Noname manuscript No.
(will be inserted by the editor)

Automatic Analysis of DMA Races Using Model Checking and k-induction

Alastair F. Donaldson · Daniel Kroening · Philipp Rümmer

Received: date / Accepted: date

Abstract Modern multicore processors, such as the Cell Broadband Engine, achieve high performance by equipping accelerator cores with small "scratch-pad" memories. The price for increased performance is higher programming complexity – the programmer must manually orchestrate data movement using direct memory access (DMA) operations. Programming using asynchronous DMA operations is error-prone, and *DMA races* can lead to non-deterministic bugs which are hard to reproduce and fix. We present a method for DMA race analysis in C programs. Our method works by automatically instrumenting a program with assertions modeling the semantics of a memory flow controller. The instrumented program can then be analyzed using state-of-the-art software model checkers. We show that bounded model checking is effective for detecting DMA races in buggy programs. To enable automatic verification of the correctness of instrumented programs, we present a new formulation of k-induction geared towards software, as a proof rule operating on loops. Our techniques are implemented as a tool, SCRATCH, which we apply to a large set of programs supplied with the IBM Cell SDK, in which we discover a previously unknown bug. Our experimental results indicate that our k-induction method performs extremely well on this problem class. To our knowledge, this marks both the first application of k-induction to software verification, and the first example of software model checking in the context of heterogeneous multicore processors.

Keywords Model checking · k-induction · DMA · multicore programming · Cell BE

This paper is a revised and extended version of [DKR10]. 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.

Alastair F. Donaldson
Department of Computer Science, University of Oxford
E-mail: alastair.donaldson@cs.ox.ac.uk

Daniel Kroening
Department of Computer Science, University of Oxford
E-mail: kroening@cs.ox.ac.uk

Philipp Rümmer
Department of Information Technology, Uppsala University
E-mail: philipp.ruemmer@it.uu.se

1 Introduction

Since the late 1950s until early the early 2000s, high-performance computing users (*e.g.* scientists using computers to study complex physical phenomena) have been able to simply wait for consecutive generations of computer hardware to significantly speed up the performance of applications. This was due to the success of *frequency scaling*, facilitated by Moore's law, which states that the number of transistors that can be inexpensively placed on an integrated circuit will double approximately every two years [Moo98].

Although transistor densities continue to double every 18 to 24 months, further increases in processor frequency have been found to lead to prohibitively high levels of power consumption. Instead of increasing the frequency of individual processor cores, manufacturers have opted to deliver performance by using the additional transistors afforded by Moore's law to design processors consisting of *multiple* cores.

In principle, a *homogeneous* multicore processor, consisting of n identical cores which share memory, can offer a factor of n times execution speedup over a single-core processor running at the same clock rate. However, such speedups are rarely achieved for realistic applications, for two main reasons: 1) it may not be possible to partition an application into independent components for parallel execution, and 2) contention for access to shared memory in a data-intensive parallel application may lead to a performance bottleneck. The second problem, known as the *memory wall*, means that even for highly parallel applications, adding further processor cores quickly leads to diminishing returns.

Heterogeneous multicore processors, such as the Cell Broadband Engine (BE) [Hof05, IBM09], circumvent the memory wall problem by equipping cores with small "scratch-pad" memories. These fast, private memories are not coherent with main memory, thus allowing independent calculations to be processed in parallel by separate cores without contention. Movement of data between distinct memory spaces is under software control, and can be coordinated using *direct memory access* (DMA) operations, combined with mailbox/interrupt facilities for inter-core synchronization. A common design for heterogeneous multicore architectures consists of a host core, connected to main memory, together with a number of accelerators each equipped with scratchpad memory. This situation is illustrated in Figure 1.

While the use of scratch-pad memory can boost performance, it places heterogeneous multicore programming at the far end of the concurrent programming spectrum. The programmer can no longer rely on the hardware and operating system to seamlessly transfer data between the levels of the memory hierarchy, and must instead manually orchestrate data movement between memory spaces using *direct memory access* (DMA). Low-level data movement code is error-prone: misuse of DMA operations can lead to *DMA races*, where concurrent DMA operations operate on the same portion of memory, and at least one modifies the memory. If undetected, DMA races can lead to nondeterministic bugs that are difficult to reproduce and fix.

We present a method for DMA race analysis which automatically instruments a C program containing DMA operations with assertions modeling the semantics of a memory flow controller. We consider two different encodings of DMA operations. The first encoding, initially presented in [DKR10], *explicitly* tracks a bounded history of pending DMA operations. DMA races are then detected by comparing each new DMA operation with every pending DMA operation in the history. The second is a novel encoding, sketched in [DHK11], where a single (nondeterministically selected) pending operation is tracked, against which future operations are compared. We call this the *implicit* encoding, because each DMA operation is implicitly compared with every pending DMA operation via a single, arbitrary pending operation. The implicit encoding resembles the concept of *prophecy variables* [AL91] and

permits analysis of programs where an unbounded number of DMA operations may be issued. We show how these encodings can be extended to handle *fence* and *barrier* operations which are supported by architectures such as the Cell BE; the handling of these features is not discussed in [DKR10, DHK11].

The instrumented programs are amenable to automatic verification by state-of-the-art model checkers. A DMA race involves a pair of DMAs either issued by separate threads, or by a single thread. In this paper, we restrict attention to the latter scenario: we focus on analyzing a thread program in isolation, to determine whether the thread can issue simultaneous DMA operations that race with one another. This is an important contribution, since correctly programming a single accelerator thread to issue correct sequences of DMAs is already a significant challenge. Furthermore, this restriction enables scalability: recent dramatic advances in SAT/SMT techniques have led to widespread use of bounded model checking (BMC) [BCC$^+$03, CKL04] for finding bugs in sequential software. We show experimentally that applying BMC to instrumented programs yields an effective strategy for detecting DMA races.

As well as detecting DMA races, we are interested in proving their *absence*. However, BMC is only complete if the bound exceeds a completeness threshold [KS03] for the property under consideration, which is often prohibitively large. We overcome this limitation by presenting a novel formulation of k-induction [SSS00]. The k-induction method has been shown effective for verifying safety properties of hardware designs. In principle, k-induction can be applied to software by encoding a program as a monolithic transition function. This approach has not proven successful due to the loss of control-flow structure associated with such a naïve encoding, and because important refinements of k-induction (*e.g.* restriction to loop-free paths) are not useful for software where the state-vector is very large.

We present a general proof rule for k-induction that is applicable to imperative programs with loops, and prove correctness of this rule. In contrast to the naïve encoding discussed above, our method preserves the program structure by operating at the loop level. Furthermore, it allows properties to be expressed through assertion statements rather than as explicit invariants. Our experimental results indicate that this method of k-induction performs very well when applied to realistic DMA-based programs, which use double- and triple-buffering schemes for efficient data movement. Such programs involve regularly-structured loops for which k-induction succeeds with a relatively small k.

Experimental evaluation is performed using an implementation of our techniques as a tool, SCRATCH, which checks programs written for the Synergistic Processor Element (SPE) cores of the Cell BE processor. We present an evaluation of SCRATCH using a set of 22 example programs provided with the IBM Cell SDK for Multicore Acceleration [IBM09]. We discover a previously unknown bug in one of these programs, which has been independently confirmed. We compare the explicit and implicit encodings of DMA operations empirically: when proving correctness using k-induction, we find that the implicit encoding is less amenable to k-induction analysis than the explicit encoding; for several examples, the property of DMA race freedom is k-inductive, for a small value of k, only with the explicit encoding. However, in the majority of our experiments, k-induction can successfully prove DMA race freedom with either encoding. In these cases, the implicit encoding leads to significant reductions in verification time, at best providing a $10\times$ speedup for one DMA-intensive benchmark. Our experiments also show the effectiveness of our methods in comparison to predicate abstraction: k-induction allows us to prove programs correct that cannot be verified using current predicate abstraction tools, and bug-finding is orders of magnitude faster. Additionally, SCRATCH is able to find bugs which go undetected by a runtime race-detection tool for the Cell processor.

In summary, our major contributions are:

- an encoding of DMA operations that allows automatic analysis of DMA races in multicore programs with scratch-pad memory. The new encoding presented in this paper builds on an earlier sketch [DKR11], and provides significant speedups over the encoding used in prior work [DKR10].
- a proof rule for k-induction operating on programs with loops, which we show to be effective when applied to a large set of realistic DMA-based programs. On top of the contribution of [DKR10] we present a proof of soundness for our k-induction rule, discuss methods for handling nests of loops, and experimentally compare the relative effectiveness of k-induction for our two encodings of DMA operations.
- SCRATCH, an automatic DMA race analysis tool for the Cell BE processor.

To our knowledge, this line of work marks the first application of k-induction to software verification, and of software model checking to heterogeneous multicore programs.

2 Direct memory access operations

We consider heterogeneous multicore processors consisting of a host core, connected to main memory, and a number of accelerator cores with private scratch-pad memories, as depicted in Figure 1. Each core is equipped with a *single* scratch-pad memory, to which it has exclusive access. One or more threads can run on each accelerator core, and threads do not migrate between cores during execution. Thus each thread has an associated core. We assume that the accelerator local memories are indexed by disjoint sets of addresses, so that a pointer p refers to a location in the scratch-pad memory of at most one accelerator core. This assumption is for ease of presentation only, and does not hold for some architectures. It would be trivial (but laborious) to adapt our presentation to drop this assumption, identifying a scratch-pad memory location by a pair (c, p), where c specifies a particular accelerator core, and p is an address referring to the scratch-pad memory for c.

A DMA operation[1] specifies that a contiguous chunk of memory, of a given size, should be transferred between two memory addresses l and h. The address l refers to accelerator memory (*local store*), and h to main memory (*host memory*). A *tag* (typically an integer value) must also be specified with a DMA; the operation is said to be *identified* by this tag. It is typical for DMA operations to be initiated by the accelerator cores: an accelerator *pulls* data into local store, rather than having the host *push* data. We assume this scenario throughout the paper.

DMA operations are non-blocking – having issued a DMA, an accelerator thread continues executing while the operation is handled by a specialized piece of hardware called a *memory flow controller*. Each accelerator core has its own associated memory flow controller. An accelerator thread can issue a *wait* operation, specifying a tag t. This causes the thread to block until all DMAs being processed by the memory flow controller associated with the thread's core, and identified by t, have completed. A DMA identified by tag t is *pending* until a wait operation with tag t is issued.

The asynchronous nature of DMA operations is essential to achieving high-performance. Several memory movement operations can be executed in parallel, and the latency associated with memory transfers can be hidden by overlapping computation with communication. It is also this asynchronous nature which makes DMA operations hard to program correctly.

[1] For brevity, we sometimes write "DMA(s)" rather than "DMA operation(s)".

Although a DMA *may* complete before an explicit wait operation is issued, this cannot be guaranteed. Access (by the host or accelerator) to the region of memory being modified by a pending DMA should be regarded as a bug, as should write access to either region of memory associated with a pending DMA. Failure to issue a wait operation can result in nondeterministic behavior: it may *usually* be the case that the required data has arrived, but occasionally the lack of an explicit wait may result in reading from uninitialized memory, leading to incorrect computation. This nondeterminism means that bugs arising due to misuse of DMA can be extremely difficult to reproduce and fix. This motivates the need for formal analysis techniques to aid programmers in the development of correct DMA-based programs.

2.1 DMA primitives and properties of interest

We consider the following basic primitives for DMA operations:

- $\mathsf{put}(l, h, s, t)$: issues a transfer of s bytes from local store address l to host address h, identified by tag t
- $\mathsf{get}(l, h, s, t)$: issues a transfer of s bytes from host address h to local store address l, identified by tag t
- $\mathsf{wait}(t)$: blocks until completion of all pending DMA operations identified by tag t

In addition, we consider variants of put and get which allow sequences of DMA operations identified by the same tag to be efficiently synchronized without requiring wait operations:

- $\mathsf{putf/getf}(l, h, s, t)$ (put/get with *fence*): same as put/get, except that the operation will not commence until all currently pending operations identified by tag t have completed
- $\mathsf{putb/getb}(l, h, s, t)$ (put/get with *barrier*): same as put/get, except that the operation, and any future operations identified by tag t, will not commence until all currently pending operations identified by tag t have completed[2]

For each accelerator core, we assume hardware-imposed maximum values M and T for the number of bytes that may be transferred by a single DMA, and the number of distinct tags, respectively. We assume that tags are integers in the range $[0, T-1]$.

We have informally described the notion of memory being corrupted by DMA operations. A special case of memory corruption is where two pending DMAs refer to overlapping regions of memory, and at least one of the DMAs modifies the region of memory. We call this a *DMA race*, and focus our attention on the detection of DMA races for the remainder of the paper. This focus is for ease of presentation only: our techniques can be readily adapted to detect races where the buffer referred to by a pending DMA is accessed by non-DMA statements.

In the remainder of the paper, we use the following predicate:

$$\mathsf{disjoint}(a_1, s_1, a_2, s_2) \triangleq (a_1 + s_1 \leq a_2) \vee (a_2 + s_2 \leq a_1)$$

specifying that the memory regions $[a_1, a_1 + s_1)$ and $[a_2, a_2 + s_2)$ are disjoint.

[2] Note that a barrier operation identified by tag t protects prior operations identified by t from itself and future operations that use tag t. However, the barrier does *not* protect *itself* from such future operations.

Definition 1 Let $\mathsf{op}_1(l_1, h_1, s_1, t_1)$ and $\mathsf{op}_2(l_2, h_2, s_2, t_2)$ be a pair of simultaneously pending DMA operations, where $\mathsf{op}_1, \mathsf{op}_2 \in \{\mathsf{put}, \mathsf{get}\}$. The pair is said to be *race free* if the following holds:

$$\big((\mathsf{op}_1 = \mathsf{put} \wedge \mathsf{op}_2 = \mathsf{put}) \vee \mathsf{disjoint}(l_1, s_1, l_2, s_2)\big) \wedge$$
$$\big((\mathsf{op}_1 = \mathsf{get} \wedge \mathsf{op}_2 = \mathsf{get}) \vee \mathsf{disjoint}(h_1, s_1, h_2, s_2)\big).$$

The first conjunct in Definition 1 asserts that the local store regions referred to by op_1 and op_2 do not overlap, unless both are put operations (which do not modify local store); the second conjunct asserts that the host memory regions do not overlap, unless both op_1 and op_2 are get operations (which do not modify host memory). We say there is a *DMA race* when some pair of pending DMA operations is not race free.

The conditions for race freedom with fence and barrier operations are more complex, and are discussed in detail in §5.4.

2.2 DMA operations in the Cell BE processor

The Cell BE processor [Hof05, IBM09] is a heterogeneous multicore architecture consisting of a host Power Processor Element (PPE) core, together with 8 accelerator cores, known as Synergistic Processor Elements (SPEs). The PPE is a regular CPU core connected to a large main memory, whereas the SPE cores are fast vector processors, each equipped with a 256K scratch-pad memory. As discussed in §1, these scratch-pad memories are *not* coherent with main memory, and SPE software must use DMA operations to transfer data between scratch-pad memory and main memory.

Each SPE is equipped with a memory flow controller supporting up to 16 concurrent DMA operations, which may be executed out-of-order. If an SPE attempts to issue a DMA operation when the hardware limit of 16 concurrent operations has been reached, SPE execution stalls until some DMA operation completes.

The maximum amount of data that can be transferred by a single DMA operation is 16K, thus $M = 16384$. The number of distinct tags, T, is 32. This allows a set of tags to be represented using a 32-bit word.

2.3 Illustrative example: triple-buffering

Figure 2, adapted from an example provided with the IBM Cell SDK [IBM09], 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 store 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 as they are not relevant to the data race.

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

At this point in execution the operations marked (*) are both pending, since the only intervening wait operation uses a distinct tag. The operations are not race free according to Definition 1 since they use the same region of local store and one is a get. The race can be avoided by inserting a wait with tag `tags[get_buf]` before the get at (5), or replacing the get at (5) with a getf operation.

We discovered this bug using SCRATCH, our automatic DMA analysis tool, described in §7, 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 via the Cell BE forum has been confirmed by IBM engineers. In the remainder of the paper, we present the techniques employed by SCRATCH to enable these results.

3 Overview of our method

We present a DMA race analysis technique geared towards verification of a single single accelerator thread, which may be running as part of a concurrent application. Our method can detect races between multiple DMA operations issued by the same accelerator thread, but not races between operations issued by distinct threads. Because an accelerator thread can issue many concurrent DMA operations, writing correct code for a single thread can be challenging, as demonstrated by the triple-buffering example of §2.3. Providing a DMA race analysis technique for sequential software is thus an important contribution. Restricting the analysis to consider a single thread enables a scalable method, avoiding the state-space explosion associated with multiple thread interleavings. Nevertheless, the detection of DMA races between multiple threads remains an important problem; in §9 we discuss plans for future work in this area.

Our method is summarised by the flowchart of Figure 3, and has been implemented, for the Cell BE processor, in the SCRATCH tool (see §7). We now discuss the various components of this flow-chart.

Program slicing. Typically, programs for scientific or media-processing computation contain a significant amount of intricate code that is irrelevant to the way in which DMA is used to transfer data between host and accelerator memory. To remove as much of this detail as possible, we first perform program slicing, reducing the program code to the DMA-relevant statements.

Encoding DMA operations. After slicing, the program is instrumented with assertions that check the conditions required for a DMA race to occur. Instrumentation depends on an appropriate encoding of DMA operations; in §5 we present two encoding schemes, which are implemented in SCRATCH. Once instrumentation is complete, the resulting program can be analysed in an attempt to reveal potential DMA races, or prove their absence.

Bounded model checking, for detection of DMA races. If a program is suspected to give rise to a DMA race, then BMC techniques [BCC+03] can be applied to search for races up to a

user-specified execution depth. SCRATCH uses CBMC [CKL04], a bounded model checker for C programs, for this purpose. If CBMC finds a violation of an assertion introduced by DMA instrumentation then the corresponding counterexample demonstrates how a DMA race may manifest.

Proving absence of DMA races using k-induction. Alternatively, our method can attempt to prove DMA race-freedom for a program using the k-induction technique, applied at the level of program loops. In §6, we present a proof rule for k-induction which operates on a *single* loop. This rule can be applied to programs containing multiple loops by first rewriting all program loops as a single, monolithic loop, using a standard technique [Har80] which we recap in §7. This is the *loop transformation* stage in the flowchart of Figure 3.

Applying k-induction involves checking a base case and a step case for a given value of k. Our k-induction rule guarantees that these are straight-line programs, whose correctness can be established in a straightforward manner. SCRATCH invokes CBMC for such checks. If the base case fails, this indicates that the program can give rise to a DMA race, and a counterexample is reported. If both the base and step cases pass, soundness of our k-induction rule (proved in §6) means that the program has been shown to be free of DMA races. Otherwise, in the case where the base case passes but the step case fails, there are two options: the verification attempt can be abandoned, or a larger value of k can be tried.

Extent of automation. While SCRATCH attempts to perform automatic analysis of SPE programs for the Cell BE processor, we do not describe the tool as *fully* automatic, for two reasons. First, SPE programs typically make heavy use of single instruction multiple data (SIMD) intrinsic functions that are specific to the Cell architecture. The prototype SCRATCH tool does not support reasoning about this large set of functions, thus program slicing is performed manually in our experiments. However, it can in principle be automated. Second, and more crucially, the success of k-induction in automatically producing a conclusive verification result is largely dependent on the strength of assertions appearing in the program text. As discussed in §6.2, and demonstrated experimentally in §7, our encodings of DMA operations usually lead to assertions that are inductive already for small values of k. However, in some of our experiments, it proved necessary to supply additional strengthening assertions by hand. In related work we have shown that such assertions can often be inferred automatically using abstract interpretation [DHK11].

4 Goto programs

We present our results in terms of a simple goto language, which is minimal, but general enough to uniformly translate C programs like the one in Figure 2. The syntax of the goto language is shown in the following grammar, in which $x \in X$ ranges over integer variables, $a \in A$ over array variables, ϕ and e over boolean and integer expressions (for which we do not define syntax, assuming the standard operations), and $l_1, \ldots, l_k \in \mathbb{Z}$ over integers:

$$\begin{array}{lll} \mathit{Prog} & ::= & 1\colon \mathit{Stmt}; \ldots; n\colon \mathit{Stmt} \\ \mathit{Stmt} & ::= & \mathit{VarRef} := * \mid \mathsf{assume}\ \phi \mid \mathsf{assert}\ \phi \mid \mathsf{goto}\ l_1, \ldots, l_k \end{array} \qquad \begin{array}{lll} \mathit{VarRef} & ::= & x \mid a[e] \end{array}$$

A goto program is a list of statements numbered from 1 to n.

The language includes assertions, nondeterministic assignment ($\mathit{VarRef} := *$), assumptions (which can constrain variables to specific values), and nondeterministic gotos. Execution of a goto statement, which is given a sequence of integer values as argument (the *goto*

targets), causes the value of one of these (possibly negative) integers to be added to the instruction pointer. We use $x := e$ and $a[i] := e$ as shorthands for assignments to variables and array elements, respectively, which can be expressed in the syntax above via a sequence of nondeterministic assignments and assumptions. For simplicity, we assume variables and array elements range over the mathematical integers, $\mathbb{Z}$; when translating C programs into the goto language the actual range of variables will always be bounded, so SAT-based analysis of goto programs by means of bit-blasting is possible.

The transition system described by a program $\alpha = 1\colon \alpha_1; \ldots; n\colon \alpha_n$ is a graph (S, E_α), where S is the set of program states and E_α the transition relation. Program states are given by the set

$$S = \left\{(\sigma, pc) \mid \sigma : (X \cup (A \times \mathbb{Z})) \to \mathbb{Z},\ pc \in \mathbb{Z}\right\} \cup \left\{\lightning\right\}$$

in which σ is a store mapping variables and array locations to integer values, pc is the instruction pointer, and $\lightning$ is a distinguished state that designates erroneous termination of a program.

We write t^σ for the value of an expression given the variable assignment σ, denote the set of all storage locations by $L = X \cup (A \times \mathbb{Z})$, and define tt, ff to be the truth values of boolean expressions. The set of transitions E_α is as follows:

$$\begin{aligned}
E_\alpha \;=\; & \{(\sigma, pc) \to (\sigma', pc+1) \mid \alpha_{pc} = x := *,\ \forall l \in L \setminus \{x\}.\ \sigma(l) = \sigma'(l)\} \\
& \cup \{(\sigma, pc) \to (\sigma', pc+1) \mid \alpha_{pc} = a[e] := *,\ \forall l \in L \setminus \{(a, e^\sigma)\}.\ \sigma(l) = \sigma'(l)\} \\
& \cup \{(\sigma, pc) \to (\sigma, pc+1) \mid \alpha_{pc} = \mathsf{assume}\ \phi,\ \phi^\sigma = tt\} \\
& \cup \{(\sigma, pc) \to (\sigma, pc+1) \mid \alpha_{pc} = \mathsf{assert}\ \phi,\ \phi^\sigma = tt\} \\
& \cup \{(\sigma, pc) \to \lightning \mid \alpha_{pc} = \mathsf{assert}\ \phi,\ \phi^\sigma = ff\} \\
& \cup \{(\sigma, pc) \to (\sigma, pc + l_i) \mid \alpha_{pc} = \mathsf{goto}\ l_1, \ldots, l_k,\ i \in \{1, \ldots, k\}\}
\end{aligned}$$

If the context α is clear, we just write $s \to s'$ for the membership $(s \to s') \in E_\alpha$.

Proper termination of α in a state s is denoted by $s\downarrow$ and occurs if the instruction pointer of s does not point to a valid statement: $s\downarrow \;\equiv\; s = (\sigma, pc) \wedge pc \notin [1, n]$. Note that no transitions exist from states s with $s\downarrow$.

The set $\mathit{traces}(\alpha)$ of (finite and infinite) traces of a program α is defined in terms of its transition system:

$$\begin{aligned}
\mathit{traces}(\alpha) \;=\; & \left\{ s_1\, s_2\, \cdots\, s_k \;\middle|\; \begin{array}{l} \exists\sigma.\ s_1 = (\sigma, 1),\ s_k\downarrow \text{ or } s_k = \lightning, \\ \forall i \in \{1, \ldots, k-1\}.\ s_i \to s_{i+1} \end{array} \right\} \\
& \cup \{s_1\, s_2\, \cdots \mid \exists\sigma.\ s_1 = (\sigma, 1),\ \forall i \in \mathbb{N}.\ s_i \to s_{i+1}\}
\end{aligned}$$

In particular, no traces exist on which assumptions fail.[3] A program α is considered *correct* if no trace in $\mathit{traces}(\alpha)$ terminates erroneously, *i.e.* no trace contains $\lightning$.

5 Encoding DMA operations in goto programs

We now consider the goto language extended with the DMA primitives of §2.1:

$$\begin{aligned}
\mathit{Stmt} \;::=\; & \ldots \mid \mathsf{get}(e,e,e,e) \mid \mathsf{put}(e,e,e,e) \mid \mathsf{getf}(e,e,e,e) \mid \mathsf{putf}(e,e,e,e) \\
& \mid \mathsf{getb}(e,e,e,e) \mid \mathsf{putb}(e,e,e,e) \mid \mathsf{wait}(e)
\end{aligned}$$

[3] In our context, this is preferable to modeling failed assumptions via a distinguished "blocked program" state: it simplifies the notion of sequential composition of programs (*cf.* §6.1).

We present two methods for translating a goto program in this extended language to a standard goto program, replacing DMA operations with suitable instrumentation code such that a potential DMA race results in a failed assertion.

In §5.1 we present an explicit encoding, where DMA races are analyzed by logging a bounded history of previously pending operations. This is the encoding used in [DKR10]. We then present, in §5.2, a new, implicit encoding where a single pending DMA operation is selected nondeterministically for tracking. This allows analysis of programs which may issue an unbounded number of DMA operations.

We choose the names *explicit* and *implicit* for our encodings because the first encoding detects a DMA race by comparing a newly issued DMA explicitly with every pending DMA, while the second encoding performs this comparison implicitly, by comparing the newly issued DMA with a single, arbitrary pending DMA.

In §7 we show experimentally that the implicit encoding can provide significant improvements in verification time over the explicit encoding. We illustrate the difference between these approaches using examples in §5.3.

In both cases, we do not initially present details of the way fence and barrier operations are encoded. In §5.4 we show how this is achieved for the implicit encoding (a similar approach can be applied to the explicit encoding).

5.1 An explicit encoding

Our goal is to ensure that, during program execution, a thread does not issue a DMA that races with another DMA issued previously by the thread. Suppose, for a given program, we have an upper bound D on the number of DMAs that may be concurrently pending. In this case, we can check the conditions for a DMA race by explicitly logging the set of DMAs that are concurrently pending during program execution. This requires recording a history of at most size D. When a new DMA is issued, we first assert that the operation does not race with any existing DMA. We then assert that the size of the set of concurrently pending DMAs is smaller than D. Finally, we add the new DMA to the log. A wait operation of the form $\mathsf{wait}(t)$, where t is a tag, is encoded by removing from the log any DMA identified by tag t.

The log of DMA operations is encoded as a series of *tracker arrays*, as follows, with $0 \leq j < D$:

- $valid$: $valid[j] = 1$ if values at position j in the other arrays are being used to track a DMA, otherwise $valid[j] = 0$ and values at position j in the other arrays are meaningless
- is_get: $is\_get[j] = 1$ if the j-th tracked DMA is a get, otherwise $is\_get[j] = 0$
- $local$, $host$, $size$, tag: element j records the local store address, host address, size and tag of the j-th tracked DMA, respectively

Figure 4 shows how a program with basic DMA primitives can be translated into a standard goto program, where get, put and wait operations are replaced with assertions and assignments over the tracker arrays. The translation makes use of the disjoint predicate, defined in §2.1. We use $\forall_{0 \leq j < D}\ Stmt$ to indicate that $Stmt$ should be duplicated D times with increasing values for j. Note that, because D is fixed at translation time, this leads to D consecutive statements, rather than the generation of a loop. Since the rules of Figure 4 replace single statements with multiple statements, it is necessary to perform a re-numbering of program statements and goto targets after translation; we omit details of this re-numbering.

The encoding of DMAs is based on Definition 1, and is designed to prohibit the issue of DMAs that are simultaneously pending but not race free. Note that in our simple goto language we do not model actual movement of data via DMA. In practice, to achieve soundness, we must set the memory locations written to by a DMA operation to nondeterministic values.

This explicit encoding is natural as it mirrors the idea of runtime logging of DMA operations, which is performed for example by the IBM Race Check library [IBM08]. The disadvantage of the encoding is that it depends on the existence of a limit, D, for the number of DMAs that may be concurrently pending in a given program. The cost of the encoding increases proportionally with D. This can lead to scalability issues when analyzing instrumented programs using SAT-based bounded model checking. Furthermore, the encoding does not allow reasoning about programs that may issue an arbitrary number of simultaneous DMAs.

5.2 A more efficient, implicit encoding

We now present a more efficient encoding of DMA operations which does not require the upper limit D. This encoding allows us to analyze programs which may issue an unbounded number of DMAs.

The key insight which leads to a more efficient encoding is the fact that checking for DMA races only requires *pairwise* consideration of DMA operations. In translating put and get in Figure 4, we use a universal quantifier to compare the new DMA operation against every previously issued DMA operation that is still live. Observe that it suffices to nondeterministically record details of a single, *arbitrary* DMA operation, and check further operations for races with respect to this operation. A wait operation of the form $\mathsf{wait}(t)$ can be encoded by an assumption that the currently tracked DMA (if any) does not have associated tag t. In other words, the encoding ensures that we discard execution traces along which it was chosen to track, and subsequently wait for completion of, a given DMA operation. This approach resembles the concept of *prophecy variables* [AL91], since the future program execution determines whether a DMA operation should be tracked or not. As discussed above, we call this encoding *implicit* because a new DMA is implicitly compared with all pending DMAs by tracking a single, arbitrary pending DMA.

Translation rules for this implicit encoding are presented in Figure 5. Note that $valid$, $local$, $host$, $size$ and tag are no longer arrays: they are now scalar variables that collaboratively track a *single* DMA operation when $valid = 1$. For clarity in the translation rules, we use labels *track* and *after* to denote the targets of a nondeterministic goto statement, rather than explicit integer offsets to the instruction pointer as the goto program syntax of §4 strictly requires.

The encoding of Figure 5 is more compact than the explicit encoding of Figure 4, since it involves tracking just one DMA operation, rather than an array of D operations. In §7 we demonstrate experimentally that the implicit encoding results in faster verification compared with the explicit encoding.

5.3 Examples

We illustrate the difference between the explicit and implicit encodings using the simple examples of Figure 6.

The program of Figure 6(a) is clearly race-free: the get operation identified by tag t is immediately followed by a wait operation using tag t, ensuring that the get operation completes before the put operation at line 3 commences. The left-hand-side of Figure 7 illustrates the transition system corresponding to this program when DMA operations are translated using the explicit encoding of Figure 4. Each rectangle represents a state, labeled with a program counter location followed by a set of pending DMA operations. The figure illustrates that, as the program is executed, the set of tracked DMAs is modified accordingly. The right-hand-side of Figure 7 illustrates the corresponding transition system when the implicit encoding of Figure 5 is used. Dashed lines are used to illustrate paths through the transition system which end in failed assumptions, thus not forming part of the set of program traces. States are labeled by a program counter location followed by details of a single tracked DMA operation, with '-' used to represent the case where no operation is tracked (*i.e.* when $valid = 0$). From the initial state, a nondeterministic choice determines whether operation $\mathsf{get}(l, h, s, t)$ is tracked. The right-hand branch represents the case where this operation is tracked. The encoding of the $\mathsf{wait}(t)$ operation assumes that no operation with tag t is tracked, invalidating this branch. As a result, the only valid state from which $\mathsf{put}(l, h + s, s, t)$ can be executed is the state $[3, \text{-}]$, in which case there is clearly no DMA race.

The program of Figure 6(b) is the same as that of Figure 6(a), except that the $\mathsf{wait}(t)$ operation has been removed. This results in a potential DMA race, since data may be simultaneously read from and written to l. The left- and right-hand-sides of Figure 8 illustrate the transition systems for this program when the explicit and implicit encodings are used, respectively. Comparing the right-hand-sides of Figures 7 and 8, observe the same nondeterministic choice to track $\mathsf{get}(l, h, s, t)$ is initially made. However, in Figure 8 the choice to track this operation is *not* invalidated by a subsequent wait operation, since no such operation occurs in the program of Figure 6(b). As a result, an attempt to execute operation $\mathsf{put}(l, h + s, s, t)$ can be made from the state $[2, \mathsf{get}(l, h, s, t)]$, leading to a DMA race.

5.4 Handling fences and barriers

As discussed in §2.1, DMA subsystems such as that of the Cell processor typically support modified versions of get and put that use *fences* and *barriers* [IBM09].

Recall from §2.1 that a fenced put/get operation (denoted putf/getf) identified by tag t will not commence until all currently pending operations identified by tag t have completed. However, the fenced operation provides no guarantees for future operations on t. A put/get operation with barrier (denoted putb/getb) identified by tag t similarly will not commence until all currently pending operations identified by tag t have completed. In addition, future operations identified by tag t will not commence until operations pending before issue of the barrier have completed. However, a barrier does not protect *itself* from future operations identified by t.

Note that the operation $\mathsf{putb/getb}(l, h, s, t)$ is *not* equivalent to the sequence $\mathsf{wait}(t)$; $\mathsf{put/get}(l, h, s, t)$. In the latter case, the $\mathsf{wait}(t)$ operation causes execution to block until the DMA operation on tag t has completed. In the former case, the DMA barrier is a non-blocking operation.

To extend our existing encodings to handle fences we weaken the DMA race check condition to allow a new DMA to overlap with an existing DMA if the new DMA is a fenced operation and both DMAs are identified by the same tag.

Adapting our encodings to support barriers requires an additional $protected$ flag to be associated with each tracked DMA. For any DMA (including a barrier), $protected$ is initially set to 0. When a barrier DMA is issued with identifying tag t, $protected$ is set to *true* for every existing DMA identified by t. The DMA race check condition is then weakened (for all types of DMA, not just barriers) to permit a new DMA to overlap with an existing DMA if both DMAs are identified by the same tag and $protected$ holds for the existing DMA.

In Figure 9, we show how the implicit encoding of Figure 5 can be extended with support for fence and barrier operations. The explicit encoding of Figure 4 is extended in a similar manner. In Figure 9, to avoid redundancy in our presentation of the translation for barrier operations, we write "translation for $\mathsf{get}/\mathsf{put}(l, h, s, t)$" to denote the statements obtained by applying the translation for $\mathsf{get}/\mathsf{put}(l, h, s, t)$ verbatim.

Note that $protected$ is set to 0 for all newly issued DMA operations, including barriers. This is because, as discussed above, a barrier identified by tag t protects all prior operations identified by t, *but not itself*, from future operations identified by t.

6 k-Induction for goto programs

Our encodings of DMA programs are directly amenable to analysis via bounded model checking [BCC$^+$03] as an effective method to discover DMA races. However, BMC alone cannot be used to verify the (unbounded) *absence* of DMA races in programs with loops.

The k-induction procedure [SSS00], proposed as a method to allow verification of hardware designs (represented as finite state machines) using a SAT solver, is a stronger version of the standard invariant approach to verify safety properties. Using normal invariants, proving that a system satisfies a safety property ϕ requires showing that

(i) some formula I (possibly identical to ϕ) holds in all initial states,
(ii) I is preserved by all state transitions of the system (I is *inductive*), and
(iii) I implies ϕ.

The main difficulty of this method is the construction of inductive formulae I. The k-induction principle addresses this difficulty by weakening (ii) to the property that I has to be preserved only if it held in the previous k states of execution. In return, (i) has to be strengthened appropriately.

We describe the principle using the notation of [ES03]. Let $\mathbf{I}(s)$ and $\mathbf{T}(s, s')$ be formulae encoding the initial states and transition relation for a finite state system over sets of propositional state variables s and s', and $\mathbf{P}(s)$ a formula representing states satisfying a safety property. For $k \geq 0$, to prove $\mathbf{P}$ by k-induction it is required first to show that $\mathbf{P}$ holds in all states reachable from an initial state within k steps, *i.e.* that the following formula (the base case) is unsatisfiable:

$$\mathbf{I}(s_1) \wedge \mathbf{T}(s_1, s_2) \wedge \cdots \wedge \mathbf{T}(s_{k-1}, s_k) \wedge (\overline{\mathbf{P}(s_1)} \vee \cdots \vee \overline{\mathbf{P}(s_k)}) \ .$$

Secondly, it is required to show that whenever $\mathbf{P}$ holds in k consecutive states $s_1, \ldots, s_k$, $\mathbf{P}$ also holds in the next state s_{k+1} of the system. This is established by checking that the following formula (the step case) is unsatisfiable:

$$\mathbf{P}(s_1) \wedge \mathbf{T}(s_1, s_2) \wedge \cdots \wedge \mathbf{P}(s_k) \wedge \mathbf{T}(s_k, s_{k+1}) \wedge \overline{\mathbf{P}(s_{k+1})} \ .$$

In principle, k-induction can be used for SAT-based software model checking "out-of-the-box". A program can be encoded as a monolithic transition function, where the program

counter is an explicit variable. Assertions appearing in the original program can be gathered together into a single invariant. The encoded program and invariant can be represented as a SAT formula, to which k-induction can be applied.

This naïve encoding has not shown success in practice due to the loss of structure associated with the translation process. Furthermore, an important refinement which boosted the applicability of k-induction to hardware designs is the restriction to loop-free paths [SSS00]. This refinement is not useful when dealing with software, where the state-vector is very large, leading to extremely long loop-free paths. To see this, consider a program consisting of a simple loop, which increments a counter until some statically unknown maximum value is reached:

```
void f(unsigned int x) {
  unsigned int i;
  i = 0;
  while(i <= x) {
    i++;
  }
}
```

For a fixed word size d, the length of the longest loop-free path in the program's state-space exceeds 2^d, the number of possible integer values for this word size. Because realistic programs involve many loops, frequently with unknown bounds, applying k-induction with a value of k as large as the longest loop-free path in the state space is not feasible.

To verify absence of DMA races in goto programs, we present a novel formulation of k-induction, which operates at the loop level, and prove its correctness (§6.1). We then give some intuition as to why k-induction is effective in proving absence of DMA races in example programs written for the Cell processor (§6.2); this intuition is backed up by experimental results in §7.

6.1 A proof rule for k-induction with loops

To present our proof rule for k-induction we require some additional machinery and notation. Given programs $\alpha = 1\colon \alpha_1; \ldots; m\colon \alpha_m$ and $\beta = 1\colon \beta_1; \ldots; n\colon \beta_n$, the *size* of α, denoted $|\alpha|$, is m, and we define the sequential composition of α and β as follows:

$$\alpha \fatsemi \beta \;=_{\text{def}}\; 1\colon \alpha_1; \ldots; m\colon \alpha_m;\; m+1\colon \beta_1; \ldots; m+n\colon \beta_n \;.$$

For $i > 0$, we use α^i to denote the sequential composition of i copies of α, and α^0 to denote the empty program. For a single-statement program of the form $1\colon \alpha_1$, we drop the leading $1\colon$, writing simply α_1.

Definition 2 A program α is *self-contained*, denoted $contained(\alpha)$, if, for each goto statement $i\colon$ goto $\ldots, l, \ldots$ appearing in α, we have $(i + l) \in \{1, \ldots, |\alpha| + 1\}$.

In other words, goto statements can only change the instruction pointer to the locations of statements inside α, or to the location immediately following α. The traces of programs formed using concatenation of self-contained components can be derived from the traces of the components (this is formalized in Lemma 1 below).

In our version of k-induction, assertions contained in a program are used as invariants. To turn assertions into induction hypotheses in this setting, we define a function that replaces all assertions in a program with assumptions. Given a program $\alpha = 1\colon \alpha_1; \ldots; n\colon \alpha_n$, the corresponding program $\alpha_{assume} = 1\colon \alpha'_1; \ldots; n\colon \alpha'_n$ is defined by: $\alpha'_i = \mathsf{assume}\ \phi$ if $\alpha_i = \mathsf{assert}\ \phi$, and $\alpha'_i = \alpha_i$ otherwise.

Finally, we present k-induction as a proof rule operating on distinguished loops in a goto program of the following form:

$$\alpha \fatsemi \mathsf{goto}\ 1, (|\beta| + 2) \fatsemi \beta \fatsemi \mathsf{goto}\ (-|\beta| - 1) \fatsemi \gamma$$

where α, β and γ are self-contained. The program consists of a prelude α, a loop with body β and a tail γ. Other than self-containedness, we do not make any assumptions about the shape of components α, β and γ, which may contain further (nested) loops and arbitrary control structure. We do not demand the presence of an explicit loop condition: loop condition b can be simulated by choosing $\mathsf{assume}\ b$ as the first statement of the loop body, and $\mathsf{assume}\ \neg b$ as the first statement of the tail. Note that the restriction to self-contained components is mild, *e.g.* early exit from the loop via a break statement can be simulated by a flag together with an appropriate loop condition. Our implementation of k-induction (see §7) can be applied to C programs with arbitrary loop structures, provided the control-flow-graph associated with a given program is reducible [ALSU06].

Proof rule for k-induction

$$\frac{\begin{array}{c} \mathit{contained}(\alpha) \qquad \mathit{contained}(\beta) \qquad \mathit{contained}(\gamma) \qquad k \geq 0 \\ \alpha \fatsemi \gamma \text{ is correct} \qquad \{\alpha_{assume} \fatsemi \beta^{i-1}_{assume} \fatsemi \beta \fatsemi \gamma \text{ is correct}\}_{i \in \{1,\ldots,k\}} \\ \beta^{k}_{assume} \fatsemi \beta \text{ is correct} \qquad \beta^{k}_{assume} \fatsemi \gamma \text{ is correct} \end{array}}{\alpha \fatsemi \mathsf{goto}\ 1, (|\beta| + 2) \fatsemi \beta \fatsemi \mathsf{goto}\ (-|\beta| - 1) \fatsemi \gamma \text{ is correct}}$$

In this rule, the assertions present in the program (*e.g.* the formulae in Figures 4, 5 and 9) take the role of the inductive invariant needed for verification. The premises include base cases requiring the program to be shown correct when the prelude, followed by between zero and k loop iterations, are executed. The premises $\beta^{k}_{assume} \fatsemi \beta$ *is correct* and $\beta^{k}_{assume} \fatsemi \gamma$ *is correct* form the induction step, establishing that if it is possible to execute k loop iterations from an arbitrary state without violating any assertions then it is possible to successfully execute a further loop iteration, or the loop tail.

Theorem 1 (Correctness) *The above proof rule is sound.*

Before we can prove the correctness of Theorem 1, we need to characterize the traces of programs constructed by concatenation of self-contained programs. The next lemma follows directly from the trace semantics of goto programs (§4) and the definition of self-contained programs (Definition 2):

Lemma 1 *Suppose α, β are self-contained programs such that $|\alpha| = n$. Given the state $t = (\sigma, pc) \in S$, we write $t^n = (\sigma, pc + n)$ for the state with instruction pointer shifted by $n \in \mathbb{Z}$ (with the special case $\lightning^n = \lightning$). The traces of $\alpha \fatsemi \beta$ are:*

$$\begin{aligned} \mathit{traces}(\alpha \fatsemi \beta) \;=\; & \left\{ s_1\, s_2 \cdots s_{k-1}\, t_1^n\, t_2^n \cdots \;\middle|\; \begin{array}{r} s_1\, s_2 \cdots s_k \in \mathit{traces}(\alpha), \\ t_1\, t_2 \cdots \in \mathit{traces}(\beta), \\ \exists \sigma, l.\ s_k = (\sigma, l) \wedge t_1 = (\sigma, 1) \end{array} \right\} \\ & \cup \{ s_1\, s_2 \cdots s_k\, \lightning \mid s_1\, s_2 \cdots s_k\, \lightning \in \mathit{traces}(\alpha) \} \\ & \cup \{ s_1\, s_2 \cdots \mid s_1\, s_2 \cdots \in \mathit{traces}(\alpha) \} \end{aligned}$$

In the first case, $t_1\, t_2 \cdots$ denotes both finite and infinite traces of β, while $s_1\, s_2 \cdots$ in the last case is an infinite trace.

We also require the following simple result about the relationship between programs α and α_{assume}:

Lemma 2 *The traces of α and α_{assume} are related as follows:*

$$traces(\alpha_{assume}) \;=\; \{s \in traces(\alpha) \mid s \textit{ does not contain } \lightning\}$$

Proof (Theorem 1) With the help of Lem. 1, it can be observed that the program

$$\alpha \fatsemi \mathsf{goto}\ 1, (|\beta| + 2) \fatsemi \beta \fatsemi \mathsf{goto}\ (-|\beta| - 1) \fatsemi \gamma$$

is correct if and only if the programs α, $\alpha \fatsemi \beta^i$, and $\alpha \fatsemi \beta^i \fatsemi \gamma$ are correct for each $i \in \mathbb{N}$. Furthermore, for any self-contained programs α_1, α_2 it is the case that:

$$\alpha_1 \fatsemi \alpha_2 \text{ is correct} \;\Rightarrow\; \alpha_1 \text{ is correct} \tag{1}$$

$$\alpha_1 \text{ and } \alpha_{1\,assume} \fatsemi \alpha_2 \text{ are correct} \;\Rightarrow\; \alpha_1 \fatsemi \alpha_2 \text{ is correct} \tag{2}$$

We have:

- α *is correct:* by (1), this follows from the correctness of $\alpha \fatsemi \gamma$.
- $\alpha \fatsemi \beta^i$ *is correct for* $i \in \{1, \ldots, k\}$*:* this is proven by induction on i. We assume that $\alpha \fatsemi \beta^i$ is correct (for $i \in \{0, \ldots, k-1\}$) and show the correctness of $\alpha \fatsemi \beta^{i+1} = \alpha \fatsemi \beta^i \fatsemi \beta$. From the correctness of $\alpha_{assume} \fatsemi \beta^i_{assume} \fatsemi \beta \fatsemi \gamma$ and (1), we know that the program $\alpha_{assume} \fatsemi \beta^i_{assume} \fatsemi \beta$ is correct. By the induction hypothesis and (2), this implies that $\alpha \fatsemi \beta^{i+1}$ is correct.
- $\alpha \fatsemi \beta^i$ *is correct for* $i > k$*:* again, we reason by induction over i and assume that $\alpha \fatsemi \beta^i$ is correct for some $i \geq k$. Because $\beta^k_{assume} \fatsemi \beta$ is correct, so is $\alpha_{assume} \fatsemi \beta^{i-k}_{assume} \fatsemi \beta^k_{assume} \fatsemi \beta = \alpha_{assume} \fatsemi \beta^i_{assume} \fatsemi \beta$ (by Lem. 1), which together with (2) and the induction hypothesis entails the correctness of $\alpha \fatsemi \beta^{i+1}$.
- $\alpha \fatsemi \gamma$ *is correct:* given as premise of the rule.
- $\alpha \fatsemi \beta^i \fatsemi \gamma$ *is correct for* $i \in \{1, \ldots, k\}$*:* follows from the correctness of the programs $\alpha_{assume} \fatsemi \beta^{i-1}_{assume} \fatsemi \beta \fatsemi \gamma$ and $\alpha \fatsemi \beta^{i-1}$, and (2).
- $\alpha \fatsemi \beta^i \fatsemi \gamma$ *is correct for* $i > k$*:* because $\beta^k_{assume} \fatsemi \gamma$ is correct, so is the program $\alpha_{assume} \fatsemi \beta^{i-k}_{assume} \fatsemi \beta^k_{assume} \fatsemi \gamma = \alpha_{assume} \fatsemi \beta^i_{assume} \fatsemi \gamma$. Together with (2) and the correctness of $\alpha \fatsemi \beta^i$, this entails that $\alpha \fatsemi \beta^i \fatsemi \gamma$ is correct. □

By presenting k-induction using a general proof rule, we do not restrict the method to a SAT-based implementation. Although our practical implementation is SAT-based, the rule could also be used in any (possibly interactive) deductive verification system.

6.2 k-induction for DMA programs

Through our experiments in §7 we observe that k-induction works extremely well for checking assertions representing DMA race-freeness, generated by the rules in Figures 4, 5 and 9. For realistic example programs written for the Cell processor, the generated assertions are mostly inductive already for small k, with no further annotations required to verify correctness. The result is a verification method that is fully automatic and efficient on a large range of Cell programs. For a small number of benchmarks we find that the implicit encoding of DMAs does not yield inductive assertions, while the explicit encoding does. We discuss this in detail in §7.5.

Intuitively, k-induction works well in this application domain 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 [AFF+05].

7 Experimental evaluation

7.1 SCRATCH

We have implemented a prototype tool, SCRATCH[4], built on top of the bounded model checker CBMC [CKL04]. SCRATCH accepts an arbitrary C program written for an SPE core of the Cell BE processor, and checks for DMA races involving scratch-pad memory. The tool uses the encodings described in §5 to transform the input program into a form where DMAs are replaced with assertions and assignments over tracker variables. In the case of the explicit encoding, the size D of the tracker arrays is specified as a command-line argument to SCRATCH.

Having translated the input program into an instrumented form, SCRATCH can apply bounded model checking to check for DMA races up to a certain execution depth. To prove absence of races, SCRATCH combines bounded model checking with k-induction, using the loop-level formulation of §6. For a program consisting of a single, non-nested loop (with prelude and tail), k-induction is applied starting with $k = 0$, and incrementing k by one until either the base case fails (a DMA race has been detected), both the base case and step case succeed (the program has been proved free of DMA races), or k exceeds 10. The starting value, step size and upper limit for k can be configured via command-line arguments. For such restricted loops, the combination of k-induction and SAT-based bounded model checking is sound: the base and step case programs generated by the k-induction proof rule are loop-free, thus SAT-based bounded model checking can provide an exhaustive analysis.

Handling multiple loops. SCRATCH can be applied to arbitrary sequences of loop nests (as long as the control-flow-graph associated with the input program is reducible [ALSU06]). A sequence of loop nests is automatically transformed into a single, monolithic loop simulating the nest, using the following well-known technique [Har80]. A *position* variable is used to record which of the original loop bodies is due to be executed. The body of the monolithic loop consists of a case-split on this variable, where each case contains the body of the corresponding loop, together with code to update the position variable appropriately. Once this transformation has been applied, k-induction can be used to directly analyze the monolithic loop, since unwinding this loop leads to a loop-free program.

We also experimented with handling sequences of loop nests by applying the k-induction rule recursively as follows. Given a sequence of loop nests, k-induction is applied to the first outer loop. Unwinding this loop to yield a base case and step case results in programs which themselves contain loops. The k-induction rule can then be applied again to each of these programs, in an attempt to prove their correctness by induction. This process leads to a tree of programs to be analyzed: an interior node n of this tree is a program containing loops;

[4] SCRATCH, together with source code for all benchmarks, is available online: http://www.cprover.org/scratch/.

the left and right children of n are the base and step cases associated with a k_n-inductive proof of correctness for n. (Note that the values k_n used for separate inductive proofs are independent.) The leaf nodes of the tree are loop-free programs which can be analyzed using SAT-based bounded model checking.

We have implemented both of the above approaches. In the absence of heuristics for guessing effective starting and increment values for k, the transformation to a monolithic loop leads to faster verification of our benchmark examples. This transformation is used in our experimental evaluation.

7.2 Benchmarks and experimental platform

We evaluate SCRATCH using a set of 22 benchmarks adapted from examples supplied with the IBM Cell SDK for Multicore Acceleration [IBM09], categorized as follows:

- **x-buf ($x \in \{1, 2, 3\}$)** Eleven data processing programs which use single-, double- or triple-buffering for data-movement (*cf.* Figure 2). 'I/O' in the benchmark title indicates that separate buffers are used for input and output. Some variants of these programs use fences/barriers, indicated by '+ fence'/'+ barrier' in the benchmark title
- **race check, simple dma** Examples which illustrate data races and use of DMA
- **sync atomic/mutex** Programs illustrating the use of SDK synchronization primitives for atomic operations and mutexes, in conjunction with DMA operations
- **cpaudio, normalize** Applications which copy one channel of a stereo audio file to the other, and normalize the volume of a mono audio file, respectively
- **checksum** Computes a checksum on data in host memory. Multiple buffers are used to coordinate data-movement efficiently
- **Euler simple/complex** Particle simulation using Euler integration. The simple version uses separate individual buffers for position, velocity and mass data; the complex version uses double-buffering
- **Julia** n Quaternion Julia set ray-tracing, where an SPE renders n columns of output

As discussed in §3, manual program slicing has been applied to each benchmark to remove portions of code that do not affect DMA operations. This routine slicing could be automated: the sliced code uses vector datatypes and intrinsic functions specific to the Cell processor, which the slicer would need to understand. After slicing, most of the benchmark examples consist of a single, non-nested loop. The *Euler simple*, *Euler complex* and *Julia* benchmarks each involve a loop containing a nested inner loop.

We apply SCRATCH to correct and buggy versions of the benchmarks. With the exception of *3-buf* and *cpaudio*, bugs are injected into the examples, either by removing a `wait` operation, changing the tag used to identify a DMA, or switching an operation from `get` to `put` (or vice-versa). The *3-buf* benchmark is the triple-buffering example discussed in §2.3, in which SCRATCH uncovered an existing bug. A DMA race occurs when the *cpaudio* benchmark is executed with zero frames of audio. This is arguably a bug since the precondition that the number of frames should be positive is not specified.

We present results demonstrating the effectiveness of SCRATCH, equipped with either the explicit or implicit encoding of §5, for bug-finding and proving correctness with respect to DMA races. Experiments are performed on a 3GHz Intel Xeon quad-core machine with 48 GB RAM, running Ubuntu. MiniSat 2.0, compiled with full optimizations, is used as a back-end SAT solver. It has been reported to perform comparatively to state-of-the-art SMT solvers for SMT-$\mathcal{BV}$ [CFMS09] on this type of workload. All times reported are averaged over 5 consecutive runs.

7.3 Bug-finding

With both encodings, bounded model checking proves extremely effective for detecting DMA races. For each benchmark and each encoding, we performed repeated bounded model checking runs to find the minimum execution depth required to exhibit a DMA race. For the explicit encoding, we also iteratively computed, for each benchmark, the smallest value of D (the size of the tracker arrays) necessary to allow race detection. We then measured the time taken for verification (instrumentation + bounded model checking) using these minimum values, averaged over multiple runs.

With the explicit encoding, the maximum verification time across all benchmarks is 1.32s; this is for one of the *race check* examples. For each benchmark, time taken for verification with the implicit encoding is identical or marginally lower, with a maximum time of 0.87s for the same *race check* example. This gap of 0.45s is the largest difference between the two encodings exhibited for our benchmarks with respect to bug-finding—a speedup of $1.52\times$ using the implicit encoding instead of the explicit encoding. The bugs in our benchmarks are relatively shallow; all bugs are found within an execution depth of 523 and 320 statements in the C program for the explicit and implicit encodings, respectively.

By reporting verification times for the explicit encoding when the optimal value of D is used, we have shown the explicit encoding in a favorable light. When using the explicit encoding in practice, one would have to guess a suitable value for D. Guessing a larger value than necessary would result in large BMC instances, while guessing too small a value would result in failed verification attempts. The implicit encoding does not suffer from this practical constraint.

7.4 Proving correctness.

Figure 10 presents experimental data obtained applying SCRATCH to correct versions of the benchmarks, using both the explicit and implicit encodings. For each encoding, the time (in seconds) taken for verification is shown, together with the smallest value of k required to prove correctness. Assuming that the required k is m (for some $m \geq 0$), verification time is the sum of the times for program instrumentation, construction of base/step case programs for $0 \leq k \leq m$, successful bounded model checking of base case programs for $0 \leq k \leq m$, unsuccessful bounded model checking of step case programs for $0 \leq k < m$, and successful bounded model checking of the final step case program for $k = m$.

As with the results for bug-finding, we iteratively computed the smallest value of D required to prove correctness using the explicit encoding, and report results for the explicit encoding when optimum values for D are used; the optimum value for D is shown, for each benchmark, in Figure 10. Again, this shows the explicit encoding in a favorable light: in practice one would have to guess a suitable value for D; this is not necessary when the implicit encoding is used.

Figure 10 also shows the size (number of variables and number of clauses) of the largest SAT instance solved for each benchmark during k-induction. In all cases, this corresponds to the SAT instance associated with verification of the final step-case program.

In five cases, we found that DMA race freedom could be proved automatically using k-induction, for a small value of k, with the explicit encoding but *not* with the implicit encoding. This scenario is indicated by '-' entries in Figure 10, and indicates that k-induction did not succeed for $k \leq 10$. We discuss reasons for this in detail in §7.5. In each case, it is possible to manually add a simple strengthening assertion to allow k-induction to succeed

with the implicit encoding; we give an example of this in §7.5. In Figure 10, rows marked 'strengthened' show results for variants of benchmarks with strengthening assertions.

When k-induction succeeds for both encodings, the values of k required are identical, or differ by one due to benchmark-specific differences in the strength of the induction hypothesis yielded by assertions associated with each encoding.

The results of Figure 10 indicate that k-induction provides a tractable method for proving correctness for this set of benchmarks: for both encodings (excepting cases where k-induction does not succeed for the implicit encoding), the maximum time taken for verification is less than 73 seconds. Restricting our attention to benchmarks where k-induction succeeds for both encodings (17 of the standard benchmarks, plus 5 strengthened benchmarks), we see that the implicit encoding outperforms the explicit encoding in the majority of cases. This is highlighted by the scatter plot of Figure 11, which plots (using a logarithmic scale) the relationship between verification times with both encodings for each benchmark; benchmarks are identified by their row number in Figure 10. In some cases, the implicit encoding significantly outperforms the explicit encoding: verification with the implicit encoding is 10 times faster for the *Euler complex* benchmark. This is due to the dependence of the explicit encoding on D, the maximum number of operations that may be simultaneously pending. For the *Euler complex* benchmark, D is relatively large, which necessitates the solving of large SAT instances, as shown in Figure 10. By comparison, the largest SAT instance associated with the implicit encoding for this benchmark is almost ten times smaller, since the implicit encoding is independent of the parameter D.

The *Julia* benchmark contains a loop for which the number of iterations is a fixed parameter n, the columns of a raytraced image to be computed by one SPE. For this example, k-induction succeeds with $k = n+6$ for the explicit encoding, and $k = n+5$ for the implicit encoding. The results in Figure 10 are for the case where $n = 2$. In Figure 12, we illustrate the scalability of k-induction by plotting the time taken for verification of the Julia benchmark when we vary parameter n between 2 and 18. Growth is less than cubic, showing that our k-induction method scales well.

7.5 Examples where the implicit encoding is too weak for k-induction

As mentioned above, and as indicated in Figure 10, our benchmarks reveal examples where k-induction succeeds with the explicit encoding of DMA operations, but not with the implicit encoding. We discuss the reasons for this by considering one of the examples, the *checksum* benchmark, in detail.

The *checksum* benchmark uses four local buffers, numbered 0, 1, 2 and 3, to receive data from host memory via DMA. Each buffer has size 2^{12} bytes, and the collection of four buffers is implemented as a 4×2^{12} array of type **`char`**, called `buf`. At the start of the program, the value 2^{12} is written to a variable, $size$. There follows a data processing loop with induction variable i such that, on iteration i ($i = 0, 1, \ldots$), a get operation is issued to copy 2^{12} bytes of data from host memory into buffer $i\%4$, *i.e.* into the region $[(i\%4) \times 2^{12}, (i\%4 + 1) \times 2^{12})$ of `buf`. However, the variable $size$, rather than the explicit constant 2^{12}, is specified as the size argument to the get command. The loop is structured so that DMA operations targeting all four buffers may be simultaneously pending.

When proving the step case for k-induction, the value of $size$ is havocked: the model checker considers paths from the start of the loop from states where $size$ is arbitrary. In particular, states where $size$ is larger than 2^{12} are considered. Such states can clearly lead to DMA races, *e.g.* a get operation requesting that more than 2^{12} bytes are transferred into

buffer 0 will result in buffer overflow: the get operation will target memory locations that form part of buffer 1. This will lead to a DMA race during the next loop iteration, when a get is issued targeting buffer 1. Such states are, of course, unreachable, since $size$ is set to 2^{12} before the loop and is never modified further. The assertion $size \leq 2^{12}$ is part of the invariant required to show correctness of the program.

With the explicit encoding, this assertion is computed indirectly via k-induction. A history of four DMA operations is tracked ($D = 4$). Thus, when proving the step case for $k = 4$ the model checker assumes that, on loop iteration $i \geq 4$, the DMA operations issued in previous loop iterations did not race. This assumption is clearly false in states where $size$ is greater than 2^{12}, thus states which do not satisfy the invariant $size \leq 2^{12}$ are not considered by the model checker as initial states when checking the step case.

With the implicit encoding, increasing k does not yield this invariant. This is because the implicit encoding tracks a *single*, arbitrary DMA operation. Consider a state where $size = 2^{12} + 1$ and $\mathrm{i}\%4 = 3$. Executing four iterations of the loop results in get operations targeting buffers 3, 0, 1 and 2, in that order. Call these operations get_3, get_0, get_1 and get_2. Since $size = 2^{12} + 1$, operation get_j targets the region $r_j = [j \times 2^{12}, (j + 1) \times 2^{12}]$ of `buf`. Consider a trace where get_3 is nondeterministically chosen for tracking. When get_0, get_1 and get_2 are issued, they are checked with respect to get_3. No race is detected for get_0 and get_1, since $r_3 \cap r_0 = \emptyset$ and $r_3 \cap r_1 = \emptyset$. However, $r_3 \cap r_2 = \{3 \times 2^{12}\}$, thus there is a DMA race between get_3 and get_2. For any k, we can construct a path consisting of k loop iterations, ending in a state where $size = 2^{12} + 1$ and $\mathrm{i}\%4 = 3$, along which no DMA is tracked. We can correctly assume that no DMA race is detected along such a path. However, as demonstrated above, the path can be extended so that a DMA race occurs. Thus, for any k, we have a counterexample to k-induction.

The essential point here is that with the explicit encoding, the combination of multiple DMA operations yields the invariant $size \leq 2^{12}$, whereas the implicit encoding considers operations in isolation, and does not derive this invariant.

The implicit encoding with k-induction fails on four further benchmarks, as indicated in Figure 10, for similar reasons. In all cases, the induction hypothesis can be strengthened so that k-induction succeeds for the implicit encoding, by adding a simple assertion to the body of the loop. For example, for the *checksum* benchmark, adding assert $size == 2^{12}$ to the loop body suffices. Experimental results for strengthened benchmarks are presented in rows marked 'strengthened' in Figure 10.

7.6 Comparison with predicate abstraction

The translation implemented by SCRATCH operates at the level of control flow graphs. In order to compare with other tools, we have hand-translated three of our benchmarks, *1-buf*, *2-buf* and *3-buf*, into C programs that track DMA operations as described in §5. We aimed to compare with BLAST [BHJM07] and SATABS [CKSY05] but were unable to obtain results using BLAST due to a bug in the tool, which we have reported to the BLAST developers.

Figure 13 shows results for proving correctness and finding bugs using SATABS (version 2.5), with Cadence SMV as a back-end model checker. For each example, we show the number of refinement iterations required (*iterations*), the time taken for verification (*time*), and the speed-up factor obtained by using SCRATCH over SATABS (obtained by comparing with the results of Figure 10). The explicit encoding is used in all cases. For buggy versions of *1-buf* and *2-buf*, SATABS is able to find the bug within 100 iterations, but is two orders of magnitude slower than SCRATCH when applied to *2-buf*. The abstraction-refinement process

leads to a conclusive verification result when applied to the correct version of *1-buf*, but is almost five times slower than our k-induction technique. SATABS was not able to find the bug in the buggy version of *3-buf*, or prove correctness of correct versions of *2-buf* or *3-buf* within 100 refinement iterations.

7.7 Comparison with IBM Race Check library

The IBM Cell SDK [IBM09] comes with a library for detecting DMA races [IBM08] at runtime. The library maintains a log of pending operations, checking each new operation against entries in the log in a manner similar to our explicit encoding. If a DMA race is detected, then an error message is written to the console.

Using a Sony PlayStation 3 console, which is equipped with a Cell processor, we tested the Race Check library on each of our buggy examples. DMA races are detected for all but two benchmarks, and race detection takes less than $0.1\,s$ in each case. The bug in *cpaudio* was not detected since the example runs on a specific input file that does not expose the bug. Although the buggy version of *1-buf I/O* crashes when executed on the Cell hardware, the Race Check library does not detect the DMA race responsible for this crash. This false negative appears to be a bug in the Race Check library rather than a fundamental limitation, since *1-buf I/O* is similar to examples where the Race Check library successfully detects DMA races.

Note that runtime race detection cannot be used to prove *absence* of DMA races, unlike our k-induction method.

8 Related work

The concept of k-induction was first published in [SSS00, BC00], targeting the verification of hardware designs represented by transition relations (although the basic idea had already been used in earlier implementations [LS99] and a version of one-induction used for BDD-based model checking [DM97]). A major emphasis of these two papers is on the restriction to loop-free or shortest paths, which we do not consider in our k-induction rule due to the size of state vectors and the high degree of determinism in software programs. Several optimizations and extensions to the technique have been proposed, including property strengthening to reduce induction depth [VH07], improving performance via incremental SAT solving [ES03], and supporting verification of temporal properties [AFF$^+$05]. Applications of k-induction have focused exclusively on hardware designs [SSS00, BC00, LS99], synchronous programs [HT08, Fra06] and, recently, SystemC designs [GLD10]. A principle related to k-induction has also been used for circular reasoning about liveness properties [McM99].

To the best of our knowledge, there has been no previous work on applying k-induction to imperative programs comparable to our procedure in §6. Our formulation of k-induction, and the explicit encoding of DMA operations, were first presented in [DKR10]. A technique for strengthening k-induction in the context of DMA race analysis, using static analysis techniques, has been proposed [DHK11]; this work sketches the implicit encoding of DMA operations presented in full in this paper. As discussed in §7.5, the implicit encoding of DMA operations may lead to assertions that are not k-inductive, where the explicit encoding would yield k-inductive assertions. The strengthening techniques of [DHK11] can be used

to overcome this weakness of the implicit encoding, automatically inferring the kinds of strengthening assertions discussed in §7.5. The SCRATCH tool is also described in [DKR11].

Techniques for detecting data races in shared memory multithreaded applications have been extensively studied. Notable static methods are based on formal type systems [FF00], or use classic pointer-analysis techniques; the latter approach is used by tools such as RACERX [EA03] and CHORD [NAW06]. The ERASER tool [SBN+97] uses binary rewriting to monitor shared variables and to find failures of the locking discipline at runtime. Other dynamic techniques include [FG05], which is based on state-less search with partial-order reduction, and [HMMCM06], which is based on a partial-order reduction technique for SystemC similar to the method of Flanagan and Godefroid [FG05]. None of these race detection techniques are applicable to software for heterogeneous multicore processors with multiple memory spaces. The only race detection tool we are aware of which is geared towards heterogeneous multicore is the IBM Race Check library [IBM08], which we compare with in §7. The speed of runtime race detection with this library is attractive, but the library requires access to commodity hardware and can only be used to find bugs which are revealed by a particular set of inputs. In contrast, our k-induction technique can prove *absence* of DMA races, and allows analysis to be carried out on any standard PC platform. Furthermore, BMC is able to detect potential races by assuming that input parameters may take *any* value.

Our focus in this paper has been on the use of formal analysis to aid programmers in writing correct DMA-based programs. An alternative approach is to relieve programmers of the need to write DMA operations by providing a higher-level programming formalism, where data-movement between memory spaces is not explicit. Low level, DMA-based code can then be automatically generated from high-level programs. High-level programming models for the Cell processor which avoid explicit DMA programming include Sequoia [FHK+06], CellSs [BPBL06], CellFS [INM09], Sieve C++ [DKL09] and Offload C++ [CDD+10, DDRR10]. A formal approach along these lines involves using session types to specify communication [YVPH08, HVY09]. Given a communications protocol expressed using session types, subtyping rules allow send/receive operations to be automatically reordered so that independent communications may be efficiently overlapped. We regard these high-level approaches as complimentary to our techniques. While higher-level programming formalisms are clearly desirable, there is always an associated performance trade-off: to achieve optimal performance on architectures like the Cell BE, it is typically necessary to write low-level code with explicit DMA operations, which our techniques can be used to analyze. Furthermore, our methods could be used to analyze the correctness of code generated from higher-level formalisms. Finally, the higher-level approaches discussed above come equipped with runtime libraries, which are written using low-level C, and could clearly benefit from formal verification using the techniques we have developed.

Our novel encoding of DMA races resembles the concept of prophecy variables [AL91]. Similar techniques have found application in other areas of formal verification, including liveness-to-safety rewriting for bounded model checking [SB06], reducing concurrent analysis to sequential analysis under a context bound [LR09, EQR11], and proving LTL properties of infinite-state programs [CK11].

9 Summary and Future Work

We have contributed an automatic technique for analyzing DMA races in heterogeneous multicore programs which manage scratch-pad memory. At the heart of our method is a novel formulation of k-induction. We have demonstrated the effectiveness of this technique

experimentally via a prototype tool, SCRATCH. SCRATCH is able to detect or prove absence of DMA races in a wide range of example programs for the Cell BE processor, handling examples which cannot be verified using current predicate abstraction tools, and finding bugs which go undiscovered by a runtime race checking library.

We plan to extend this work in the following ways.

We intend to generalize and make precise our intuitions as to why k-induction works well for DMA-based programs. Our vision is a set of conditions for identifying classes of programs amenable to verification by k-induction, thus making the technique more broadly applicable for software analysis.

SCRATCH focuses on analyzing DMA races for accelerator memory by analyzing accelerator source code in isolation. It is not possible to check meaningful properties of host memory without some knowledge of how this memory is structured. To check DMA races for host memory we plan to design a method which analyses host and accelerator source code side-by-side. A further challenge is the problem of DMA race checking between concurrently executing accelerator cores in a heterogeneous system. A starting point towards this goal could involve combining our methods with adapted versions of race checking techniques for shared memory concurrent software (*cf.* §8).

Our technique is currently limited to sequential analysis: we check memory safety for a *single* accelerator thread. This restriction allows us to check useful properties of DMA-based programs in a scalable manner. However, we would ideally like an analysis capable of handling concurrent threads, to detect or prove absence of DMA races between threads running on distinct accelerator cores. Naïvely applying a BMC-based approach in the concurrent setting is not a scalable solution, due to the exponential number of thread interleavings that must be considered. We plan to investigate the use of BMC techniques specifically tailored towards concurrent analysis, as proposed in [Cor10]. Preliminary results for a recent, complementary approach to race analysis for concurrent programs using asynchronous memory operations using separation logic also show promise [BDDP11].

Acknowledgment

We are grateful to Matko Botinčan, Leopold Haller and the anonymous reviewers for their comments on an earlier draft of this work.

References

[AFF+05] 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.

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 2006.

[BC00] Per Bjesse and Koen Claessen. SAT-based verification without state space traversal. In *FM-CAD*, volume 1954 of *LNCS*, pages 372–389. Springer, 2000.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[BDDP11] Matko Botincan, Mike Dodds, Alastair F. Donaldson, and Matthew J. Parkinson. Automatic safety proofs for asynchronous memory operations. In *PPOPP*, pages 313–314. ACM, 2011.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *STTT*, 9(5-6):505–525, 2007.

[BPBL06] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. CellSs: a programming model for the Cell BE architecture. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC)*, page 86. ACM, 2006.

[CDD+10] Pete Cooper, Uwe Dolinsky, Alastair F. Donaldson, Andrew Richards, Colin Riley, and George Russell. Offload – automating code migration to heterogeneous multicore systems. In *HiPEAC*, volume 5952 of *LNCS*, pages 337–352. Springer, 2010.

[CFMS09] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. In *ASE*, 2009.

[CK11] Byron Cook and Eric Koskinen. Making prophecies with decision predicates. In *POPL*, pages 399–410. ACM, 2011.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[CKSY05] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, volume 3440 of *LNCS*, pages 570–574. Springer, 2005.

[Cor10] Lucas Cordeiro. SMT-based bounded model checking for multi-threaded software in embedded systems. In *ICSE (2)*, pages 373–376. ACM, 2010.

[DDRR10] Alastair F. Donaldson, Uwe Dolinsky, Andrew Richards, and George Russell. Automatic offloading of C++ for the Cell BE processor: a case study using Offload. In *MuCoCoS*, pages 901–906. IEEE, 2010.

[DHK11] Alastair F. Donaldson, Leopold Haller, and Daniel Kroening. Strengthening induction-based race checking with lightweight static analysis. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 6538 of *LNCS*, pages 169–183. Springer, 2011.

[DKL09] Alastair F. Donaldson, Paul Keir, and Anton Lokhmotov. Compile-time and run-time issues in an auto-parallelisation system for the Cell BE processor. In *Euro-Par 2008 Workshops*, volume 5415 of *LNCS*, pages 163–173. Springer, 2009.

[DKR10] Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer. Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, volume 6015 of *LNCS*, pages 280–295. Springer, 2010.

[DKR11] Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer. SCRATCH: a tool for automatic analysis of DMA races. In *PPOPP*, pages 311–312. ACM, 2011.

[DM97] David Déharbe and Anamaria Martins Moreira. Using induction and BDDs to model check invariants. In *CHARME*, volume 105 of *IFIP Conference Proceedings*, pages 203–213. Chapman & Hall, 1997.

[EA03] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252. ACM, 2003.

[EQR11] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. Delay-bounded scheduling. In *POPL*, pages 411–422. ACM, 2011.

[ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4), 2003.

[FF00] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *PLDI*, pages 219–232. ACM, 2000.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121. ACM, 2005.

[FHK+06] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In *Supercomputing (SC)*, page 83. ACM, 2006.

[Fra06] Anders Franzén. Using satisfiability modulo theories for inductive verification of Lustre programs. *Electr. Notes Theor. Comput. Sci.*, 144(1):19–33, 2006.

[GLD10] Daniel Große, Hoang M. Le, and Rolf Drechsler. Proving transaction and system-level properties of untimed SystemC TLM designs. In *MEMOCODE*, pages 113–122. IEEE, 2010.

[Har80] David Harel. On folk theorems. *Commun. ACM*, 23(7):379–389, 1980.

[HMMCM06] C. Helmstetter, F. Maraninchi, L. Maillet-Contoz, and M. Moy. Automatic generation of schedulings for improving the test coverage of systems-on-a-chip. In *FMCAD*, pages 171–178. IEEE, 2006.

[Hof05] H. Peter Hofstee. Power efficient processor architecture and the Cell processor. In *HPCA*, pages 258–262. IEEE, 2005.

[HT08] George Hagen and Cesare Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *FMCAD*, pages 109–117. IEEE, 2008.

[HVY09] Kohei Honda, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Type-directed compilation for multicore programming. *Electr. Notes Theor. Comput. Sci.*, 241:101–111, 2009.

[IBM08] IBM. *Example Library API Reference, version 3.1*, July 2008.

[IBM09] IBM. Cell BE resource center, October 2009. `http://www.ibm.com/developerworks/power/cell/`.

[INM09] Latchesar Ionkov, Aki Nyrhinen, and Andrey Mirtchovski. CellFS: Taking the "DMA" out of Cell programming. In *IPDPS*, pages 1–8. IEEE, 2009.

[KS03] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 2575 of *LNCS*, pages 298–309. Springer, 2003.

[LR09] Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.

[LS99] Carl Johan Lillieroth and Satnam Singh. Formal verification of FPGA cores. *Nord. J. Comput.*, 6(3):299–319, 1999.

[McM99] Kenneth L. McMillan. Circular compositional reasoning about liveness. In *CHARME*, volume 1703 of *LNCS*, pages 342–345. Springer, 1999.

[Moo98] Gordon E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86:82–85, 1998.

[NAW06] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *PLDI*, pages 308–319. ACM, 2006.

[SB06] Viktor Schuppan and Armin Biere. Liveness checking as safety checking for infinite state spaces. *Electr. Notes Theor. Comput. Sci.*, 149(1):79–96, 2006.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[SSS00] 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.

[VH07] Vishnu C. Vimjam and Michael S. Hsiao. Explicit safety property strengthening in SAT-based induction. In *VLSID*, pages 63–68. IEEE, 2007.

[YVPH08] Nobuko Yoshida, Vasco Thudichum Vasconcelos, Hervé Paulino, and Kohei Honda. Session-based compilation framework for multicore programming. In *FMCO*, volume 5751 of *LNCS*, pages 226–246. Springer, 2008.

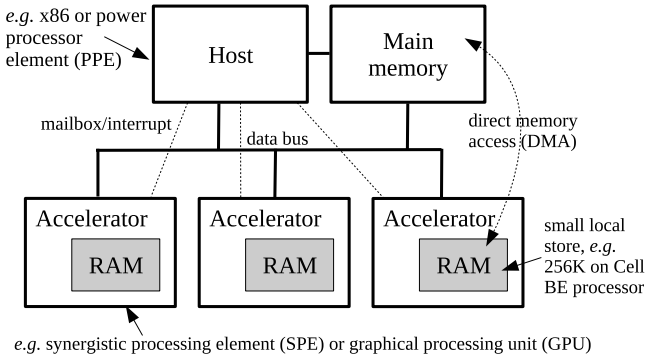

Fig. 1 Structure of a heterogeneous multicore architecture consisting of a host core with a number of accelerators. Each accelerator core is equipped with a single scratch-pad memory, to which it has exclusive access. Direct memory access (DMA) is used to transfer data between main and scratch-pad memory

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

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

void process_data(float* buf) { ... }  // Unspecified data-processing procedure

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]);                       // Wait for and process
    process_data(buffers[0]);            //   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;

    }

    put(buffers[put_buf], out, CHUNK, tags[put_buf]); // Put data processed during
    out += CHUNK;                                     //     final loop iteration
    wait(tags[process_buf]);                          // Wait and process final
    process_data(buffers[process_buf]);               //    chunk of data
    put_buf = process_buf;
    put(buffers[put_buf], out, CHUNK, tags[put_buf]); // Put final result
    wait(tags[put_buf]);      // Wait for transfer of final result to complete
}
```

Fig. 2 Source code of the triple-buffering example

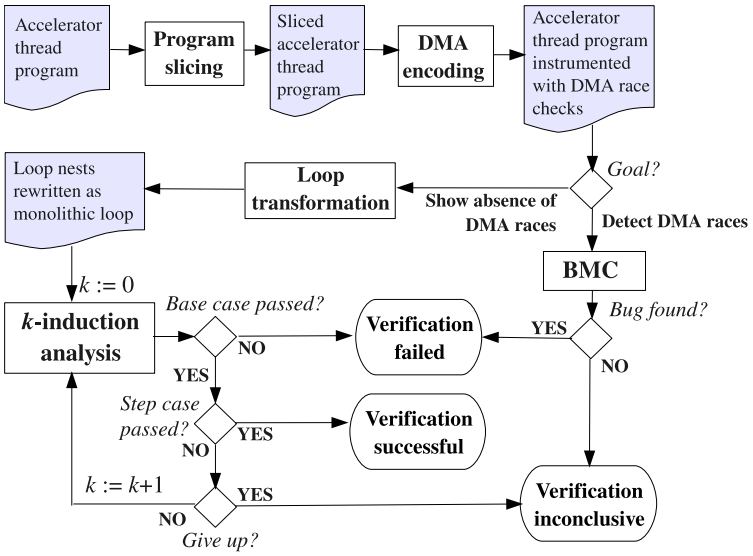

Fig. 3 Overview of our DMA race analysis method

Statement	Translated form	Notes
program start	$\forall_{0 \le j < D}$ assume $\neg valid[j]$;	No pending DMAs initially
get(l, h, s, t)	assert $0 \le s \le M \wedge 0 \le t < T$; $\forall_{0 \le j < D}$ assert $\neg valid[j] \vee$ (disjoint$(l, s, local[j], size[j]) \wedge$ $(is\_get[j] \vee$ disjoint$(h, s, host[j], size[j])))$; assert $\neg(valid[0] \wedge \cdots \wedge valid[D-1])$; $i := *$; assume $0 \le i < D \wedge \neg valid[i]$; $valid[i] := 1$; $is\_get[i] := 1$; $local[i] := l$; $host[i] := h$; $size[i] := s$; $tag[i] := t$;	Check size/tag within range Consider every position in log Either: no DMA at position j, or no race on local regions and no race on host regions Assert log not full Pick arbitrary free position Log details of new DMA
put(l, h, s, t)	assert $0 \le s \le M \wedge 0 \le t < T$; $\forall_{0 \le j < D}$ assert $\neg valid[j] \vee$ (disjoint$(h, s, host[j], size[j]) \wedge$ $(\neg is\_get[j] \vee$ disjoint$(l, s, local[j], size[j])))$; assert $\neg(valid[0] \wedge \cdots \wedge valid[D-1])$; $i := *$; assume $0 \le i < D \wedge \neg valid[i]$; $valid[i] := 1$; $is\_get[i] := 0$; $local[i] := l$; $host[i] := h$; $size[i] := s$; $tag[i] := t$;	Similar to get(l, h, s, t) Roles of local/host regions reversed in race check
wait(t)	assert $0 \le t < T$; $\forall_{0 \le j < D}$ $valid[j] := valid[j] \wedge \neg(t = tag[j])$	Check tag within range Remove operations with tag t from log

Fig. 4 Explicit encoding of DMA operations. The rules translate DMA operations into assertions/assignments over tracker arrays of size D

Statement	Translated form	Notes
program start	assume $\neg valid$;	Initially no DMA is tracked
get(l, h, s, t)	assert $0 \le s \le M \wedge 0 \le t < T$; assert $\neg valid \vee$ (disjoint$(l, s, local, size)$ $\wedge(is_get \vee$ disjoint$(h, s, host, size)))$; goto $track$, $after$ $track$: $valid := 1$; $is\_get := 1$; $local := l$; $host := h$; $size := s$; $tag := t$; $after$: ...	Check size/tag within range Check new DMA does not race with tracked DMA, if any Nondeterministically choose whether to track new DMA Log details of new DMA
put(l, h, s, t)	assert $0 \le s \le M \wedge 0 \le t < T$; assert $\neg valid \vee$ (disjoint$(h, s, host, size)$ $\wedge(\neg is_get \vee$ disjoint$(l, s, local, size)))$; goto $track$, $after$ $track$: $valid := 1$; $is\_get := 0$; $local := l$; $host := h$; $size := s$; $tag := t$; $after$: ...	Similar to get(l, h, s, t) Roles of local/host regions reversed in race check
wait(t)	assert $0 \le t < T$; assume $\neg(tag = t)$;	Check t within range; assume no DMA with this tag was tracked

Fig. 5 An implicit encoding of DMA operations. During execution, at most a single, nondeterministically chosen DMA operation is tracked

```
1: get(l, h, s, t);
2: wait(t);
3: put(l, h + s, s, t);
4: wait(t);
```

(a) A race-free program

```
1: get(l, h, s, t);
2: put(l, h + s, s, t);
3: wait(t);
```

(b) Program with a potential DMA race due to use of same local store region by get and put, with no intervening wait

Fig. 6 Two simple programs using DMA operations

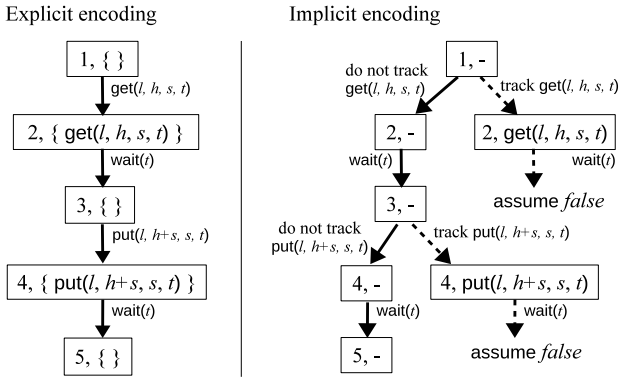

Fig. 7 A comparison of traces for the race-free program of Figure 6(a) obtained via the explicit encoding of Figure 4 (left) and the implicit encoding of Figure 5 (right)

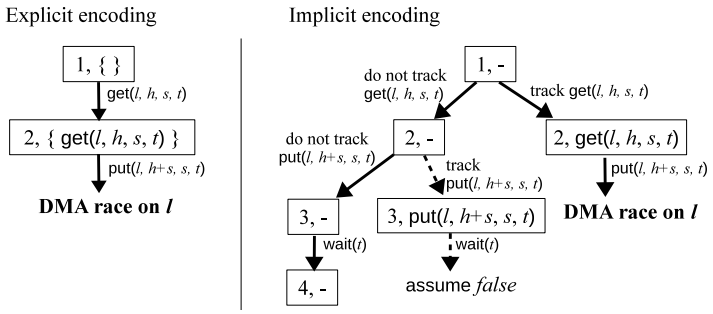

Fig. 8 A comparison of traces for the program of Figure 6(b), which leads to a potential DMA race, obtained via the explicit encoding of Figure 4 (left) and the implicit encoding of Figure 5 (right)

Statement	Translated form	Notes
program start	assume $\neg valid$;	Initially no DMA is tracked
$\mathsf{get}(l,h,s,t)$	assert $0 \le s \le M \wedge 0 \le t < T$; assert $\neg valid \vee$ $(protected \wedge tag = t) \vee$ $(\mathsf{disjoint}(l,s,local,size) \wedge$ $(is\_get \vee \mathsf{disjoint}(h,s,host,size)))$; goto $track, after$; $track$: $valid := 1$; $is\_get := 1$; $local := l$; $host := h$; $size := s$; $tag := t$; $protected := 0$; $after$: ...	Check size/tag within range Check either: a) tracked DMA protected by barrier and shares tag with new DMA, or (b) no race between new/tracked DMAs Nondeterministically choose whether to track new DMA Log details of new DMA DMA not initially protected by a barrier
$\mathsf{put}(l,h,s,t)$	assert $0 \le s \le M \wedge 0 \le t < T$; assert $\neg valid \vee$ $(protected \wedge tag = t) \vee$ $(\mathsf{disjoint}(h,s,host,size) \wedge$ $(\neg is\_get \vee \mathsf{disjoint}(l,s,local,size)))$; goto $track, after$; $track$: $valid := 1$; $is\_get := 1$; $local := l$; $host := h$; $size := s$; $tag := t$; $protected := 0$; $after$: ...	Similar to $\mathsf{get}(l,h,s,t)$ Roles of local/host regions reversed in race check
$\mathsf{getf}(l,h,s,t)$	assert $0 \le s \le M \wedge 0 \le t < T$; assert $\neg valid \vee tag = t \vee$ $(\mathsf{disjoint}(l,s,local,size) \wedge$ $(is\_get \vee \mathsf{disjoint}(h,s,host,size)))$; goto $track, after$; $track$: $valid := 1$; $is\_get := 1$; $local := l$; $host := h$; $size := s$; $tag := t$; $protected := 0$; $after$: ...	Similar to $\mathsf{get}(l,h,s,t)$, but fence allows new DMA and tracked DMA memory regions to overlap if both DMAs use the same tag Nondeterministically choose whether to track new DMA Log details of new DMA DMA not initially protected by a barrier
$\mathsf{putf}(l,h,s,t)$	assert $0 \le s \le M \wedge 0 \le t < T$; assert $\neg valid \vee tag = t \vee$ $(\mathsf{disjoint}(h,s,host,size) \wedge$ $(\neg is\_get \vee \mathsf{disjoint}(l,s,local,size)))$; goto $track, after$; $track$: $valid := 1$; $is\_get := 1$; $local := l$; $host := h$; $size := s$; $tag := t$; $protected := 0$; $after$: ...	Similar to $\mathsf{getf}(l,h,s,t)$ Roles of local/host regions reversed in race check
$\mathsf{getb}(l,h,s,t)$	$protected := (tag = t \; ? \; 1 : protected)$; translation for $\mathsf{get}(l,h,s,t)$	Barrier protects tracked DMA if tags match
$\mathsf{putb}(l,h,s,t)$	$protected := (tag = t \; ? \; 1 : protected)$; translation for $\mathsf{put}(l,h,s,t)$	Similar
$\mathsf{wait}(t)$	assert $0 \le t < T$; assume $\neg(tag = t)$;	Check tag within range, and assume no DMA with this tag was tracked

Fig. 9 Rules to translate DMA operations into assertions and assignments to tracker arrays, with support for fences and barriers

	Benchmark	Explicit encoding					Implicit encoding			
		D	time	k	vars	clauses	time	k	vars	clauses
1	race check 1	2	0.15	0	1409	3518	0.15	0	794	1625
2	race check 2	4	0.17	0	3893	11550	0.15	0	1669	4616
3	sync atomic op	1	0.19	1	8212	22552	0.22	1	7074	19628
4	sync mutex	1	0.22	1	7517	21301	0.23	1	6465	19599
5	simple dma	1	0.15	0	3585	9857	0.14	0	3441	9453
6	1-buf	1	0.31	1	7968	21994	0.34	1	6861	19163
7	1-buf I/O	1	0.40	1	7973	22011	0.43	1	6866	19180
8	2-buf	2	0.56	1	28524	84783	1.04	2	23591	70828
9	2-buf + fence	4	1.75	3	65566	202220	1.02	2	18038	54268
10	2-buf + barrier	4	1.86	3	66322	205724	1.03	2	18187	54964
11	2-buf I/0	4	2.20	3	65064	199696	1.23	2	17950	53807
12	3-buf	3	5.89	3	76835	236073	2.90	3	35335	108929
13	3-buf + fence	5	11.81	3	86578	269935	2.63	3	27931	86849
14	3-buf I/O	2	4.17	4	70445	216902	-	-	-	-
	strengthened	2	1.94	3	57772	177478	2.60	3	35475	109597
15	3-buf I/O + fence	4	5.71	4	87727	274294	-	-	-	-
	strengthened	4	2.42	3	72519	226087	2.41	3	28071	87517
16	3-buf I/O + barrier	4	8.03	4	88779	279190	-	-	-	-
	strengthened	4	2.54	3	73415	230263	2.43	3	28294	88561
17	cpaudio	4	2.93	3	108418	327857	1.69	2	25886	76796
18	normalize	8	13.85	3	262822	812394	12.32	4	54889	163873
19	checksum	4	2.07	4	64328	195350	-	-	-	-
	strengthened	4	1.91	4	64339	195413	1.24	4	23543	71356
20	Euler simple	5	1.60	2	79057	253441	-	-	-	-
	strengthened	5	1.75	2	74262	239056	1.21	2	17467	54451
21	Euler complex	10	38.75	3	495504	1596834	3.89	2	50855	161300
22	Julia 2	3	72.08	8	337622	981717	43.80	7	207616	590011

Fig. 10 Benchmark results for proving correctness using k-induction, with explicit and implicit encodings of DMA operations

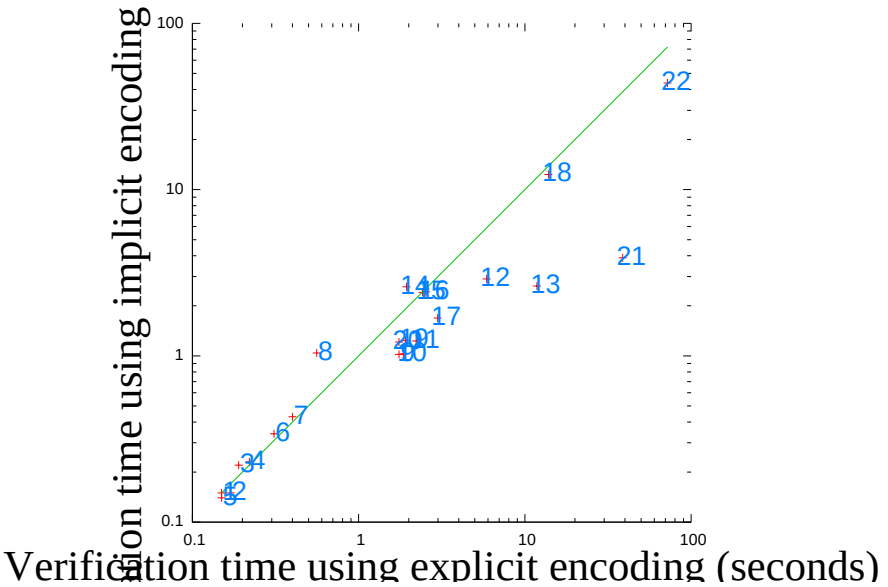

Fig. 11 Comparison of explicit and implicit encodings of DMA operations when proving correctness using k-induction, using a logarithmic scale. Numbers are used to identify benchmarks according to the row numbers of Figure 10

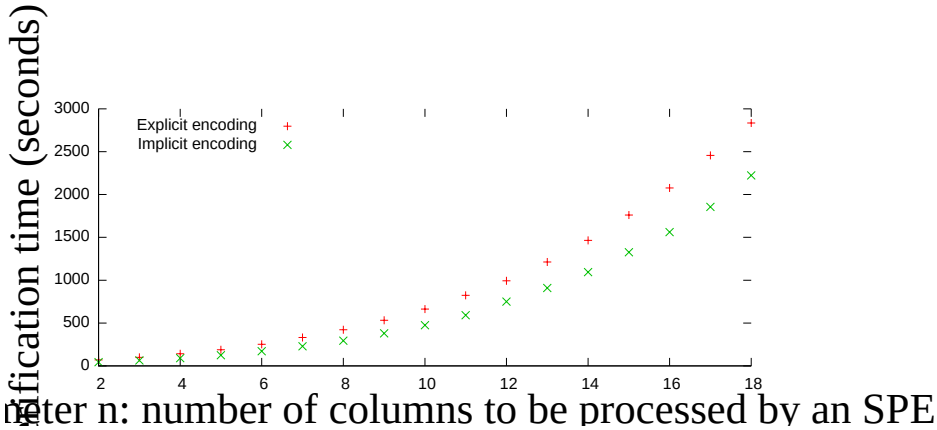

Fig. 12 Verification time for the *Julia* benchmark increases cubically with k. The values of k required to prove correctness with the explicit and implicit encodings are $k = n + 6$ and $k = n + 5$ respectively, where n is the number of image columns processed by an SPE

Benchmark	iterations	time	SCRATCH speedup
1-buf correct	15	4.49	14.48 ×
2-buf correct	>100	>1025.27	>1830.83 ×
3-buf correct	>100	>2646.87	>449.38 ×
1-buf buggy	3	0.39	2.07 ×
2-buf buggy	20	24.21	93.11 ×
3-buf buggy	>100	>5226.91	>14934.02 ×

Fig. 13 Results applying CEGAR-based verification to three of the Cell SDK examples using SATABS in comparison to SCRATCH. The explicit encoding of DMA operations is used