Automatic Analysis of Scratch-pad Memory Code for Heterogeneous Multicore Processors

Alastair F. Donaldson

EPSRC Postdoctoral Research Fellow

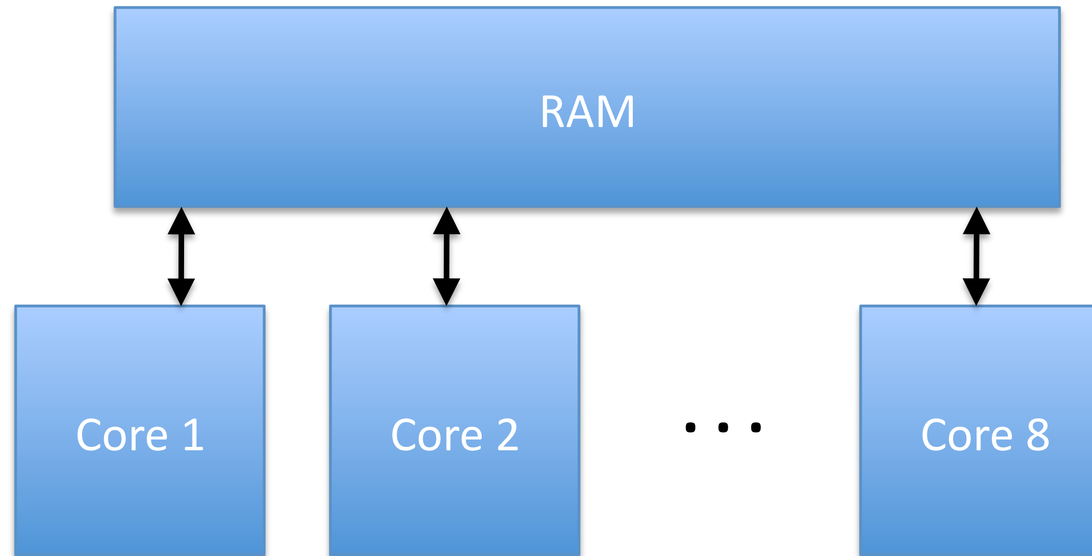
Oxford University Computing Laboratory

Joint work with Daniel Kroening and Philipp Ruemmer

Note

- In this PDF export, some slides may look weird as they contain PowerPoint animations that have been flattened. If you would like the original power point slides, please email alastair [dot] donaldson [at] comlab [dot] ox [dot] ac [dot] uk

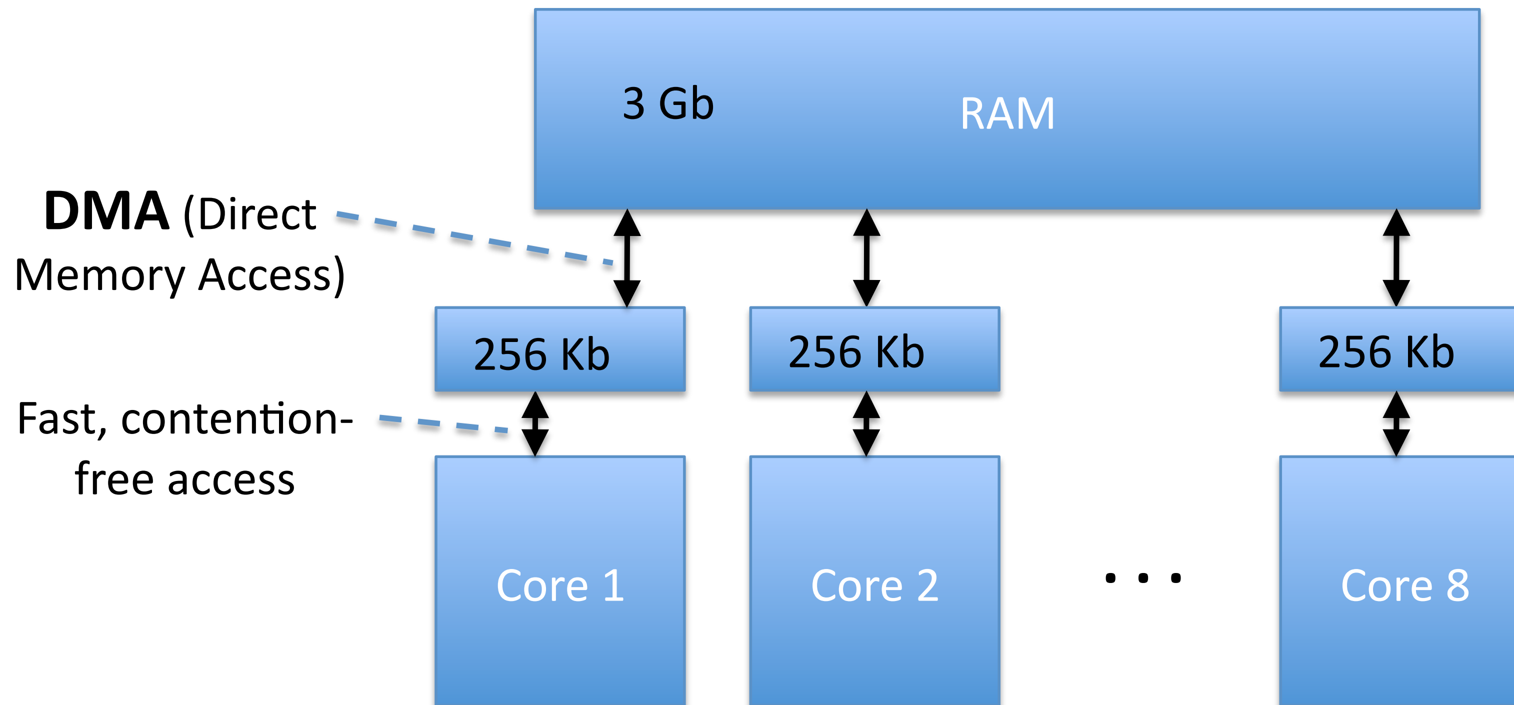
The memory wall problem



Memory becomes a bottleneck

Gains due to parallelism diminish as more cores are added

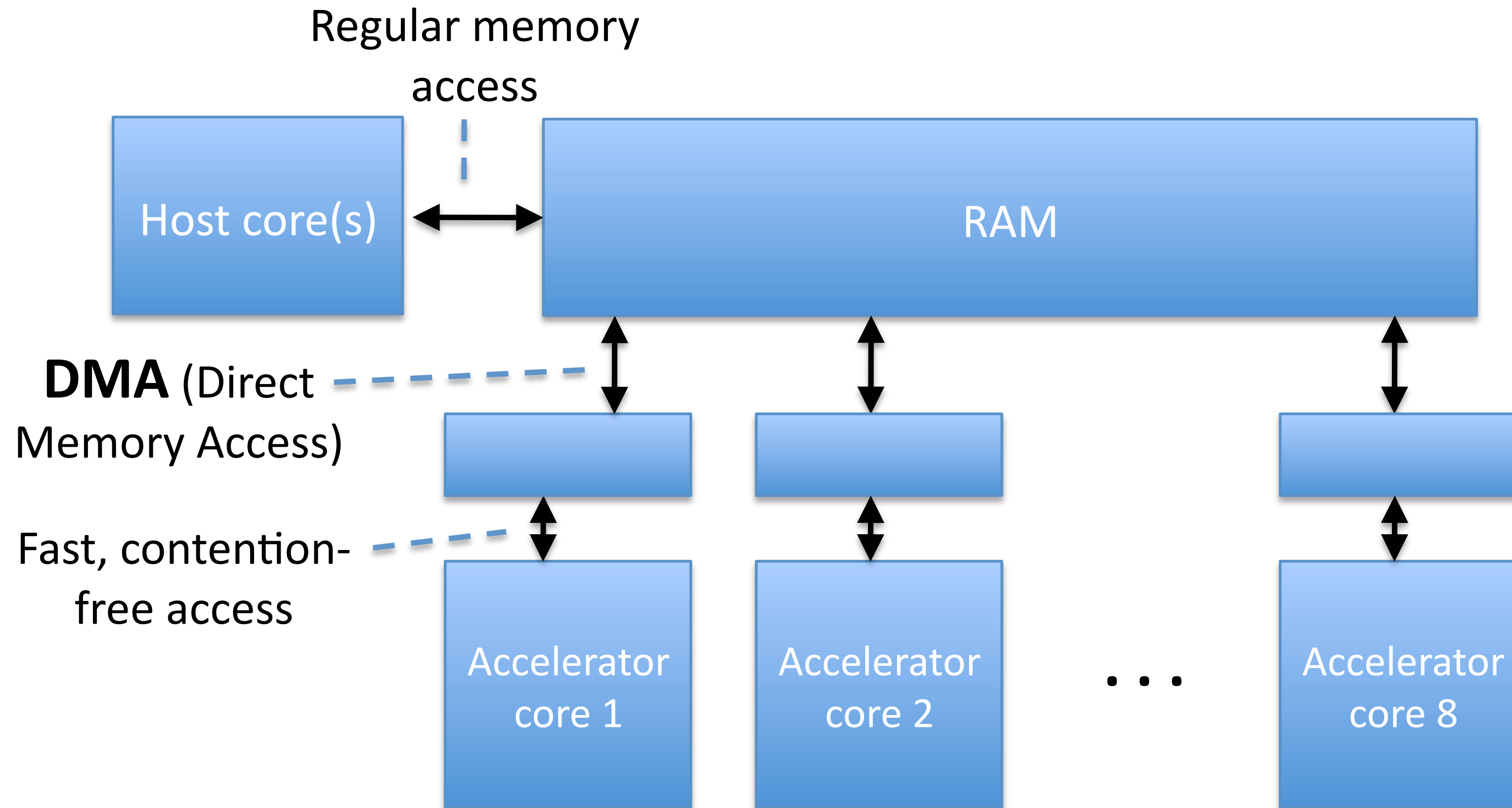
Scratch-pad memories



Cores have small, local memories, AKA "scratch-pad" memories

No coherency between memory spaces

Common pattern: host + accelerators

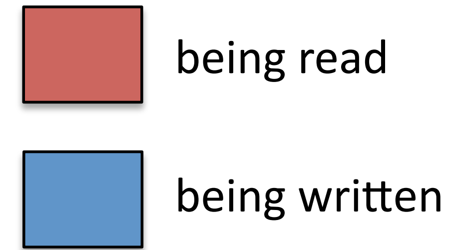


Cell BE processor: Power Processor Element host, Synergistic Processor Element accelerators

Problem, and our contribution

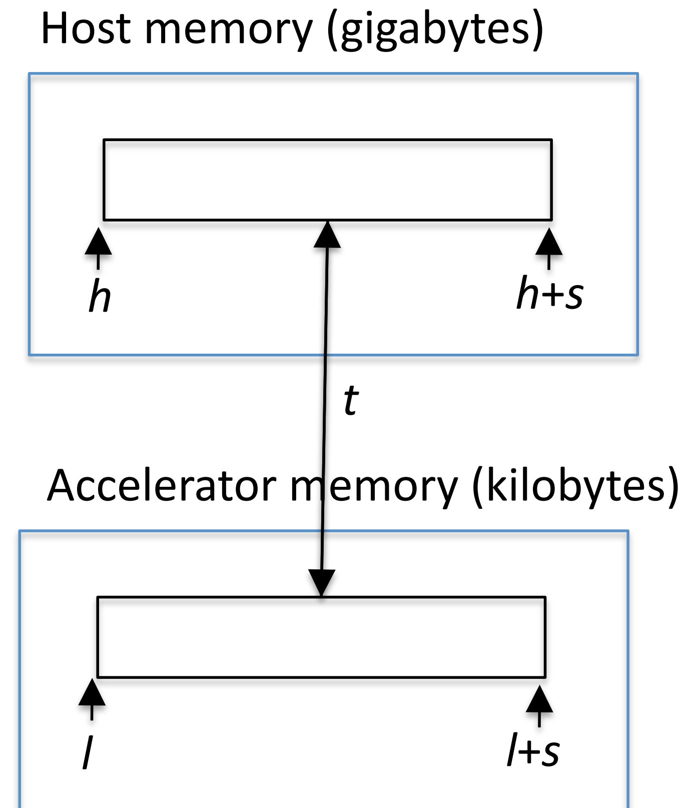
- Scratch-pad memories can lead to high-performance
- Price: programming complexity
- **Massive** scope for errors with DMA operations
- We apply formal verification to:
  - find bugs in use of DMA operations
  - prove DMA-correctness of programs
- Technology: bounded model checking, *k*-induction

DMA operations



Operations initiated by accelerator thread

put(l, h, s, t)
statements;
wait(t)
get(l, h, s, t)



DMAs execute asynchronously, handled by dedicated hardware

Concurrency: multiple threads, concurrent DMAs per thread

We focus on analysing DMA operations for one accelerator thread, in isolation

Note on DMAs

- System-defined max number of simultaneous DMAs: M

  $M = 32$ on Cell BE processor

Properties of interest

- Consider simultaneously pending operations:

  $op_1 (l_1, h_1, s_1, t_1)$; $op_2(l_2, h_2, s_2, t_2)$;

  where op_1, op_2 in { put, get }

disjoint(a_1, s$_1$, a_2, s$_2$)

- OK if the following holds:

  $(l_1 + s_1 \le l_2 \lor l_2 + s_2 \le l_1) \land (h_1 + s_1 \le h_2 \lor h_2 + s_2 \le h_1)$

- Refinement:

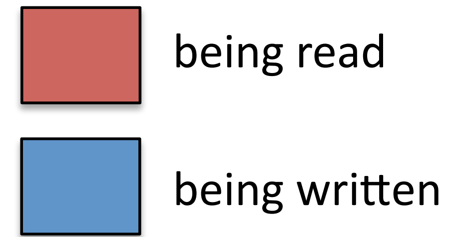
  $((op_1\text{=put} \land op_2\text{=put}) \lor \text{disjoint}(l_1, s_1, l_2, s_2)) \land$

  $((op_1\text{=get} \land op_2\text{=get}) \lor \text{disjoint}(h_1, s_1, h_2, s_2))$

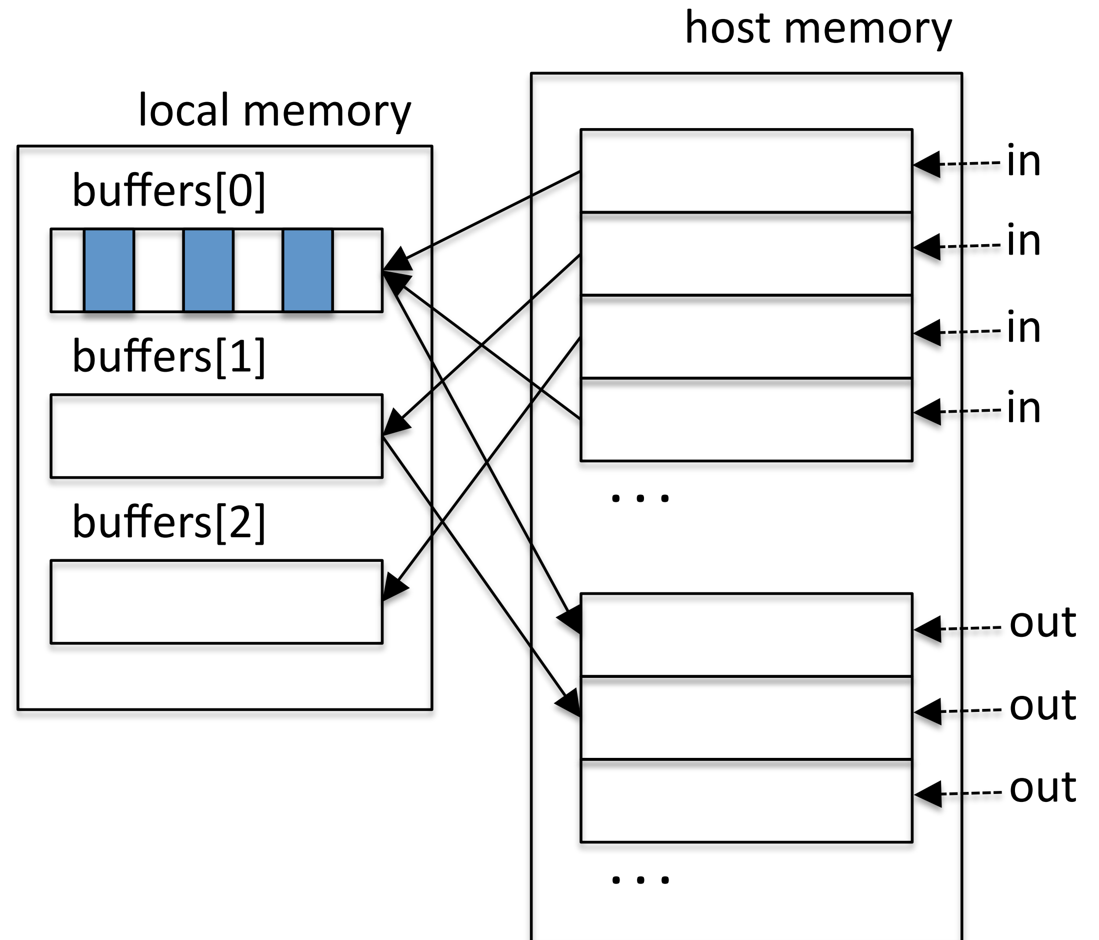
Example: triple buffering

```
#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
}
```

Illustration of bug




```
get(buffers[0], in, CHUNK, tags[0]);
in += CHUNK;
get(buffers[1], in, CHUNK, tags[1]);
in += CHUNK;
wait(tags[0]);
process_data(bufs[0]);
put(buffers[0], out, CHUNK, tags[0]);
out += CHUNK;
get(buffers[2], in, CHUNK, tags[2]);
in += CHUNK;
wait(tags[1]);
process_data(bufs[1]);
put(buffers[1], out, CHUNK, tags[1]);
out += CHUNK;
get(buffers[0], in, CHUNK, tags[0]);
```



Checking DMA races by program instrumentation

- *M*: max no. DMAs. Introduce tracker arrays, with size *M*:

  valid[*i*] = 1 if position *i* of arrays is tracking a DMA

  is_get[*i*] = 1 if *i*-th tracked DMA is a get

  local[*i*] = local store address for *i*-th DMA

  host[*i*] = host address for *i*-th DMA

  size[*i*] = size for *i*-th DMA

  tag[*i*] = tag for *i*-th DMA

Translate operations into updates to/expressions over trackers

May use *M* smaller than max no. DMAs

Translating get operations

get(l, h, s, t) becomes:

```
assert( !valid[0] ∨(disjoint (l, s, local[0], size[0]) ∧
        (is_get[0] ∨ disjoint (h, s, host[0], size[0]) ) ) );
…
assert( !valid[M-1] ∨(disjoint (l, s, local[M-1], size[M-1]) ∧
        (is_get[M-1] ∨ disjoint (h, s, host[M-1], size[M-1]) ) ) );
assert( !valid[0] ∨ …  !valid[M-1] );
pick i such that !valid[i];
valid[i] = 1; is_get[i] = 1; local[i] = l; host[i] = h; size[i] = s; tag[i] = t;
```

Translating wait operations

wait(t) becomes:

```
valid[0] = (valid[0] ∧ ! (t == tag[0]));
…
valid[M-1] = (valid[M-1] ∧ ! (t == tag[M-1]));
```

We do not model effects of get/put operations

Cell BE processor also supports "fence" and "barrier" operations – these can be translated with the addition of one more tracker array

Checking instrumented programs

- Trackers + translation designed so that assertion failure = DMA race (or too many DMAs)
- Translated C program can be fed to your favourite C prover
- **Model checking** very attractive as it is automatic and gives counterexamples

Suitable model checkers

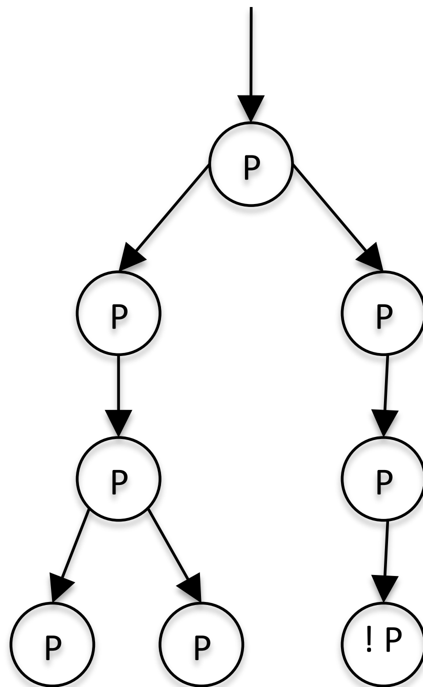
- Bounded model checkers, e.g. CBMC
  - Good for finding bugs
  - Cannot prove correctness alone.
- CEGAR-based:
  - SLAM, BLAST, SatAbs
  - Can prove correctness, but may not terminate
  - On our correct examples, these tools either do not work, or diverge
- Line of attack: use BMC + *k*-induction
- Much easier to check properties of local memory, so we concentrate on this (more later…)

The *k*-induction method

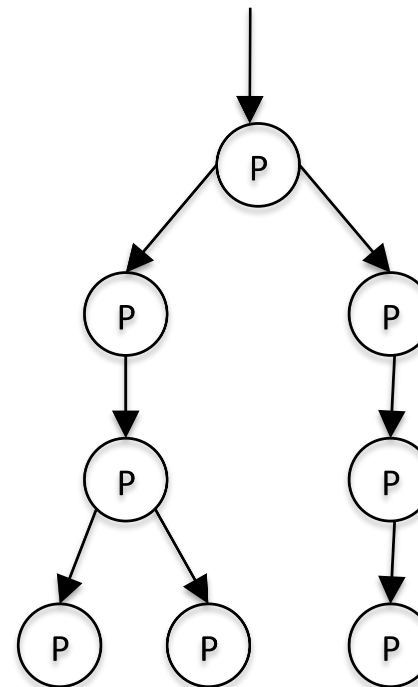
Due to Sheeran, Singh and Stålmarck, 2000

Safety property P

Pick k (*e.g*. k=3), do BMC up to depth *k*



Bug!
We are done



System OK up
to depth 3

Induction step

Consider paths of length *k* from *all potential states*

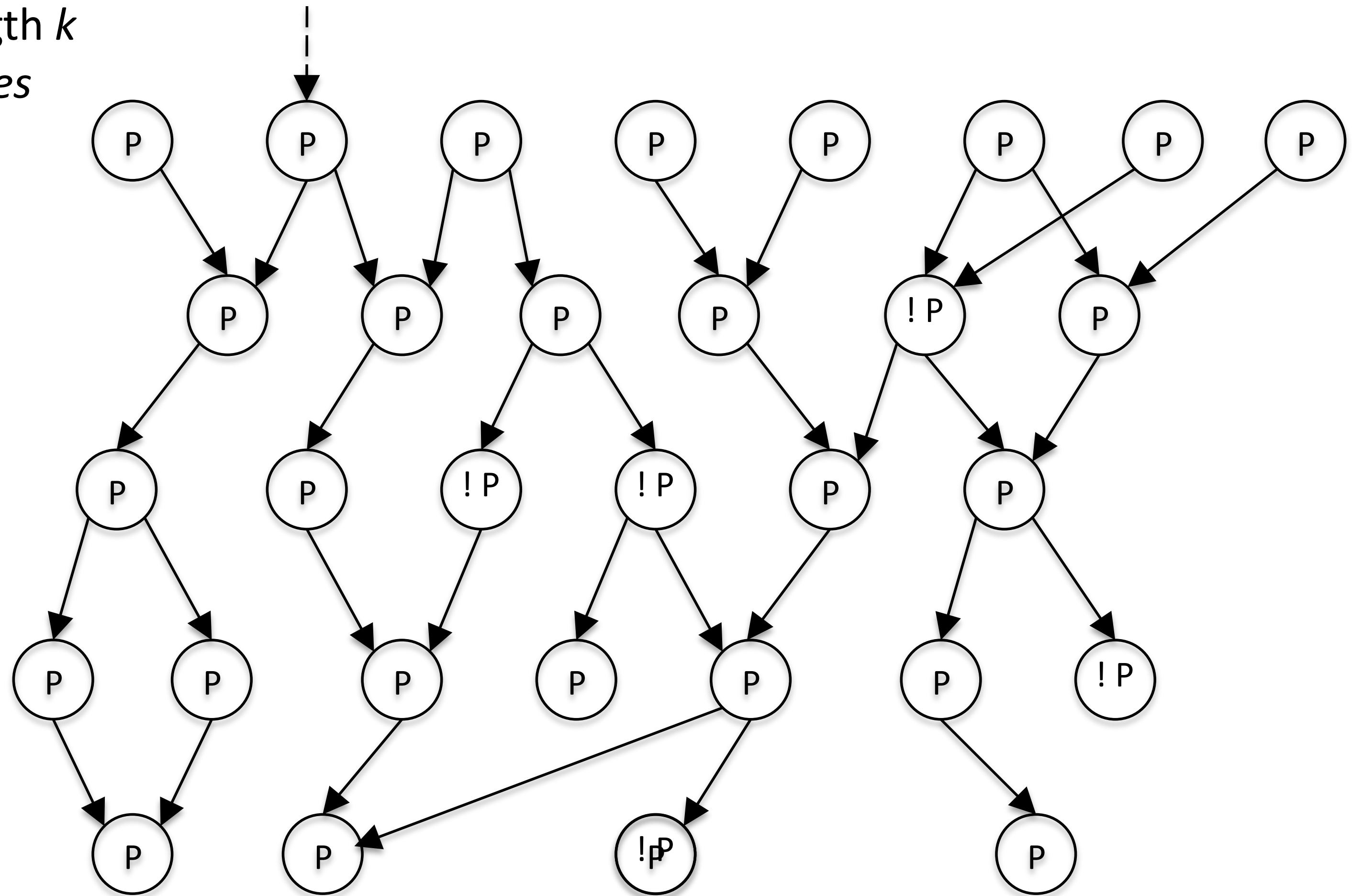
Eliminate paths where P fails

Check P for one further step

P holds in all new states? Then P holds in all reachable states

P does not hold in some state?

We know nothing… try larger *k*



k-induction in SAT-based BMC

Variables $x_1, \dots, x_n$ with finite domains $D_1, \dots, D_n$
Potential states: $S = D_1 \times D_2 \times \dots \times D_n$

Formulae: Initial states $I : S \to \{ true, false \}$
Transition relation $T : S \times S \to \{ true, false \}$
Property $P : S \to \{ true, false \}$

Base case:

$$I(s_0) \land T(s_0, s_1) \land \dots \land T(s_{k-1}, s_k) \land (!P(s_0) \lor \dots \lor !P(s_k))$$

If *satisfiable* then we have a bug + counterexample

Induction step:

$$T(s_0, s_1) \land \dots \land T(s_{k-1}, s_k) \land P(s_0) \land \dots \land P(s_k) \land T(s_k, s_{k+1}) \land !P(s_{k+1})$$

If *unsatisfiable* then, combined with base case, property holds
Otherwise try larger k

k-induction for programs

- Can think of program as monolithic transition function
  - given valuation of globals, heap, stack and program counter, program text tells us where to go next
- Why not just apply *k*-induction "out-of-the-box"?
  - *k*-induction works when state-graph exhibits repetition
  - quite common in hardware, less common in software
  - important optimisations for hardware do not work well for software (*e.g.* restriction to loop-free paths)

k-induction at loop level

Consider program consisting of single loop

Informally: check that we can successfully execute k iterations from initial state of program

Prove that, from an *arbitrary* state, given that we managed to execute *k* iterations successfully, we can execute one more successfully

assume(*e*) – execution trace is discarded if *e* is false

For program fragment β, assume(β) shorthand for:
"assert-replaced-with-assume in β"

k-induction at loop level

α ; $\underbrace{\text{assume}(e)\ ;\ \beta\ ;}_{j \text{ times}}$ assume(!e) ; γ *is correct* ($0 \leq j \leq k$)

$\underbrace{\text{assume}(e)\ ;\ \text{assume}(\beta)\ ;}_{k \text{ times}}$ assume(e); β *is correct*

$\underbrace{\text{assume}(e)\ ;\ \text{assume}(\beta)\ ;}_{k \text{ times}}$ assume(!e); γ *is correct*

α ; while(e) β ; γ *is correct*

This work well for DMA programs. Why?

- *k*-induction works well for sequential hardware circuits with pipelines
  - required *k* proportional to pipeline depth
- *k*-induction works well for DMA programs managing data using buffering
  - required *k* proportional to number of buffers
  - e.g. triple-buffering requires $k = 3$
- Additional heuristics can help
  - *e.g.* asserting no DMA pends forever

Experiments

- Implemented tool: **SCRATCH** – analyses source code for Cell BE processor, built on top of CBMC
- Applied tool to 22 benchmarks from IBM Cell SDK
- Manual program slicing required as examples use Cell intrinsics
- Buggy and correct versions, most bugs injected, **found two real bugs**

Experiments

- All bugs found in less than 5 seconds, except one which takes over 2 hours to find
  - difficult bug involves too many legal DMA operations
  - "no DMA pends forever" heuristic finds bug in less than 2 seconds
- Correctness proved in less than 16 seconds, maximum k required: 10
- Observed some examples for which *k*-induction does not work

The problem with host memory

```
Node * in; // refers to a list node in host memory
Node myNode, nextNode; // Local ‘Node’ variables

get(&myNode, in, sizeof(Node), t);
wait(t);
get(&nextNode, myNode.next, sizeof(Node), t);
process(&myNode);
put(&myNode, in, sizeof(Node), t);
```

Without any knowledge of host memory, must consider case where `in->next == in`, which leads to (probably spurious) DMA race

Could something like separation logic help?
E.g., if we are traversing a host-memory linked-list via local memory, specifying this may allow tractable verification

Summary

- Designed technique for detecting DMA races
- Implementation for Cell BE processor based on CBMC
- To prove correctness, re-formulated *k*-induction using loops
- Next steps:
  - Can loop-based formulation of *k*-induction be generally useful?
  - Can separation logic help verify properties of host memory?
  - Checking DMA interference between multiple threads
  - Investigate verification opportunities for OpenCL language