Analysing DMA Races in Multicore Software

Alastair F. Donaldson, Daniel Kroening, Philipp Rümmer[*]
Oxford University Computing Laboratory, Oxford, UK

Abstract

We present ongoing work on applying model checking techniques to automatically analyse multicore software where data is managed explicitly by the programmer via direct memory access (DMA) operations. We describe SCRATCH, a DMA race analysis tool for the Cell BE processor which employs bounded model checking and k-induction. We then outline our plans to extend this work.

1 Introduction

In this position paper, we present ongoing work on applying model checking techniques to automatically verify software for multicore processors. Our focus is on multicore architectures with multiple memory spaces, where it is the programmer's responsibility to orchestrate movement of data between memories in an efficient manner, to achieve high performance. Such architectures are extremely difficult to program correctly, and there is great scope for formal verification techniques to be of use.

The Cell Broadband Engine (BE) architecture [5] consists of a host core, the Power Processor Element (PPE), and a number of accelerator cores, the Synergistic Processor Elements (SPEs). The PPE is a regular processor connected to main memory, whereas each SPE is equipped with a private, 256 K "scratch-pad" memory, also known as *local memory*. The SPE local memories are not coherent with each other, or with main memory. As a result, an SPE can access its local memory very efficiently, without contention. However, an SPE thread can only access main memory using direct memory access (DMA). A DMA operation is a request to copy a chunk of data between host and local memory, and is handled asynchronously by dedicated hardware. High performance can be achieved by organising DMA operations so that computation and communication are overlapped, via buffering techniques. The price for performance is programming complexity: writing data-movement code is error-prone. Programmer errors related to DMA operations can result in memory corruption, due to multiple DMA operations simultaneously accessing the same memory (DMA races). Errors due to misuse of DMA can result in nondeterministic bugs that are difficult to consistently reproduce and fix.

In prior work, we have designed a tool, SCRATCH, to detect, or prove absence of, DMA races in SPE programs [4]. The tool uses a combination of bounded model checking [2] and k-induction [9] to automatically analyse SPE threads in isolation, checking for DMA races on local memory. After describing DMA operations in more detail (§2) we summarise our existing work (§3). We then discuss plans to extend this work (§4). Due to lack of space, we do not discuss related work in detail. Please refer to [4] for such a discussion.

2 Direct memory access operations

We consider three DMA primitives: $\mathsf{get}(l,h,s,t)$, which issues a transfer of s bytes from host memory address h to local memory address l, and is identified by tag t; $\mathsf{put}(l,h,s,t)$, which analogously transfers data from local to host memory; and $\mathsf{wait}(t)$, which blocks until all DMA operations identified by tag t have completed. On the Cell BE processor a tag is an integer in the range $0,\ldots,31$, and it is legal for

[*]Alastair F. Donaldson is supported by EPSRC grant EP/G051100. Daniel Kroening and Philipp Rümmer are supported by EPSRC grant EP/G026254/1, the EU FP7 STREP MOGENTES, and the EU ARTEMIS CESAR project.

```
#define CHUNK 16384 // Process data in 16K chunks

float buffers[3][CHUNK/sizeof(float)]; // Triple-buffering requires 3 buffers

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

/* 'in' and 'out' are pointers to host memory */
void triple_buffer(char* in, char* out, int num_chunks) {
    unsigned int tags[3] = { 0, 1, 2 }, tmp, put_buf, get_buf, process_buf;

(1) get(buffers[0], in, CHUNK, tags[0]); // Get triple-buffer scheme rolling
    in += CHUNK;
(2) get(buffers[1], in, CHUNK, tags[1]);
    in += CHUNK;
(3) wait(tags[0]); process_data(buffers[0]); // Wait for and process first buffer
    put_buf = 0; process_buf = 1; get_buf = 2;
    for(int i = 2; i < num_chunks; i++) {
(4)   put(buffers[put_buf], out, CHUNK, tags[put_buf]); // Put data processed
      out += CHUNK;                                     //   last iteration
(5)   get(buffers[get_buf], in, CHUNK, tags[get_buf]);  // Get data to process
      in += CHUNK;                                      //   next iteration
(6)   wait(tags[process_buf]);              // Wait for and process data
      process_data(buffers[process_buf]);   //   requested last iteration

      tmp = put_buf; put_buf = process_buf; // Cycle the buffers
      process_buf = get_buf; get_buf = tmp;
    }
    ... // Handle data processed/fetched on final loop iteration
}
```

Figure 1: Triple-buffering example, adapted from an example provided with the IBM Cell SDK [6]

one SPE to issue up to 32 concurrent DMAs (thus each DMA can, in principle, be identified by a distinct tag). Note that DMA operations are always issued from the point of view of an accelerator core (SPE), *e.g.* get always denotes movement of data into local memory. It is usual for all DMA operations to be initiated by the SPE cores, and we restrict our attention to this scenario.

Two DMA operations are said to *race* if they are pending simultaneously, operate on a common region of (host or local) memory, and at least one modifies this region.

Example: triple-buffering. Figure 1, adapted from an example provided with the IBM Cell SDK [6], is part of an SPE program, and illustrates the use of DMA operations to stream data from host memory to local store to be processed, and to stream results back to host memory. Triple-buffering is used to overlap communication with computation: each iteration of the loop in `triple_buffer` puts results computed during the previous iteration to host memory, gets input to be processed next iteration from host memory, and processes data which has arrived in local memory.

If `num_chunks` is greater than three, this example exhibits a local memory DMA race, which we can observe by logging the first six DMA operations. To the right of each operation we record its source code location and, if appropriate, its loop iteration. We omit host address parameters, irrelevant to the race:

```
      get(buffers[0], ..., CHUNK, tags[0])   (1)
      get(buffers[1], ..., CHUNK, tags[1])   (2)
      wait(tags[0])                          (3)
(*)   put(buffers[0], ..., CHUNK, tags[0])   (4), i=2
      get(buffers[2], ..., CHUNK, tags[2])   (5), i=2
      wait(tags[1])                          (6), i=2
      put(buffers[1], ..., CHUNK, tags[2])   (4), i=3
(*)   get(buffers[0], ..., CHUNK, tags[0])   (5), i=3
```

At this point in execution the operations marked (*) race with one another: they operate on the same local memory, the second operation modifies the memory, and is not protected by an intervening wait. The race can be avoided by inserting a wait with tag `tags[get_buf]` before the get at (5).

We discovered this bug using SCRATCH, our automatic DMA analysis tool, which can also show that the fix is correct. The bug occurs in an example provided with the IBM Cell SDK, and was, to our knowledge, previously unknown. Our bug report has been confirmed by an engineer at IBM.

3 Scratch: an automatic DMA race analyser for Cell BE software

SCRATCH (so called because it analyses scratch-pad memory) takes as input a C program written for one of the SPE cores of the Cell BE processor. The program is transformed so that DMA operations are replaced with statements to check for local memory races. In principle, the transformed program can be analysed by any tool capable of checking assertions in C programs augmented with assume statements and nondeterministic choice. In practice, SCRATCH is built on top of the bounded model checker CBMC [3], which unwinds the transformed program to check for DMA races up to a user-specified depth, using SAT techniques.

Translating DMA statements. This is achieved via an array of *DMA entry* records, called the *tracker* array. A DMA entry represents a pending DMA operation, and has the form $(valid, local, size, tag)$. The *valid* field is a bit determining whether the entry represents a pending DMA, or is unused. If $valid = 1$ then the remaining fields store the local address, size, and tag associated with the DMA, otherwise they are ignored. Note that host memory locations are *not* tracked, since SCRATCH does not currently analyse DMA races on host memory; we discuss this further in §4. The size of the tracker array is, by default, 32, the maximum number of DMAs which may be simultaneously issued by an SPE.

At the start of the program, *valid* is set to 0 for each DMA entry in the tracker array.

A DMA command of the form $\mathsf{op}(l,h,s,t)$, where op is get or put, is translated into:

1. $\mathsf{assert}([l, l+s) \cap [local, local+size) = \emptyset)$ for all DMA entries matching the pattern $(1, local, size, \_)$, *i.e.* the new DMA does not operate on the same local memory as any pending DMA;
2. an assertion that some DMA matches the pattern $(0, \_, \_, \_)$, *i.e.* at least one entry is not valid;
3. a statement replacing a nondeterministically chosen DMA entry matching the pattern $(0, \_, \_, \_)$ with $(1, l, s, t)$, *i.e.* an entry for the new DMA operation is added to the tracker array.

A DMA wait operation of the form $\mathsf{wait}(t)$ is translated into statements that replace any DMA entry matching the pattern $(1, \_, \_, t)$ with $(0, \_, \_, t)$.

In the translated program, issuing a DMA that races with an already pending DMA results in an assertion failure due to 1 above; an attempt to issue more than the maximum number of allowed DMAs also results in a failed assertion due to 2. The above translation is actually too strict: it prohibits concurrent, overlapping put operations, which cannot lead to local memory races. To avoid this limitation, each DMA entry is extended with a flag indicating whether an operation is a put, and the assertion in 1 is modified to ignore simultaneous put operations.

Proving absence of DMA races. While bounded model checking is good at finding bugs, it cannot be used in isolation to prove the absence of bugs. As well as detecting DMA races, we are interested in proving their absence. SCRATCH achieves this goal on many practical examples using a novel formulation of k-induction, a technique first introduced in [9].

To verify absence of DMA races for a transformed program consisting of a single *while* loop with prologue α, condition c, body β and epilogue γ, where β does not contain nested loops, 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:** α; $\underbrace{\mathsf{if}(c)\ \{\beta\} \ldots \mathsf{if}(c)\ \{\beta\}}_{k \text{ times}}$ $\mathsf{if}(!c)\ \{\gamma\}$
- **Step case:** $\mathsf{havoc};\ \underbrace{\mathsf{assume}(c);\ \beta_{\mathsf{assume}}; \ldots;\ \mathsf{assume}(c);\ \beta_{\mathsf{assume}}}_{k \text{ times}};\ \mathsf{if}(c)\ \{\beta\}\ \mathsf{else}\ \{\gamma\}$

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. For a boolean expression e, assume(e) at program point p tells the model checker to cease exploring an execution trace if e does not hold at p; β_{assume} denotes the sequence β with assert replaced by assume throughout. The step case succeeds if, from *any* potential state, if it is possible to execute k loop iterations without encountering a DMA race then it must be possible to execute one more iteration (if c still holds), or execute the loop epilogue (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. If the base case holds but the step case fails, a larger value of k must be considered. Our formulation of k-induction is the first to operate at the loop level; other presentations of the technique work at a finer granularity by unwinding the transition relation.

There is no guarantee that k-induction will terminate with a conclusive result for a feasibly small value of k. However, we find that the technique works well in practice for DMA race analysis. Intuitively, this is because DMA operations in loops are typically designed to be pending for only a bounded number of loop iterations, allowing k-induction to succeed with a value of k proportional to the bound. This is analogous to the intuition that k-induction works well for sequential hardware circuits with pipelines, where the k required for induction to succeed is proportional to the pipeline depth [1].

Experimental summary. We have evaluated SCRATCH using a set of 22 benchmarks, adapted from the IBM Cell SDK [6]. For correct and buggy version of each example, SCRATCH is able to find the DMA race, or prove absence of DMA races. Bug finding is fast, as one would expect from bounded model checking. More interestingly, for 15 of the benchmarks, correctness can be proved using k-induction in less than 10 seconds on a state-of-the-art platform, with $k \leq 5$ in all cases. One benchmark requires a verification time of 7 minutes with $k = 10$. A full discussion of experimental results is given in [4], including a comparison with model checking approaches based on predicate abstraction, and with a run-time race checking tool from IBM.

4 Further opportunities for formal verification

We plan to extend this work in several ways.

Multi-loop k-induction. SCRATCH can currently only employ k-induction to programs consisting of a *single*, non-nested loop. While we were able to verify many interesting examples with this restriction (in some cases by manually slicing away inner data-processing loops) the limitation is, in general, rather restrictive. It is always possible to transform a nest of loops into one monolithic loop, using a program counter variable and conditional statements to simulate the nesting. We have applied this transformation manually on Cell BE examples containing one nested loop, and found that k-induction succeeds, thus we plan to automate the transformation and explore larger examples. The drawback is that if one nested loop would require a large value of k to prove correct in isolation then this value will dominate the necessary unwinding of the (potentially very large) monolithic loop. Alternatively, if k-induction is viewed as a proof rule operating on loops, as in [4], the proof rule can be applied recursively to a loop nest. This approach involves solving a possibly large series of verification problems, but has the advantage that each problem can be solved using the smallest possible k. We plan to explore, and compare, both approaches.

Other extensions related to *k*-induction include using abstract interpretation to strengthen loop invariants, and exploring the success of *k*-induction in software verification more generally.

Inter-thread interference. SCRATCH currently checks for DMA races on local memory only. This simplifies the verification problem, allowing the tool to analyse a single SPE thread in isolation. We are keen to tackle the more challenging problem of checking for interference between threads running on separate cores. For the Cell BE processor this corresponds to checking for DMA races on host memory. One approach to verifying these kinds of properties is to use abstract interpretation to under-/over-approximate the memory regions accessed by individual threads, determining presence/absence of DMA races if these regions do/do not overlap. A related, but possibly more precise strategy for proving non-interference is to derive invariants for the individual threads (or an invariant for the system) that imply the absence of DMA races. Such invariants could be inferred by means of interpolation-based model checking [8]: showing via bounded model checking that threads do not interfere up to a certain search depth, using the unsatisfiability proof of the BMC formula to derive an interpolant, and iterating the BMC/interpolant-computation process until an interpolant is computed that is an inductive invariant.

Pointer validity and alignment. The Cell BE processor requires that source and target pointers for DMA operations are aligned to appropriate byte boundaries (16 for correctness, 128 for efficiency). Misalignment is a common source of DMA errors, and it can be hard to track down the cause of a misaligned pointer. In addition, the lack of separation between host and local pointers at the type level means that errors can occur, *e.g.* if the programmer accidentally passes a host pointer where a local pointer is required. We plan to consider two solutions to this problem: using type reconstruction to infer alignment and memory space properties of pointers, and tracking pointer information during bounded model checking, automatically adding assertions to check for invalid or misaligned pointer parameters to DMA operations.

Support for OpenCL. OpenCL [7] is a new, open standard language for programming heterogeneous multicore architectures including the Cell BE, and graphics processing units. OpenCL includes asynchronous memory copy primitives, which are similar to DMA transfers: the same programming problems arise, and it appears that similar solutions should be applicable. Thus, we plan to build a verification tool for OpenCL programs based on the technology behind SCRATCH.

References

[1] Roy Armoni, Limor Fix, Ranan Fraer, Scott Huddleston, Nir Piterman, and Moshe Y. Vardi. SAT-based induction for temporal safety properties. *Electr. Notes Theor. Comput. Sci.*, 119(2):3–16, 2005.

[2] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[3] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[4] Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer. Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In *TACAS*, LNCS. Springer, 2010. To appear.

[5] H. Peter Hofstee. Power efficient processor architecture and the Cell processor. In *HPCA*, pages 258–262. IEEE Computer Society, 2005.

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

[7] Khronos Group. The OpenCL specification. http://www.khronos.org/opencl.

[8] Kenneth L. McMillan. Interpolation and SAT-based model checking. In *CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.

[9] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, volume 1954 of *LNCS*, pages 108–125. Springer, 2000.