SCRATCH: a Tool for Automatic Analysis of DMA Races *

Alastair F. Donaldson Daniel Kroening Philipp Rümmer

Oxford University Computing Laboratory, Oxford, UK

{Alastair.Donaldson,Daniel.Kroening,Philipp.Ruemmer}@comlab.ox.ac.uk

Abstract

We present the SCRATCH tool, which uses bounded model checking and k-induction to automatically analyse software for multicore processors such as the Cell BE, in order to detect DMA races.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying, Verifying & Reasoning about Programs

General Terms Languages, Theory, Verification

Keywords Model checking, k-induction, Cell BE, DMA

1. Introduction

Multicore processors such as the Cell BE avoid the memory wall problem by equipping cores with local, "scratch-pad" memories. Direct Memory Access (DMA) is the mechanism used to transfer data between main and scratch-pad memories. A DMA operation requests that a contiguous chunk of memory be copied between memory spaces *asynchronously*: program execution can continue while the DMA operation is pending. This allows the latency associated with data movement to be hidden, by overlapping computation with communication using double- or triple-buffering. Correct programming of DMAs is notoriously difficult. *DMA races*, where two DMAs operate on the same region of memory and at least one modifies the memory, are easily introduced due to missing synchronization commands, and lead to nondeterministically occurring bugs that are hard to reproduce and fix.

We present SCRATCH,[1] a formal verification tool for automatic, static analysis of DMA races on scratchpad memory. SCRATCH operates on a C program for one of the Synergistic Processor Element (SPE) cores of the Cell BE processor, and uses a combination of SAT-based bounded model checking [2, 3] and k-induction [6] to detect, or prove absence of, DMA races on SPE local store. The tool uses a novel, implicit encoding of DMA operations, drawing on the notion of *prophecy* variables [1]. We describe this encoding and our use of k-induction, and show experimentally that SCRATCH is effective in DMA race analysis for a range of benchmarks.

Although implemented for the Cell platform, the techniques behind SCRATCH are general, and could easily be adapted to other multicore systems that use DMA.

* Supported by EPSRC grants EP/G051100 and EP/G026254/1, the EU FP7 STREP MOGENTES, and the EU ARTEMIS CESAR project.

[1] Tool and benchmarks available at `http://www.cprover.org/scratch`.

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.
ACM 978-1-4503-0119-0/11/02.

2. An implicit encoding of DMAs

SCRATCH uses four tracker variables to collaboratively track a single, *arbitrary* DMA operation. DMA races are detected by instrumenting the program so that on issue, a DMA is compared with the DMA currently tracked by the variables (if any). Because the tracked DMA is chosen arbitrarily from the set of previously issued DMAs, any potential DMA race will be detected by symbolic execution of the instrumented program up to a sufficient depth. We call this encoding *implicit*: a DMA is implicitly compared with all prior DMAs; no explicit history of active DMAs is required. The tracker variables are: `valid`, a flag which is *false* if no DMA operation is tracked, and *true* otherwise; `addr`, `sz` and `tag`, which record the local store address, size and identifying tag for a pending DMA operation if `valid`=*true*, and whose values are irrelevant if `valid`=*false*. Initially, `valid` is set to *false*, and the other tracker variables are nondeterministically assigned.

A DMA `get` operation has the form `get(`l`,`h`,`s`,`t`)`, and requests a data transfer of s contiguous bytes from host memory region $[h, h + s)$ to local memory region $[l, l + s)$. The operation is identified by tag t (a value in the range $[0..ws - 1]$, where ws is the processor word-size); this tag can be used subsequently to synchronize with the `get` operation. The size parameter s must be less than max, a hardware-imposed maximum transfer size. The SCRATCH instrumenter replaces a `get` operation with the following code:

```
(1)  assert((unsigned)t < ws && (unsigned)s < max);
(2)  assert(!valid || l+s <= addr || addr+sz <= s);
(3)  memset(l,*,s);
(4)  if(*) { valid = true; addr = l; sz = s; tag = t; }
```

Statement (1) ensures that the size and tag associated with the operation are within the range permitted by the hardware. Statement (2) checks that if the tracker variables are tracking a DMA operation d, the new DMA operation does not locally race with d. Statement (3) over-approximates the effect of the DMA by modelling transfer of *arbitrary* data from host memory. Here $*$ denotes a nondeterministic value, and the intended semantics is that $*$ is evaluated separately for each byte in the region $[l, l + s)$. The statements labelled (4) assign details of the new DMA operation to the tracker variables, setting `valid` to *true*. By nondeterministically guarding these statements, we ensure that an *arbitrary* DMA operation is tracked in the instrumented program. A DMA `put` operation is dual to a `get` operation and is encoded similarly.

A DMA `wait` operation has the form `wait(`$mask$`)`, and causes execution to block until all DMA operations identified by tag i have completed, for each $0 \leq i < ws$ such that $mask[i] = 1$. A `wait` operation is encoded by SCRATCH as follows:

```
assume( ((1<<tag)&mask ) == 0 );
```

If `assume(`e`)` is executable with e=*false*, the current execution trace is discarded; otherwise the statement is a no-op. Rather than handling a `wait` by explicitly erasing details of a DMA if it is identified by a tag whose bit is set in $mask$, the encoding simply decides that no DMA with this property was tracked in the first place! The

instrumentation variables can be regarded as *prophetic* [1] since the future program execution determines whether a DMA operation should be tracked.

SCRATCH handles the full range of DMA operations supported by the Cell architecture, including fence, barrier and list operations, via appropriate adaptations of the above instrumentation.

Comparison with an existing, explicit encoding. An alternative encoding of DMA operations for race analysis has been presented [4]. This *explicit* encoding mirrors the runtime monitoring approach employed by the IBM Race Check library [5], recording a bounded history of D pending DMA operations (for some $D > 0$), and checking new DMA operations against the history log. Although more intuitive than our new implicit encoding, the explicit encoding is less efficient when SAT-based symbolic model checking is used for analysis, since the size of the representation is proportional to the parameter D, which may be large for complex examples. Furthermore, the explicit encoding cannot handle programs where the number of DMA operations that may be issued simultaneously is unbounded; our new encoding does not suffer from this restriction.

3. SAT-based model checking with k-induction

SCRATCH builds on the SAT-based model checker CBMC [3], which allows instrumented programs to be symbolically executed up to a given depth, and bit-blasted to yield a SAT representation for which satisfying assignments correspond to DMA races. This facilitates bug-finding, but cannot prove *absence* of DMA races. To achieve this latter goal, SCRATCH combines SAT-based analysis with k-induction [6], applying k-induction directly to program loops. To verify unbounded absence of DMA races for an instrumented program of the form α; `while(`c`)` $\{\beta\}$ γ, SCRATCH solves a series of verification problems using bounded model checking. For increasing values of k, starting with $k = 0$, a base case and a step case are checked:

Base case: $\alpha;\ \underbrace{\texttt{if}(c)\ \{\beta\}\ldots\texttt{if}(c)\ \{\beta\}}_{k \text{ times}}\ \texttt{if}(\texttt{!}c)\ \{\gamma\}$

Step case: $\texttt{havoc};\ \underbrace{\texttt{assume}(c);\ \overline{\beta};\ldots;\ \texttt{assume}(c);\ \overline{\beta};}_{k \text{ times}}\ \texttt{if}(c)\ \{\beta\}\ \texttt{else}\ \{\gamma\}$

The base case consists of the loop unwound k times. A base case failure yields a counterexample exposing a DMA race; otherwise we know that a DMA race cannot occur within k loop iterations.

In the step case, `havoc` sets every program variable to a nondeterministic value, and $\overline{\beta}$ denotes the sequence β with `assert` replaced by `assume` throughout. The step case succeeds when, from *any* potential state, if k loop iterations can be executed without encountering a DMA race then a further iteration can be executed (if c still holds), or the loop epilogue can be executed (if c does not hold), without a DMA race occurring. If there is some k for which both the base case and step case hold, the theory of k-induction guarantees that a DMA race can never occur. SCRATCH tries k-induction with increasing values for k until a result is obtained, or a "give up" value for k is reached. Nested loops are handled via translation to a single, monolithic loop.

We find k-induction works well in our application domain because DMA operations in loops are typically designed to be pending for only a bounded number of loop iterations. This often allows k-induction to succeed with a value of k proportional to the bound.

4. Experimental evaluation

Benchmarks. We evaluate SCRATCH using a set of 22 benchmarks adapted from examples supplied with the IBM Cell SDK [5]. The benchmarks include a variety of data processing programs, using single-, double- or triple-buffering for data-movement; two audio processing applications; two particle simulations using Euler integration; and quaternion Julia set ray-tracing.

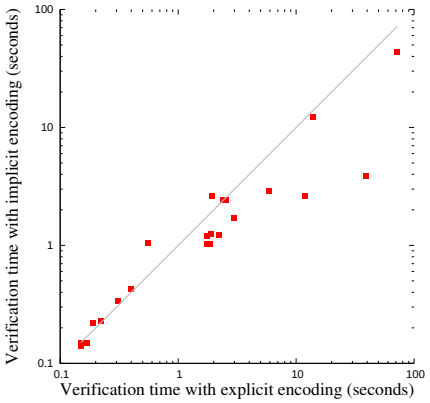

Figure 1. Verification times for correct benchmarks using an explicit encoding of DMAs [4] and our new, implicit encoding

We apply SCRATCH to correct and buggy versions of the benchmarks. In two cases we found genuine bugs. For the remaining benchmarks, bugs are injected into the examples, either by removing a `wait` operation, changing the tag used to identify a DMA, or switching an operation between `get` and `put`. Experiments are performed on a 3GHz Intel Xeon platform with 48Gb RAM, running Ubuntu. MiniSat 2.0 is used as a back-end SAT solver.

Bug-finding. Bounded model checking proves extremely effective for detecting DMA races in buggy versions of our benchmarks. With our new encoding, the maximum verification time across all benchmarks (averaged over multiple runs) is 0.87s. Performance is slightly better than with the explicit encoding of [4], a speedup of $1.52\times$ is achieved using our new encoding in the best case.

Proving race freedom. Each point in the scatter plot of Figure 1 represents one of our benchmarks; its x- and y-coordinates show the time (in seconds) taken to verify the correct version of this benchmark using the explicit and implicit encoding of DMA operations, respectively. Points appearing below the diagonal line correspond to benchmarks where our new encoding is faster. The results show that k-induction provides a tractable method for proving absence of races. Sometimes the implicit encoding significantly outperforms the explicit encoding: verification with the implicit encoding is 10 times faster for one of the *Euler* examples. In this benchmark, an SPE can issue 10 concurrent DMAs. The explicit encoding tracks all these DMAs, leading to a large SAT instance, while the largest SAT instance for the implicit encoding, which is independent of the number of issued DMAs, is almost $10\times$ smaller.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[3] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, pages 168–176, 2004.

[4] A. F. Donaldson, D. Kroening, and P. Rümmer. Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In *TACAS*, pages 280–295, 2010.

[5] IBM. Cell BE resource center, October 2009. `http://www.ibm.com/developerworks/power/cell/`.

[6] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, pages 108–125, 2000.