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

Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer

Oxford University Computing Laboratory, Oxford, UK

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 programming complexity – the programmer must manually orchestrate data movement using direct memory access (DMA) operations. Programming using asynchronous DMAs is error-prone, and *DMA races* can lead to nondeterministic bugs which are hard to reproduce and fix. We present a method for DMA race analysis which automatically instruments the program with assertions modelling the semantics of a memory flow controller. To enable automatic verification of instrumented programs, we present a new formulation of k-induction geared towards software, as a proof rule operating on loops. We present 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 for heterogeneous multicore processors.

1 Introduction

Heterogeneous multicore processors such as the Cell Broadband Engine (BE) circumvent the shared memory bottleneck by equipping cores with small "scratch-pad" memories [1, 2]. These fast, private memories are not coherent with main memory, and allow independent calculations to be processed in parallel by separate cores without contention. While this can boost performance,[1] 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 refer to the same portion of memory, and at least one modifies the memory. There is an urgent need for techniques and tools to analyse DMA races, which, if undetected, can lead to nondeterministic bugs that are difficult to reproduce and fix.

* 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.

[1] A supercomputer comprised of Cell processors recently assumed #1 spot on the Top 500 list.

We present a method for DMA race analysis which automatically instruments the program with assertions modelling the semantics of a memory flow controller. The instrumented programs are amenable to automatic verification by state-of-the-art model checkers. Recent dramatic advances in SAT/SMT techniques have led to widespread use of Bounded Model Checking (BMC) [3, 4] for finding bugs in software. 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 [5] for the property, which is often prohibitively large. We overcome this limitation by presenting a novel formulation of k-induction [6]. 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. We investigate heuristics to further boost the applicability of k-induction when checking for DMA races, and discuss limitations of k-induction in this application domain.

We have implemented 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 [2], in which we discover a previously unknown bug, which has been independently confirmed. Our experiments 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 automatic technique for instrumenting programs with assertions to check for DMA races, enabling verification of multicore programs with scratch-pad memory.
- a new 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.

To our knowledge, this 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 memory. A

DMA operation[2] 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 – an accelerator thread which issues a DMA continues executing while the operation is handled by a specialised piece of hardware called a *memory flow controller*. An accelerator thread can issue a *wait* operation, specifying a tag t, which causes execution to block until all DMAs identified by t have completed. A DMA with tag t is *pending* until a wait operation with tag t is issued.

Although a DMA *may* complete before an explicit wait operation is issued, this cannot be guaranteed, thus access by the host or accelerator to memory that is due to be modified by a pending DMA should be regarded as a bug. Failure to issue a wait operation may result in nondeterministic behaviour: it may *usually* be the case that the required data has arrived, but occasionally the lack of a wait may result in reading from uninitialised memory, leading to incorrect computation. This nondeterminism means that bugs arising due to misuse of DMA can be extremely difficult to reproduce and fix.

2.1 DMA primitives and properties of interest

We consider the following 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

For each accelerator core, we assume hardware-imposed maximum values D and M for the number of DMAs that may be pending simultaneously and the number of bytes that may be transferred by a single DMA, respectively. We assume that tags are integers in the range $[0, D-1]$. On the Cell processor, $D = 32$ and $M = 16384$ (16K).

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 reasons of space 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.

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:*

[2] For brevity, we sometimes write "DMA" rather than "DMA operation".

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

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

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

void triple_buffer(char* in, char* out, int num_chunks) {
    unsigned int tags[3] = { 0, 1, 2 }, tmp, put_buf, get_buf, process_buf;

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

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

Fig. 1. Triple-buffering

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

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.

2.2 Illustrative example: triple-buffering

Figure 1, adapted from an example provided with the IBM Cell SDK [2], 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.

$$\mathsf{get}(\texttt{buffers[0]}, \ldots, \texttt{CHUNK}, \texttt{tags[0]}) \quad (1)$$
$$\mathsf{get}(\texttt{buffers[1]}, \ldots, \texttt{CHUNK}, \texttt{tags[1]}) \quad (2)$$

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

At this point in execution the operations marked (*) 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).

We discovered this bug using SCRATCH, our automatic DMA analysis tool, described in §6, 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 independently confirmed. In the remainder of the paper, we present the new techniques of SCRATCH that enable these results.

3 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 1. 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 arrays 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:

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

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

The language includes assertions, nondeterminisic 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 $\alpha = 1\text{: } \alpha_1; \ldots; n\text{: } \alpha_n$ is the graph (S, E_α). $S = \{(\sigma, pc) \mid \sigma : (X \cup (A \times \mathbb{Z})) \to \mathbb{Z},\ pc \in \mathbb{Z}\} \cup \{\lightning\}$ is the set of program states, where 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. E_α is the set of transitions (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):

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

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 $\lightning$.

4 Encoding DMA operations in goto programs

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

$$\mathit{Stmt} \;::=\; \ldots \;\big|\; \mathsf{get}(e, e, e, e) \;\big|\; \mathsf{put}(e, e, e, e) \;\big|\; \mathsf{wait}(e)$$

For a goto program with DMAs, we introduce a series of array variables with size D (see §2.1), which we call *tracker arrays*. These "ghost variables" log the state of up to D pending DMA operations during program execution. The tracker arrays are as follows, with $0 \leq j < D$:

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

To check properties of DMA operations we translate a program with DMA primitives into a standard goto program, where get, put and wait operations are replaced with

[3] In our context, this is preferable to modelling failed assumptions via a distinguished "blocked program" state: it simplifies the notion of sequential composition of programs (*cf.* §5.1).

Statement	Translated form
start of program	$\forall_{0 \le j < D}\ assume\ valid[j] = 0;$
$\mathsf{get}(l, h, s, t)$	$assert\ 0 \le s \le M \wedge 0 \le t < D;$ $\forall_{0 \le j < D}\ assert\ \neg valid[j] \vee (disjoint(l, s, local[j], size[j]) \wedge$ $(is\_get[j] \vee disjoint(h, s, host[j], size[j])));$ $assert\ \neg(valid[0] \wedge valid[1] \wedge \cdots \wedge valid[D-1]);$ $i := *;\ assume\ 0 \le i < D \wedge \neg valid[i];$ $valid[i] := 1;\ is\_get[i] := 1;\ local[i] := l;\ host[i] := h;\ size[i] := s;$
$\mathsf{put}(l, h, s, t)$	$assert\ 0 \le s \le M \wedge 0 \le t < D;$ $\forall_{0 \le j < D}\ assert\ \neg valid[j] \vee (disjoint(h, s, host[j], size[j]) \wedge$ $(\neg is\_get[j] \vee disjoint(l, s, local[j], size[j])));$ $assert\ \neg(valid[0] \wedge valid[1] \wedge \cdots \wedge valid[D-1]);$ $i := *;\ assume\ 0 \le i < D \wedge \neg valid[i];$ $valid[i] := 1;\ is\_get[i] := 0;\ local[i] := l;\ host[i] := h;\ size[i] := s;$
$\mathsf{wait}(t)$	$assert\ 0 \le t < D;$ $\forall_{0 \le j < D}\ valid[j] := valid[j] \wedge \neg(t = tag[j])$

Fig. 2. Rules to translate DMA operations into assertions and assignments to tracker arrays. We use $disjoint(a_1, s_1, a_2, s_2)$ as shorthand for $a_1 + s_1 \le a_2 \vee a_2 + s_2 \le a_1$

assertions about and assignments to the tracker arrays. The translation rules are given in Figure 2. We use $\forall_{0 \le j < D}$ to indicate that the following statement should be duplicated D times with increasing values for j. Since the rules of Figure 2 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 ensure that correct programs cannot issue DMA operations 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. The Cell processor supports further DMA primitives involving *fences* and *barriers*. Our implementation (§6) supports these operations via extensions of the rules in Figure 2; we do not present the extended rules due to lack of space.

5 k-Induction for goto programs

Our encoding of DMA programs is directly amenable to Bounded Model Checking [3] 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 [6], 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 program satisfies a safety property ϕ requires showing that (i) some formula I (which often is identical to ϕ) holds in all initial states, (ii) I is preserved by all state transitions of the program (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 program execution. In return, (i) has to be strengthened appropriately.

We describe the principle using the notation of [7]. Let $\mathbf{I}(s)$ and $\mathbf{T}(s, s')$ be formulae encoding the initial states and transition relation for a finite state system, 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, important refinements which boost the applicability of k-induction to hardware designs, such as the restriction to loop-free paths [6], are not useful when dealing with software where the state-vector is very large.

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.

5.1 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.

A program α is *self-contained*, denoted $\mathit{contained}(\alpha)$, if, for each goto statement $i\colon \mathit{goto}\ \ldots, l, \ldots$ 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.

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 $\alpha_{\mathit{assume}} = 1\colon \alpha'_1; \ldots; n\colon \alpha'_n$ is defined by: $\alpha'_i = \mathit{assume}\ \phi$ if $\alpha_i = \mathit{assert}\ \phi$, 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:

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

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 $\mathit{assume}\ b$ as the first statement of the loop body, and $\mathit{assume}\ \neg b$ 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.

Proof rule for k-induction

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

In this rule, the assertions present in the program (*e.g.* the formulae in Figure 2) 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 $\beta^{k}_{\mathit{assume}} \fatsemi \beta$ *is correct* and $\beta^{k}_{\mathit{assume}} \fatsemi \gamma$ *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 2 (Correctness). *The above proof rule is sound.*

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 as well be used in any (possibly interactive) deductive verification system.

5.2 Heuristics to aid k-induction for DMA programs

Through our experiments in §6 we observe that k-induction works extremely well for checking assertions representing DMA race-freeness, generated by the rules in Figure 2. For realistic example programs written for the Cell processor, the generated assertions are 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. 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 [8].

For less regular examples, our practical experience has led to the following heuristics which can be applied to help k-induction succeed, or to quickly determine when the technique is unlikely to work. These heuristics are merely optimisations to our technique; we are able to verify all benchmarks presented in §6 *without* use of heuristics.

Bounded lifetimes In practice, the programmer often knows that no DMA operation should pend for more than a small number (Z, say) of loop iterations. To take advantage of this domain specific information, the tracker arrays can be extended with a component to record the number x of enclosing loop iterations for which a DMA has been pending, asserting that x never exceeds Z. When proving the step case for $k > Z$, this allows the assumption that only DMAs issued within the last Z iterations are tracked, eliminating many unreachable states which might otherwise cause the step case to fail.

Free slots While it is legal for up to D operations to be pending simultaneously, most practical applications require significantly fewer simultaneous DMAs. Adding an assertion to the start of the loop body requiring at least Z free slots in the tracker arrays (for some $Z > 0$) can help k-induction to succeed when it otherwise would not.

Bounded periods of inactivity Generally, to prove that a DMA operation is race free, it is necessary to be able to assume that the operation was race free on a previous loop iteration. If a DMA statement might not to be executed for an *arbitrary* number of loop iterations then k-induction is unlikely to work. By introducing extra instrumentation to check that each DMA statement is executed at least once every Z iterations (for some $Z > 0$) we can set up reasonable conditions under which k-induction "gives up", resulting in a base case failure identifying a problematic DMA statement.

6 Experimental evaluation

We have implemented a prototype tool, SCRATCH[4], built on top of the CBMC model checker [4]. SCRATCH accepts an arbitrary C program written for an SPE core of the Cell BE processor, and checks for DMA races involving local memory. The translation described in §4 is applied to transform the input program into a form where DMAs are replaced with assertions and assignments to tracker arrays. BMC can be applied to the resulting program to check for DMA races up to a certain depth, and combined with k-induction, using the formulation of §5, to prove absence of races. Although our k-induction method is, in principle, applicable to arbitrary nested loops, for implementation convenience SCRATCH currently applies k-induction only to single loops. We are able to analyse many interesting examples with this restriction, in some cases by converting a nest of two loops into a single loop.

We evaluate SCRATCH using a set of 22 benchmarks adapted from examples supplied with the IBM Cell SDK for Multicore Acceleration [2], categorized as follows:

- **x-buf ($x \in \{1, 2, 3\}$)** Data processing programs which use single-, double- or triple-buffering for data-movement (*cf.* Figure 1). I/O indicates that separate buffers are used for input and output. Some variants of these programs use fences/barriers
- **race check, simple dma** Examples which illustrate data races and use of DMA

[4] SCRATCH is available online at http://www.cprover.org/scratch/.

Benchmark	Correct			Buggy			Benchmark	Correct			Buggy		
	k	D	time	D	time	depth		k	D	time	D	time	depth
race check 1	0	2	0.35	1	0.94	34	cpaudio	3	4	5.83	1	0.99	57
race check 2	0	4	0.35	3	0.95	65	3-buf I/O	3	4	12.29	2	0.67	133
sync atomic op	1	1	0.39	1	0.33	64	2-buf + barrier	3	4	3.23	2	0.56	130
sync mutex	1	1	0.43	1	0.34	74	2-buf I/0	3	4	3.53	3	0.76	137
simple dma	1	1	0.39	1	0.36	80	3-buf + fence	3	5	35.94	3	0.7	184
1-buf	1	1	0.41	1	0.43	100	normalize	3	8	71.74	12	2.34	549
1-buf I/O	1	1	0.44	2	0.54	109	Euler complex	3	10	420.54	8	3.91	273
2-buf	1	2	0.66	2	0.54	87	3-buf I/O + barrier	4	2	9.65	3	0.68	160
2-buf + fence	2	4	1.39	2	0.37	130	3-buf I/O + fence	4	4	12.99	3	0.68	159
Euler simple	2	5	4.79	3	1.32	167	checksum	4	4	3.49	4	0.59	53
3-buf	3	3	15.84	3	0.65	160	Julia 2	7	3	32.75	32	2783.4	1955

Fig. 3. Results using SCRATCH for proving correctness via k-induction, and for bug-finding, on Cell SDK benchmarks

- **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

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.

Figure 3 shows results applying SCRATCH to correct and buggy versions of the benchmarks[5]. 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 §2.2, 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. For each benchmark, we give the smallest value of k for which correctness can be proved using k-induction (*without* employing the heuristics of §5.2); the minimum number D of DMAs which it was necessary to track (setting D to a low value reduces the size of the tracker arrays, which can significantly reduce verification complexity; we compute the optimum value for D iteratively for each benchmark,

[5] Experiments are performed on a 3GHz Intel Xeon machine running Linux 2.6 (64-bit).

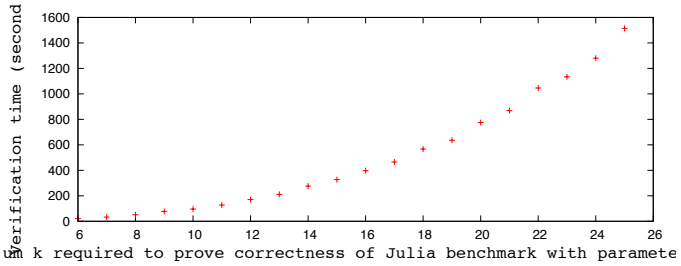

Fig. 4. Verification time for the *Julia* benchmark increases cubically with k

starting with $D = 1$), and the time, in seconds, taken for verification. We also show the smallest depth of execution required for bug-finding. Results are ordered with respect to k, Correct-D and Buggy-D. MiniSat 1.14, compiled with full optimisations, is used as a back-end SAT solver. It has been reported to perform comparatively to state-of-the-art SMT solvers for SMT-$\mathcal{BV}$ [9] on this type of workload.

The results of Figure 3 indicate that k-induction provides a tractable method for proving correctness for this set of benchmarks: verification is achieved in under 10 seconds for 15 of the 22 examples, with only *Euler complex* taking longer than two minutes to check. The *normalize* and *Euler complex* benchmarks require the largest values for D, and result in the largest SAT instances for the correct programs, taking the longest time to verify. 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 $k = n+5$ (the results in Figure 3 are for the case where $n = 2$). In Figure 4, we illustrate the scalability of k-induction by plotting the time taken for verification of the Julia benchmark against the size of k when we vary parameter n between 1 and 25. Growth is less than cubic, showing that our k-induction method scales well.

With the exception of *Julia*, bug-finding is fast, taking less than 4 seconds. The *Julia* benchmark is the only example where the bug leads to unbounded issuing of non-interfering DMAs. Thus an assertion fails only when an attempt is made to issue a DMA operation when 32 operations are already pending. This situation requires a large search depth to detect, resulting in a SAT instance with more than 1.5 million variables which takes considerable time to solve. The "bounded lifetimes" heuristic of §5.2 can be used to short-circuit the bug-finding process for this example. Requiring that no DMA pends for more than three loop iterations (which is the case for the correct version of this benchmark), bug-finding takes just $1.88\,s$, requiring a search depth of 901.

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 §4. We aimed to compare with BLAST [10] and SATABS [11] but were unable to obtain results using BLAST due to a bug in the tool, which we have reported to the BLAST developers.

	Correct			Buggy		
Benchmark	iterations	time	SCRATCH speedup	iterations	time	SCRATCH speedup
1-buf	15	9.49	23.14 ×	3	1.25	2.91 ×
2-buf	>100	>1352.43	>417.78 ×	20	33.62	59.97 ×
3-buf	>100	>4344.98	>120.9 ×	69	4969.03	6641.47 ×

Fig. 5. Results applying CEGAR-based verification to three of the Cell SDK examples using SatAbs, in comparison to bounded model checking with k-induction using CBMC

Figure 5 shows results for proving correctness and finding bugs using SATABS, 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 3). For all three examples, SATABS is eventually able to find the bug, but is three orders of magnitude slower than SCRATCH when applied to *3-buf*. The abstraction-refinement process leads to a conclusive verification result when applied to the correct version of *1-buf*, but is an order of magnitude slower than our k-induction technique. SATABS was not able to prove correctness for correctness of *2-buf* or *3-buf* within 100 refinement iterations.

Comparison with IBM Race Check library The IBM Cell SDK [2] comes with a library for detecting DMA races [12] at runtime. The library maintains a log of pending operations, checking each new operation against entries in the log. 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 three 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. The *Julia* bug, where more than 32 DMA operations may be simultaneously pending, is beyond the scope of the library. 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 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.

7 Related work

The concept of k-induction was first published in [6, 13], targeting the verification of hardware designs represented by transition relations (although the basic idea had already been used in earlier implementations [14] and a version of one-induction used for BDD-based model checking [15]). 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 optimisations and extensions to the technique have been proposed, in-

cluding property strengthening to reduce induction depth [16], improving performance via incremental SAT solving [7], and supporting verification of temporal properties [8]. Applications of k-induction have focused exclusively on hardware designs [6, 13, 14] and synchronous programs [17, 18]. A principle related to k-induction has also been used for circular reasoning about liveness properties [19]. To the best of our knowledge, there has been no previous work on applying k-induction to imperative programs comparable to our procedure in §5.

Techniques for detecting data races in shared memory multithreaded applications have been extensively studied. Notable static methods are based on formal type systems [20], or use classic pointer-analysis techniques; the latter approach is used by tools such as RACERX [21] and CHORD [22]. The ERASER tool [23] uses binary rewriting to monitor shared variables and to find failures of the locking discipline at runtime. Other dynamic techniques include [24], which is based on state-less search with partial-order reduction, and [25] which is based on a partial-order reduction technique for SystemC similar to the method of Flanagan and Godefroid [24].

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 [12], which we compare with in §6. The speed of runtime race detection with this library is attractive, but 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 BMC is able to detect potential races by assuming that input parameters may take *any* value.

8 Summary and Future Work

We have contributed an automatic technique for analysing 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.

We plan to extend this work in the following ways. We intend to generalise 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 analysing DMA races for accelerator memory by analysing 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.* §7).

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

References

1. Hofstee, H.P.: Power efficient processor architecture and the Cell processor. In: HPCA, IEEE Computer Society (2005) 258–262
2. IBM: Cell BE resource center (2009) http://www.ibm.com/developerworks/power/cell/.
3. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers **58** (2003) 118–149
4. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. Volume 2988 of LNCS, Springer (2004) 168–176
5. Kroening, D., Strichman, O.: Efficient computation of recurrence diameters. In: VMCAI. Volume 2575 of LNCS, Springer (2003) 298–309
6. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: FMCAD. Volume 1954 of LNCS, Springer (2000) 108–125
7. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci. **89** (2003)
8. Armoni, R., Fix, L., Fraer, R., Huddleston, S., Piterman, N., Vardi, M.Y.: SAT-based induction for temporal safety properties. Electr. Notes Theor. Comput. Sci. **119** (2005) 3–16
9. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. In: ASE. (2009)
10. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. STTT **9** (2007) 505–525
11. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: TACAS. Volume 3440 of LNCS, Springer (2005) 570–574
12. IBM: Example Library API Reference, version 3.1. (2008)
13. Bjesse, P., Claessen, K.: SAT-based verification without state space traversal. In: FMCAD. Volume 1954 of LNCS, Springer (2000) 372–389
14. Lillieroth, C.J., Singh, S.: Formal verification of FPGA cores. Nord. J. Comput. **6** (1999) 299–319
15. Déharbe, D., Moreira, A.M.: Using induction and BDDs to model check invariants. In: CHARME. Volume 105 of IFIP Conference Proceedings, Chapman & Hall (1997) 203–213
16. Vimjam, V.C., Hsiao, M.S.: Explicit safety property strengthening in SAT-based induction. In: VLSID, IEEE (2007) 63–68
17. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In: FMCAD, IEEE (2008) 109–117
18. Franzén, A.: Using satisfiability modulo theories for inductive verification of Lustre programs. Electr. Notes Theor. Comput. Sci. **144** (2006) 19–33
19. McMillan, K.L.: Circular compositional reasoning about liveness. In: CHARME. Volume 1703 of LNCS, Springer (1999) 342–345
20. Flanagan, C., Freund, S.N.: Type-based race detection for Java. In: PLDI, ACM (2000) 219–232
21. Engler, D., Ashcraft, K.: RacerX: Effective, static detection of race conditions and deadlocks. In: SOSP, ACM (2003) 237–252
22. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: PLDI, ACM (2006) 308–319
23. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. **15** (1997) 391–411
24. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL, ACM (2005) 110–121
25. Helmstetter, C., Maraninchi, F., Maillet-Contoz, L., Moy, M.: Automatic generation of schedulings for improving the test coverage of systems-on-a-chip. In: FMCAD, IEEE (2006) 171–178