very fast scratchpad memory, to which they have exclusive access.

- SPE cores cannot not directly access main memory.
- DMA (direct memory access) library calls move data to and from scratchpad *asynchronously*

Problem

- Scratchpad memories lead to high performance
  - this comes at the expense of program complexity!
- **Massive** scope for errors with DMA operations due to possible race conditions

Problem

- Scratchpad memories lead to high performance
  - this comes at the expense of program complexity!
- **Massive** scope for errors with DMA operations due to possible race conditions

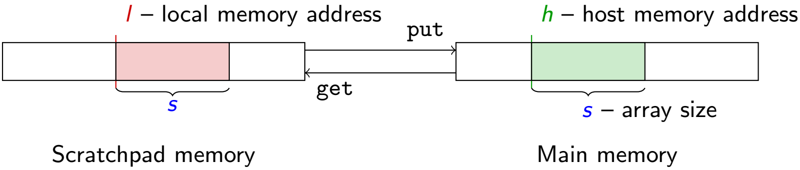
Contribution

- We apply k-induction to DMA programs to verify absence of DMA races.
- k-induction alone is too weak to verify all properties of interest.
- We strengthen k-induction using lightweight static analysis techniques

DMA operations

DMA requests are issued using library function calls:

$\texttt{get}(l, h, s, t)$ – load data into scratchpad memory
$\texttt{put}(l, h, s, t)$ – write data into main memory
$\texttt{wait}(t)$ – wait for all ops with tag t to finish



- Many concurrent DMAs can be issued simultaneously
- Latency can be hidden by using multiple buffers

DMA races

Scheduling of DMA operations changes result $\longrightarrow$ Races can occur!

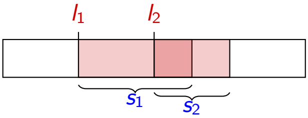
DMA races

Scheduling of DMA operations changes result $\longrightarrow$ Races can occur!

- Races between two DMA operations;

  put(l_2, h_2, s_2, t_2);

  get(l_1, h_1, s_1, t_1);



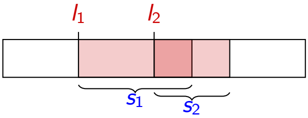
DMA races

Scheduling of DMA operations changes result $\longrightarrow$ Races can occur!

- Races between two DMA operations;

  put(l_2, h_2, s_2, t_2);

  get(l_1, h_1, s_1, t_1);



- Races between a DMA operation and local data access;

```
int a[10];
get(&a,h,10*sizeof(int),t);
a[0]=10;
```

Triple buffering code example

```
#define CHUNK 16384 // Process data in 16K chunks
float buffers[3][CHUNK/sizeof(float)]; // Three buffers for triple buffering

void process_data(float* buf) { ... }

void triple_buffer(char* in, char* out, int num_chunks) {
  unsigned int tags[3] = { 0, 1, 2 }, put_buf, get_buf, process_buf;
  get(buffers[0], in, CHUNK, tags[0]);
  in += CHUNK;
  get(buffers[1], in, CHUNK, tags[1]);
  in += CHUNK;
  wait(tags[0]);
  process_data(buffers[0]);
  put_buf = 0; process_buf = 1; get_buf = 2;
  for(int i = 2; i < num_chunks; i++) {
    put(buffers[put_buf], out, CHUNK, tags[put_buf]);
    out += CHUNK;
    get(buffers[get_buf], in, CHUNK, tags[get_buf]);
    in += CHUNK;
    wait(tags[process_buf]);
    process_data(buffers[process_buf]);
    int tmp = put_buf; put_buf = process_buf;
    process_buf = get_buf; get_buf = tmp;
  }
  ... // Handle data processed/fetched on final loop iteration
}
```

Triple buffering code example

```c
#define CHUNK 16384 // Process data in 16K chunks
float buffers[3][CHUNK/sizeof(float)]; // Three buffers for triple buffering

void process_data(float* buf) { ... }

void triple_buffer(char* in, char* out, int num_chunks) {
  unsigned int tags[3] = { 0, 1, 2 }, put_buf, get_buf, process_buf;
  get(buffers[0], in, CHUNK, tags[0]);
  in += CHUNK;
  get(buffers[1], in, CHUNK, tags[1]);
  in += CHUNK;
  wait(tags[0]);
  process_data(buffers[0]);
  put_buf = 0; process_buf = 1; get_buf = 2;
  for(int i = 2; i < num_chunks; i++) {
    put(buffers[put_buf], out, CHUNK, tags[put_buf]);
    out += CHUNK;
    get(buffers[get_buf], in, CHUNK, tags[get_buf]);
    in += CHUNK;
    wait(tags[process_buf]);
    process_data(buffers[process_buf]);
    int tmp = put_buf; put_buf = process_buf;
    process_buf = get_buf; get_buf = tmp;
  }
  ... // Handle data processed/fetched on final loop iteration
}
```

Buffers change roles in each iteration.

Illustration of bug

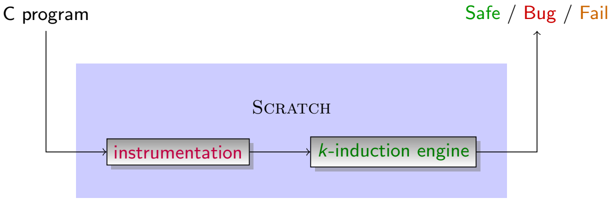
get	buffers[0]	in	CHUNK	tags[0]
get	buffers[1]	in	CHUNK	tags[1]
wait				tags[0]
process	**buffers[0]**			
Loop head				
put	buffers[0]	in	CHUNK	tags[0]
get	buffers[2]	in	CHUNK	tags[2]
wait				tags[1]
process	**buffers[1]**			
Loop head				
put	buffers[1]	in	CHUNK	tags[1]
get	buffers[0]	in	CHUNK	tags[0]

Illustration of bug

get	buffers[0]	in	CHUNK	tags[0]
get	buffers[1]	in	CHUNK	tags[1]
wait				tags[0]
process	**buffers[0]**			
Loop head				
put	buffers[0]	in	CHUNK	tags[0]
get	buffers[2]	in	CHUNK	tags[2]
wait				tags[1]
process	**buffers[1]**			
Loop head				
put	buffers[1]	in	CHUNK	tags[1]
get	buffers[0]	in	CHUNK	tags[0]

Race on buffers[0]

Asserting race-freedom with SCRATCH



- Establishes race freedom for code running on a single SPE node.
- Based on the CBMC bounded model checker
- Calls to `put`, `get`, and `wait` are instrumented with assertions.
- The resulting program is analyzed with a *k*-induction engine.

Instrumenting DMA programs

Add a tracker datastructure:

```
struct DMA_op {
  bool     valid;
  char*   address;  // Local store address
  unsigned size;     // Num bytes to transfer
  unsigned  tag;      // Identifying tag

};

struct DMA_op tracker = { 0, *, *, * };
```

Used to store one single pending DMA request.

Instrumenting DMA programs

A call get(l, h, s, t) is translated to:

```
assert(t < 32);              // Check tag in range
assert(s < 16K);             // Check DMA not too large
assert(!tracker.valid        // Check no race with prior DMA
     || l + s <= tracker.address
     || tracker.address + tracker.size <= l);
memset(l, *, s);             // Over-approximate effect of DMA

if(*) {
  tracker.valid = true;      // Nondeterministically decide
  tracker.address = l;       // whether to track this DMA
  tracker.size = s;
  tracker.tag = t;           // Model checker will try both
}                            // possibilities!
```

Instrumenting DMA programs

A call get(l, h, s, t) is translated to:

```
assert(t < 32);              // Check tag in range
assert(s < 16K);             // Check DMA not too large
assert(!tracker.valid        // Check no race with prior DMA
     || l + s <= tracker.address
     || tracker.address + tracker.size <= l);
memset(l, *, s);             // Over-approximate effect of DMA

if(*) {
  tracker.valid = true;      // Nondeterministically decide
  tracker.address = l;       // whether to track this DMA
  tracker.size = s;
  tracker.tag = t;           // Model checker will try both
}                            // possibilities!
```

A call wait(t) just becomes:

```
assume(tracker.tag != t);       // Simple as that!
```

Instrumenting DMA programs

A call get(l, h, s, t) is translated to:

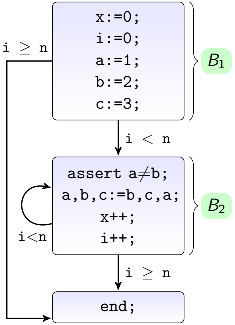
```
assert(t < 32);             // Check tag in range
assert(s < 16K);            // Check DMA not too large
assert(!tracker.valid       // Check no race with prior DMA
    || l + s <= tracker.address
    || tracker.address + tracker.size <= l);
memset(l, *, s);            // Over-approximate effect of DMA

if(*) {
  tracker.valid = true; // Nondeterministically decide
  tracker.address = l;  // whether to track this DMA
  tracker.size = s;
  tracker.tag = t;      // Model checker will try both
}                       // possibilities!
```

A call wait(t) just becomes:

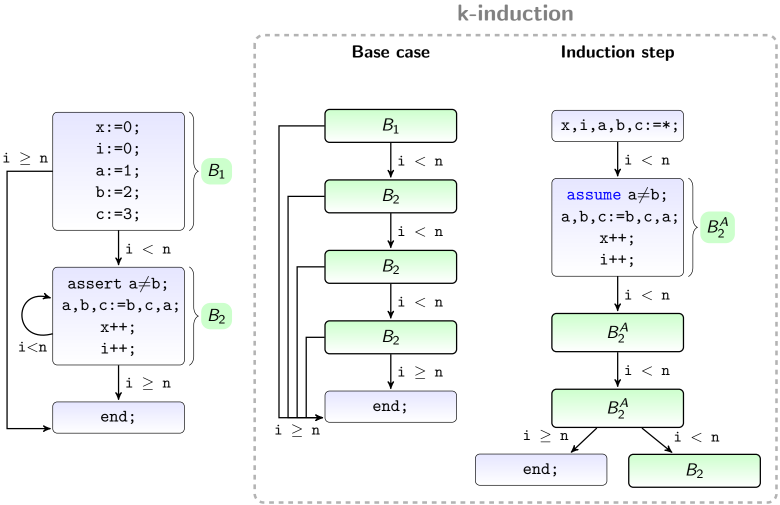
```
assume(tracker.tag != t);     // Simple as that!
```

The resulting program is checked using k-induction.

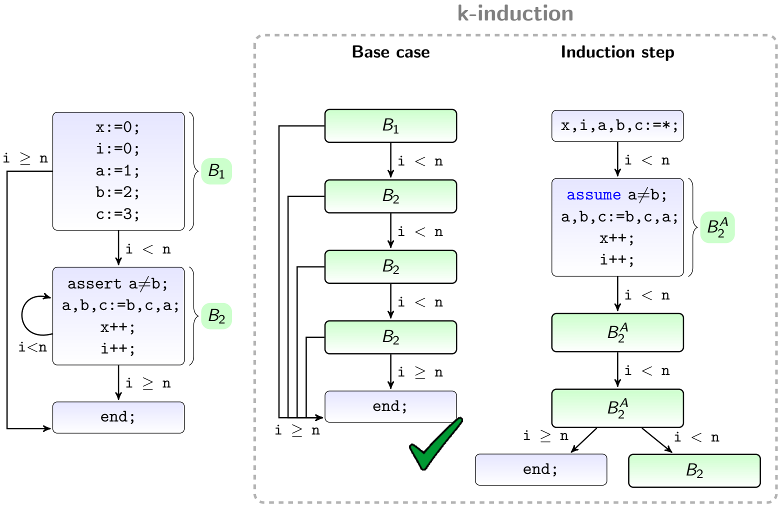
k-Induction example



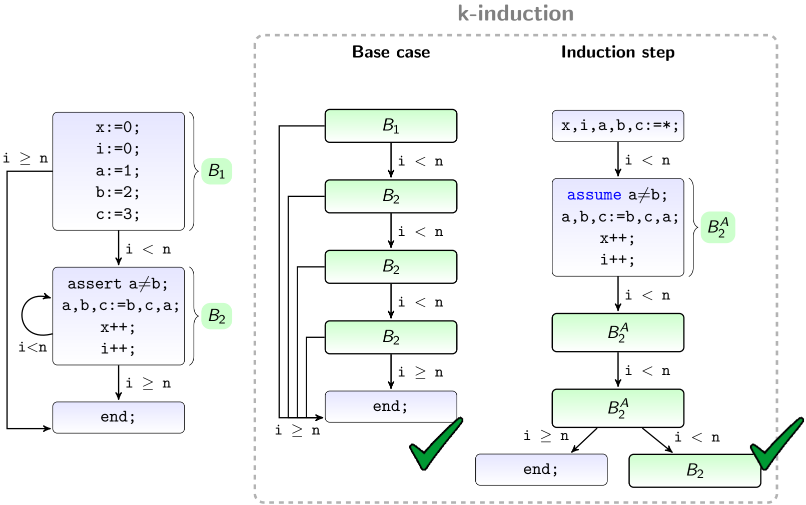
k-Induction example



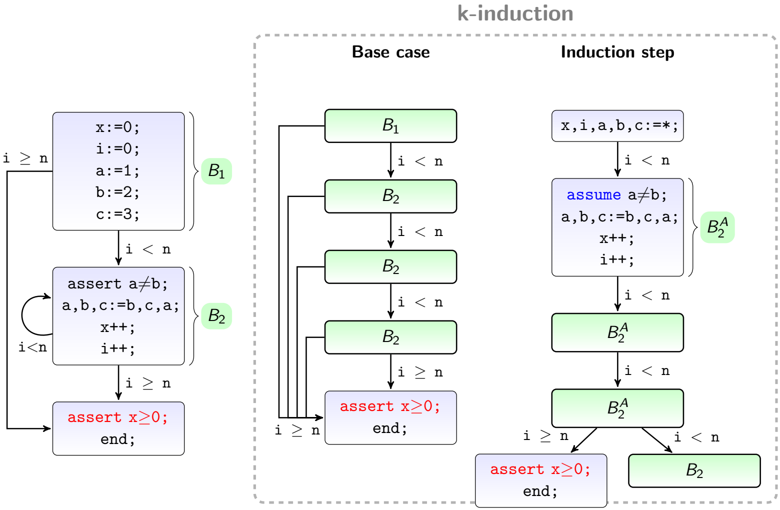
k-Induction example



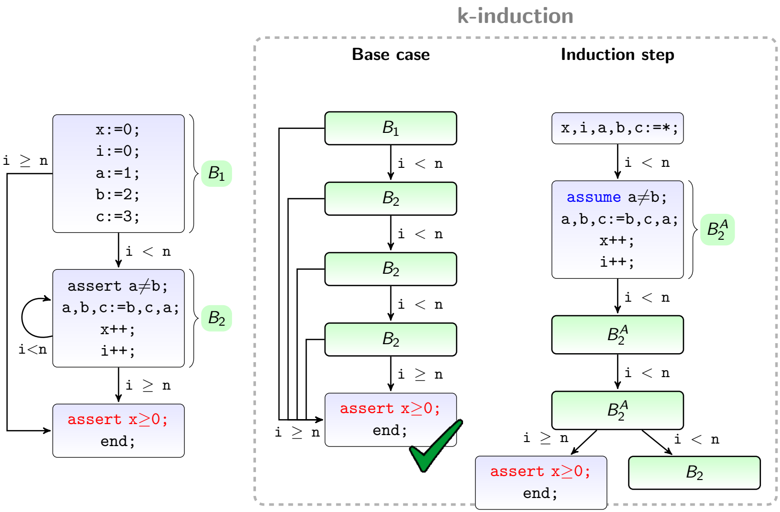
k-Induction example



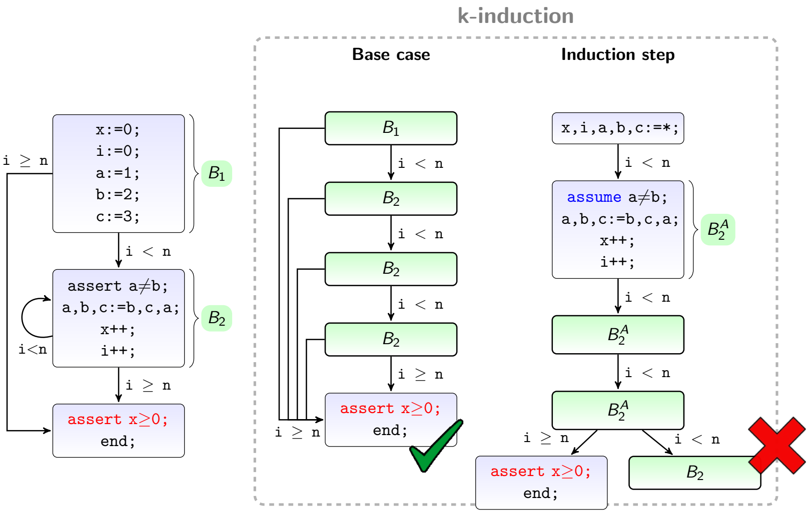
k-Induction example



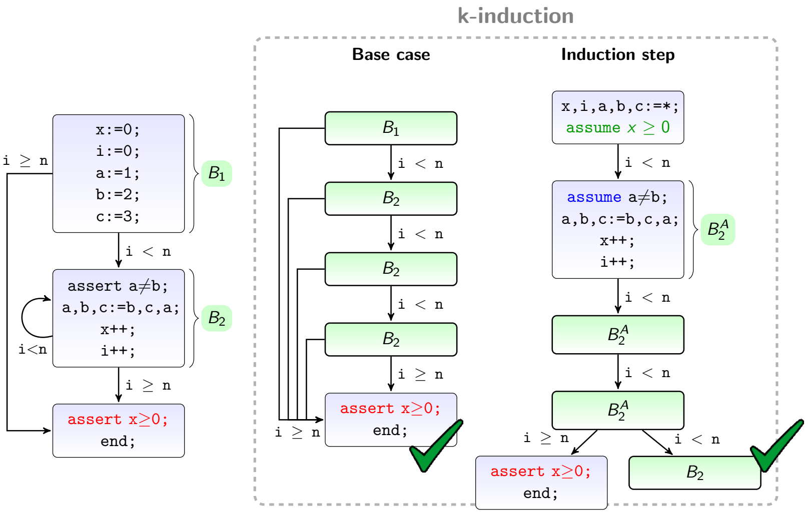
k-Induction example



k-Induction example



k-Induction example



k-Induction for software

$$\frac{\begin{array}{c@{\qquad}c} \textbf{Base case} & \textbf{Step case} \\ s_\alpha\,; & \\ \overbrace{\texttt{if}(\phi)\ \ s_\beta\ \ \ldots\ \texttt{if}(\phi)\ \ s_\beta}^{k\ \texttt{times}} & \overbrace{\texttt{assume}(\phi)\texttt{;}\ s_\beta^{assume}\texttt{;}\ \ldots\texttt{;}}^{k\ \texttt{times}} \\ \texttt{if}(\neg\phi)\ \ s_\gamma\ \ \text{is correct} & \texttt{if}(\phi)\ \ s_\beta\ \ \texttt{else}\ \ s_\gamma\ \ \text{is correct} \end{array}}{s_\alpha;\ \textbf{while}(\phi)\,\{\ s_\beta\ \};\ s_\gamma\ \ \text{is correct}}$$

k-Induction for software

$$\frac{\begin{array}{c}\textbf{Base case} \\ s_\alpha\texttt{;}\ \overbrace{\texttt{if}(\phi)\ s_\beta\ \ldots\ \texttt{if}(\phi)\ s_\beta}^{k\ \texttt{times}} \\ \texttt{if}(\neg\phi)\ s_\gamma\ \text{ is correct}\end{array} \qquad \begin{array}{c}\textbf{Step case} \\ \overbrace{\texttt{assume}(\phi)\texttt{;}\ s_\beta^{assume}\texttt{;}\ \ldots\texttt{;}}^{k\ \texttt{times}} \\ \texttt{if}(\phi)\ s_\beta\ \texttt{else}\ s_\gamma\ \text{ is correct}\end{array}}{s_\alpha;\ \textbf{while}(\phi)\,\{\,s_\beta\,\};\ s_\gamma\ \text{ is correct}}$$

- Base case failure: There is a bug of depth at most k
- Step case failure: Choose higher k, or abandon proof attempt
- Multiple loops transformed to single monolithic loop

k-Induction for software

$$\frac{\begin{array}{l}\textbf{Base case}\\ s_\alpha;\ \overbrace{\texttt{if}(\phi)\ s_\beta\ \ldots\ \texttt{if}(\phi)\ s_\beta}^{k\ \texttt{times}}\\ \texttt{if}(\neg\phi)\ s_\gamma\ \text{is correct}\end{array}\qquad\begin{array}{l}\textbf{Step case}\\ \overbrace{\texttt{assume}(\phi);\ s_\beta^{assume};\ \ldots;}^{k\ \texttt{times}}\\ \texttt{if}(\phi)\ s_\beta\ \texttt{else}\ s_\gamma\ \text{is correct}\end{array}}{s_\alpha;\ \textbf{while}(\phi)\,\{\ s_\beta\ \};\ s_\gamma\ \text{is correct}}$$

- Base case failure: There is a bug of depth at most k
- Step case failure: Choose higher k, or abandon proof attempt
- Multiple loops transformed to single monolithic loop

But as we have seen this is not always enough!

Strengthening k-induction

Transition system $M = (S, T, I)$. Set of error states E.

$post_T(Q)$, set of successors of states in Q

$safe^k(Q)$ iff no error states reachable in k steps

Strengthening k-induction

Transition system $M = (S, T, I)$. Set of error states E.

$post_T(Q)$, set of successors of states in Q

$safe^k(Q)$ iff no error states reachable in k steps

Inductive invariant

$$\frac{\text{(i)}\ \overbrace{I \subseteq Q \qquad \text{(ii)}\ post_T(Q) \subseteq Q}^{\text{Guess inductive invariant}} \qquad \text{(iii)}\ Q \cap E = \emptyset}{M \text{ safe}}$$

Strengthening k-induction

Transition system $M = (S, T, I)$. Set of error states E.

$post_T(Q)$, set of successors of states in Q

$safe^k(Q)$ iff no error states reachable in k steps

Guess inductive invariant

Inductive invariant
$$\frac{\text{(i)}\ \overbrace{I \subseteq Q \qquad \text{(ii)}\ post_T(Q) \subseteq Q} \qquad \text{(iii)}\ Q \cap E = \emptyset}{M \text{ safe}}$$

***k*-induction**
$$\frac{k \geq 0 \qquad \text{(a)}\ safe^k(I) \qquad \text{(b)}\ \forall Q.\ safe^k(Q) \Rightarrow safe^{k+1}(Q)}{M \text{ safe}}$$

Strengthening k-induction

Transition system $M = (S, T, I)$. Set of error states E.

$post_T(Q)$, set of successors of states in Q

$safe^k(Q)$ iff no error states reachable in k steps

Inductive invariant

$$\frac{\text{(i)}\ \overbrace{I \subseteq Q \qquad \text{(ii)}\ post_T(Q) \subseteq Q}^{\text{Guess inductive invariant}} \qquad \text{(iii)}\ Q \cap E = \emptyset}{M \text{ safe}}$$

k-induction

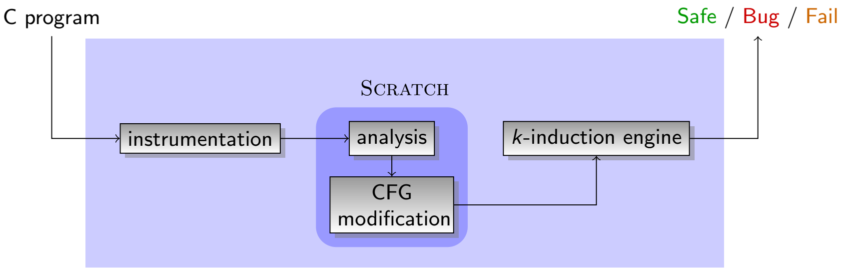
$$\frac{k \geq 0 \quad \text{(a)}\ safe^k(I) \quad \text{(b)}\ \forall Q.\, safe^k(Q) \Rightarrow safe^{k+1}(Q)}{M \text{ safe}}$$

Combined

$$\frac{\begin{array}{c} k \geq 0 \quad \text{(i)}\ I \subseteq Q \quad \text{(ii)}\ post_T(Q) \subseteq Q \\ \text{(a)}\ safe^k(I) \quad \text{(b)}\ safe^k(Q) \Rightarrow safe^{k+1}(Q) \end{array}}{M \text{ safe}}$$

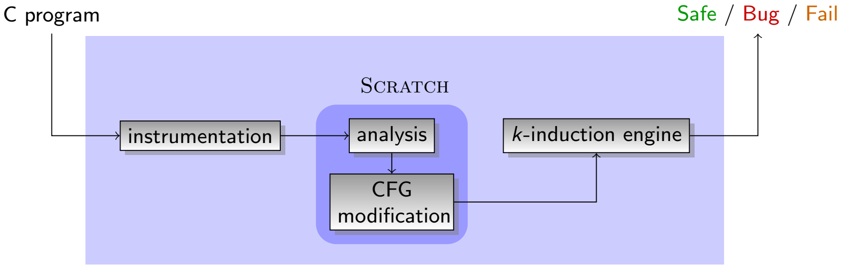
Strengthening induction-based race checking

Analysis step is added to the scratch pipeline:



Strengthening induction-based race checking

Analysis step is added to the scratch pipeline:



For modifying the CFG, we utilize

- Analysis with cheap abstract domains
- Code motion
- Assertion chunking

Abstract Domains

We utilize a *reduced product* of two domains:

- the interval domain; $x \in [c_1, c_2]$
- an equality / disequality domain; $x = y$, $x \neq y$

Then annotate CFG with assumptions

$$inv\text{: control flow locations} \rightarrow \text{local invariants.}$$

- Prepend control flow nodes with `assume` statements:

$$\begin{array}{lcl} l_1 : \texttt{s1;} & & \texttt{assume(}inv(l_1)\texttt{); s1;} \\ l_2 : \texttt{s2;} & \longrightarrow & \texttt{assume(}inv(l_2)\texttt{); s2;} \\ l_3 : \texttt{s3;} & & \texttt{assume(}inv(l_3)\texttt{); s3;} \\ l_4 : \texttt{s4;} & & \texttt{assume(}inv(l_4)\texttt{); s4;} \end{array}$$

Chunking

Strengthening `assert` statements can help the inductive step of the proof.

- Chunking analysis identifies assertions over small contiguous memory regions
- Combines them into stronger assertions:

Chunking

Strengthening `assert` statements can help the inductive step of the proof.

- Chunking analysis identifies assertions over small contiguous memory regions
- Combines them into stronger assertions:

```
for(int i=0; i < SIZE; i++) {
  assert(noDMAop(a[i],sizeof(float)));
  a[i] := 1.0f;
}
```

Chunking

Strengthening `assert` statements can help the inductive step of the proof.

- Chunking analysis identifies assertions over small contiguous memory regions
- Combines them into stronger assertions:

```
for(int i=0; i < SIZE; i++) {
  assert(noDMAop(a[i],sizeof(float)));
  a[i] := 1.0f;
}
```

```
for(int i=0; i < SIZE; i++) {
  assert(noDMAop(a[0],SIZE*sizeof(float)));
  a[i] := 1.0f;
}
```

Code motion

- For performance reasons, DMA `get` operations issued as soon as possible.
- This makes induction-based verification more difficult.

Code motion

- For performance reasons, DMA get operations issued as soon as possible.

- This makes induction-based verification more difficult.

Code motion is used to reverse this process.

```
get(l,h,s,t);
s2;
while(...)
{...}
s3;
```

→

```
get_possible(l,h,s,t);
s2;
while(...)
{...}
get(l,h,s,t);
s3;
```

- Swap independent statements to push back DMA operations.
- Insert check at original location (non-terminating loops!)
- Soundness of statement independence is checked using a SAT solver.

Experiments

Runtimes on a 3.2Ghz Intel Xeon 48GB:

Benchmark	**Lines of code**	**Time**	**of which AI**	k	**Max base case vars**	**Max step case vars**
single buffer	152	1.70	9.86%	2	5873	178305
single buffer IO	160	4.25	5.21%	3	6781	334915
double buffer	270	8.52	9.06%	2	67418	386705
double buffer IO	284	24.74	3.49%	3	132266	726512
triple buffer	379	44.32	6.46%	3	9208	650404
triple buffer IO	420	54.80	3.96%	3	9224	707592
double buffer TP	359	9.13	15.65%	2	109783	206434
double buffer IO TP	390	42.47	7.18%	3	215385	854164
triple buffer TP	611	138.10	7.13%	3	8813	958183
triple buffer IO TP	1813	422.45	3.39%	3	8824	3377134

Why does k-induction work in this domain

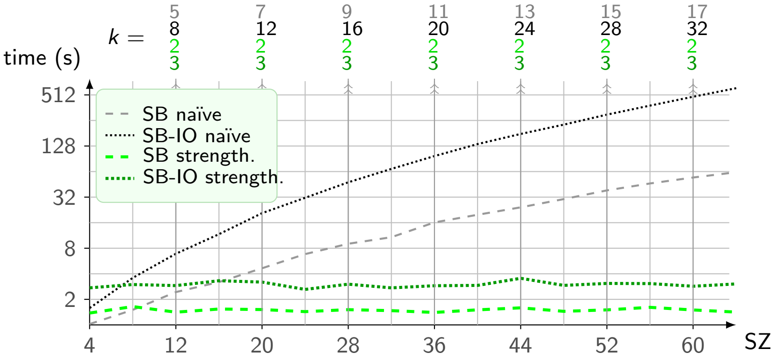
- k-induction works well for sequential hardware circuits with pipelines.
  - required k proportional to pipeline depth

Why does k-induction work in this domain

- k-induction works well for sequential hardware circuits with pipelines.
  - required k proportional to pipeline depth
- Buffering schemes used in DMA programs have a similar structure.
  - required k proportional to number of buffers

Effect of strengthening on k-induction

- Benchmarks cannot be verified without strengthening.
- To enable comparison, we verify simplified example programs, by restricting the size of the data buffer SZ.



Summary

- Detection of races in DMA programs
- Application of k-induction at loop level
- Strengthening of k-induction with lightweight static analysis
  - cheap abstract domains
  - assertion chunking
  - code motion

Summary

- Detection of races in DMA programs
- Application of *k*-induction at loop level
- Strengthening of *k*-induction with lightweight static analysis
  - cheap abstract domains
  - assertion chunking
  - code motion

Future work includes:

- Inter-thread race detection
- Widening the scope of *k*-induction beyond race checking

Thank you for your attention.