Strengthening Induction-Based Race Checking with Lightweight Static Analysis ⋆

Alastair F. Donaldson, Leopold Haller, and Daniel Kroening

Oxford University Computing Laboratory, Oxford, UK

Abstract. Direct Memory Access (DMA) is key to achieving high performance in system-level software for multicore processors such as the Cell Broadband Engine. Incorrectly orchestrated DMAs cause *DMA races*, leading to subtle bugs that are hard to reproduce and fix. In previous work, we have shown that k-induction yields an effective method for proving absence of a restricted class of DMA races. We extend this work to handle a larger class of DMA races. We show that the applicability of k-induction can be significantly improved when combined with three inexpensive static analyses: 1) abstract-interpretation-based static analysis; 2) *chunking*, a domain-specific invariant generation technique; and 3) code transformations based on statement independence. Our techniques are implemented in the SCRATCH tool. We evaluate our work on industrial benchmarks.

1 Introduction

Many modern multicore architectures such as the Cell Broadband Engine (BE) [14] include cores equipped with small, local "scratch-pad" memories, whose management is under software control. It is the programmer's responsibility to orchestrate data movement between main memory and scratch-pad memory using Direct Memory Access (DMA) operations. The use of scratch-pad memory alleviates the *memory wall problem*, where multiple cores contend for access to a single shared memory. Through use of double- and triple-buffering techniques, the latency associated with DMA operations can be hidden, leading to high performance. The increase in performance comes at the expense of programmability: the asynchronous nature of DMA operations makes them difficult to schedule correctly. *DMA races*, where data associated with a pending DMA operation is accessed simultaneously, either by another DMA or a regular memory access, lead to nondeterministic bugs that are hard to diagnose and fix.

In previous work [7] we have shown that SAT-based bounded model checking [2, 4] combined with k-induction [18] shows promise for DMA race analysis. The approach of [7] is restricted to *inter-DMA races* on scratch-pad memory: races between distinct DMA operations. This restriction allows loops that do not involve DMA operations to be removed from an input program via static slicing [19], often leaving just a *single* DMA processing loop to which k-induction can be applied. For increasing values of k, starting, *e.g.*, with $k = 1$, *base case* and *step case* programs are constructed. A successful check of the base case ensures that no DMA races occur during the first k

⋆ This research is supported by the the Toyota Motor Corporation, by the EU FP7 STREP MOGENTES (project ID ICT-216679), and by EPSRC grant EP/G051100.

loop iterations. By checking the step case, it is ensured that, from an arbitrary state, if k loop iterations can be executed without a DMA race occurring, then executing a further iteration will not lead to a DMA race. The base and step case programs are loop-free, and can be checked using bounded model checking. If both succeed, for some k, the input program has been shown to be race-free. If the base case fails, a counterexample leading to a DMA race is reported. Otherwise, a larger value of k must be tried.

In this paper, we remove the restriction to inter-DMA races, extending our analysis to detect races between DMA operations and regular memory accesses. In this setting, it is *not* possible to slice away loops that access buffers with which DMA operations are associated. As a result, k-induction alone is usually too weak to enable verification.

We show that this problem can be solved when k-induction is combined with static analysis. Lightweight static analysis is used to compute an inductive invariant of the input program. Then, k-induction is applied, but the step case is restricted to consider only states which satisfy the inductive invariant. By tailoring the static analysis to the domain of DMA race analysis, we are able to compute sufficiently strong inductive invariants to allow verification of DMA race freedom, on practical examples, for small values of k. We use three distinct static analyses: 1) abstract interpretation, to generate simple strengthening invariants; 2) *chunking*, a domain-specific invariant generation technique; and 3) *code motion*, a code transformation based on statement independence, to compute DMA-related invariants for inner loops.

Our techniques have been implemented in the SCRATCH tool for DMA race analysis [7], and we present an experimental evaluation using a set of benchmarks from the IBM Cell SDK.

2 Guiding k-induction with invariants: an overview

We begin with a high-level overview of our method for strengthening k-induction using static analysis, at the level of transition systems.

Consider a transition system $M = (S, T, I, E)$ where S is a set of states, $T \subseteq S \times S$ a transition relation, $I \subseteq S$ a set of initial states, and $E \subseteq S$ a set of error states. The successor function $post : 2^S \rightarrow 2^S$ is defined by: $post(Q) = \{\, q' \mid \exists q \in Q.\, T(q, q') \}$. We define reachability in the usual way. A set of states $Q \subseteq S$ is *safe*, written $safe(Q)$, if no state of E is reachable from any state in Q. The system M is safe if its initial states I are safe.

To prove that M is safe, it suffices to find a set Q which (i) contains the initial states ($I \subseteq Q$), (ii) is inductive ($post(Q) \subseteq Q$), and (iii) does not contain error states ($Q \cap E = \emptyset$). This can be stated as a proof rule as follows:

$$\frac{\text{(i) } I \subseteq Q \qquad \text{(ii) } post(Q) \subseteq Q \qquad \text{(iii) } Q \cap E = \emptyset}{safe(I)}$$

To implement this proof-rule as a procedure, a means for generating candidates for Q is required. One possibility is the use of some sound static analysis. Such a technique produces an invariant Q that fulfills (i) and (ii), but the results might be too weak to guarantee (iii).

The k-induction technique [18] provides another way to inductively prove correctness of transition systems by splitting the proof obligation into (a) a *base case*, showing that the transition system will not reach an error in k steps, and (b) a *step case*, showing that when no error can be reached in k steps from some arbitrary state s, no error can be reached from s in $k+1$ steps. Writing $safe^k(Q)$ to denote that no error state is reachable from a set Q in at most k steps, the k-induction proof rule for transition systems is as follows:

$$\frac{k \geq 0 \quad \text{(a) } safe^k(I) \quad \text{(b) } \forall Q.\ safe^k(Q) \Rightarrow safe^{k+1}(Q)}{safe(I)}$$

For finite-state transition systems, the premises can be checked using a SAT solver, and the method can be made complete by adding additional restrictions, detailed in [18]. In order to succeed, k-induction can require large values of k, or in the case of infinite state transition systems, may fail for every value of k. When k-induction fails to succeed for feasible values of k, the problem is often that the induction hypothesis, *i.e.*, the antecedent of premise (b), is too weak.

In order to amend this, the above proof rules can be combined into the approach we use in this paper:

$$\frac{k \geq 0 \quad \text{(i) } I \subseteq Q \quad \text{(ii) } post(Q) \subseteq Q \quad \text{(a) } safe^k(I) \quad \text{(b) } safe^k(Q) \Rightarrow safe^{k+1}(Q)}{safe(I)}$$

To implement this rule as a procedure, we first use static analysis to obtain an inductive invariant Q fulfilling premises (i) and (ii). We do not require this invariant to be strong enough to prove the target property. We then perform a k-induction check. Base case (a) is as before, but in step case (b), we strengthen the induction hypothesis by our generated invariant.

3 DMA race analysis

We consider DMA primitives **get**, **put** and **wait**. Operation **get**(l,h,s,t), where l is a pointer to local memory, h a pointer to host memory, and s and t unsigned integers, issues a request for s bytes to be copied from host memory region $[h, h+s)$ to local memory region $[l, l+s)$. The operation is identified by a *tag*, t. The **put** primitive is dual to **get**. Execution of a **get**/**put** operation is asynchronous: the thread issuing the operation may continue to execute while the memory transfer takes place. However, access to the regions $[l, l+s)$ and $[h, h+s)$ before completion of the operation leads to undefined behaviour.[1] Before accessing these regions, the operation must be synchronized via a **wait** operation of the form **wait**(t), where t is the tag used to identify the DMA. This causes execution to block until *all* DMAs identified by tag t have completed.

In the example C program of Figure 1, adapted from an example provided with the IBM Cell SDK [14], function double_buffer will be executed by a Synergistic Processor

[1] Actually, read access to $[h, h+s)$ or $[l, l+s)$ is allowed in the case of **get** or **put** respectively.

```
1  #define SZ 4096
2  float bufs[2][SZ];
3
4  void process(float* b) {
5    for(unsigned int j = 0; j < SZ; j++) { b[j] = b[j] + 1.0f; }
6  }
7
8  void double_buffer(float* in_buf, float* out_buf, unsigned int num_chunks) {
9    unsigned int cur_buf = 0, next_buf, tags[2] = { 0, 1 };
10   get(bufs[cur_buf], in_buf, SZ*sizeof(float), tags[cur_buf]); in_buf += SZ;
11   for (unsigned int i = 1; i < num_chunks; i++) {
12     next_buf = cur_buf^1;
13     wait(tags[next_buf]);
14     get(bufs[next_buf], in_buf, SZ*sizeof(float), tags[next_buf]); in_buf += SZ;
15     wait(tags[cur_buf]);
16     process(bufs[cur_buf]);
17     put(bufs[cur_buf], out_buf, SZ*sizeof(float), tags[cur_buf]); out_buf += SZ;
18     cur_buf = next_buf;
19   }
20   wait(tags[cur_buf]);
21   process(bufs[cur_buf], SZ);
22   put(bufs[cur_buf], out_buf, SZ*sizeof(float), tags[cur_buf]);
23   wait(tags[cur_buf]);
24 }
```

Fig. 1. Using asynchronous DMA to achieve double-buffered data movement

Element core of the Cell processor, which is equipped with scratch-pad memory. Array in_buf is processed by adding 1 to each element, storing the result in out_buf. The arrays in_buf and out_buf reside in host memory. Chunks of data are copied via DMA from in_buf to local store, processed, then copied back to out_buf. *Double buffering* is used to hide the latency associated with DMAs: bufs consists of two local buffers which are used simultaneously for processing and fetching of data. In the main program loop (lines 11–19 of Figure 1), the **get** operation at line 14 requests data which will be processed in the *next* loop iteration. While this request is in progress, processing of the currently available data buffer can be performed. At the start of each loop iteration, the roles of the current and next buffer are exchanged.

A DMA *race* occurs when an attempt is made to access a region of memory that is associated with a pending DMA. Suppose we remove the **wait** at line 15 of Figure 1. Then the **put** at line 17 potentially causes a DMA race: in the first iteration of the loop, this operation accesses memory region [bufs[0] , bufs[0] + SZ∗**sizeof**(**float**)), which is also accessed by the **get** operation at line 10. Without the intervening **wait**, there is no guarantee that the **get** will have completed. This is an example of an *inter-DMA* race. For similar reasons, the array access at line 5, in function process, may cause a DMA race, since it accesses elements of b, which is an alias for bufs[0] when process is

Statement	Instrumented version
start of program	tracker.l = 0; tracker.s = 0; tracker.t = $*$;
put/**get**(l, h, s, t);	**assert**(l != 0 && $s < 16K$ && $t < 32$ && **disjoint**(l, s, tracker.l, tracker.s)); **if**($*$) { tracker.l = l; tracker.s = s; tracker.t = t }
wait(t);	**assume**(tracker.t != t);
$A[e]$ = ... / ... = ...$A[e]$...	**assert**(**disjoint**(&$A[e]$, **sizeof**(T), tracker.l, tracker.s)); $A[e]$ = ... / ... = ...$A[e]$... (where T is the element type of A)

Fig. 2. Translating DMA operations into statements over instrumentation variables

invoked from line 16 during the first loop iteration. This is a race between a DMA and a regular memory access.

Encoding DMA operations. To detect, or prove absence of, local memory DMA races, we instrument a C program with additional assertions and assignments to auxiliary variables. We use the encoding of [6], which improves on that of [7], facilitating analysis of races between DMA operations and local memory accesses.

Our encoding uses an instrumentation record, tracker, which tracks a single DMA. The tracker variable has three fields: l, s and t, recording the local memory address, size and tag of the tracked DMA respectively. Figure 2 shows how program statements relevant to DMA races are translated. At the start of the program, tracker does not track any DMA.

Operation **get**/**put**(l, h, s, t) is translated into an assertion, and a nondeterministically executed assignment. The assertion checks that the pointer parameter l is non-null. The assertion also checks that the memory region accessed by the new DMA is disjoint from the region accessed by the currently tracked DMA. In Figure 2, we use the predicate **disjoint**(l_1, s_1, l_2, s_2) as shorthand for (l_1 + $s_1 \leq l_2$ || l_2 + $s_2 \leq l_2$). The nondeterministically executed assignment either updates tracker to track the new DMA operation or leaves it unchanged. Any statement involving an array access is prepended by an assertion checking that the accessed element is not part of the memory region associated with the tracked DMA. Operation **wait**(t) is translated into an assumption that the currently tracked DMA is not identified by tag t. This rules out the possibility that the last DMA nondeterministically chosen to be tracked also had tag t. The combination of non-deterministic assignments and assumptions ensures that tracker contains details of an *arbitrary* pending DMA. As a result, the assertions associated with DMA operations and array accesses *implicitly* check for races with *all* pending DMAs.

In the rules of Figure 2, and in our experiments (§6), we do not model the effect of a **get** operation on the associated region of local memory. This is sound for programs, including our benchmarks, where data fetched by DMA cannot affect control-flow. Otherwise, analysis may be unsound. To ensure soundness, an implementation could over-approximate the effect of a **get** operation by assigning nondeterministic values to the associated local memory region, which can lead to spurious reports of DMA races during analysis.

```
1   for (i = 1; i < num_chunks; i++) {
2     next_buf = cur_buf^1;
3     assume(tracker.t != tags[next_buf]);
4     assert(dma_cond(bufs[next_buf], SZ*sizeof(float), tags[next_buf]));
5     if(*) { tracker = ( bufs[next_buf], SZ*sizeof(float), tags[next_buf]); }
6     assume(tracker.t != tags[cur_buf]);
7     for(j = 0; j < SZ; j++) {
8  !    assume(0 <= cur_buf && cur_buf <= 1 && 0 <= next_buf && next_buf <= 1 &&
9  !                cur_buf != next_buf && tags[0] == 0 && tags[1] == 1);
10 !    assert(disjoint(&bufs[cur_buf][0], SZ*sizeof}(float}), tracker.l, tracker.s));
11      assert(disjoint(&bufs[cur_buf][j], sizeof(float), tracker.l, tracker.s));
12      bufs[cur_buf][j] = bufs[cur_buf][j] + 1.0f;
13    }
14    assert(dma_cond(bufs[cur_buf], SZ*sizeof(float), tags[cur_buf]));
15    if(*) { tracker = ( bufs[cur_buf], SZ*sizeof(float), tags[cur_buf]); }
16    cur_buf = next_buf;
17  }
```

Fig. 3. Instrumented version of main loop of Figure 1 after inlining. We write **dma_cond**(l, s, t) for the condition of the assertion associated with **put**/**get** operations (*cf.* column 2 of Figure 2), and tracker=(l, s, t) as shorthand for tracker.l=l; tracker.s=s;tracker.t=t. Statements marked '!' are not part of the instrumentation, and will be explained in sections §5.1 and §5.2

Figure 3 shows the main loop of the double buffering example of Figure 1 after instrumentation. The call to process has been inlined, and redundant assignments to in_buf and out_buf removed. The lines marked '!' are *not* generated by instrumentation, and will be explained in §5.1 and §5.2.

4 k-induction for loops: promise, and problems

In [7], we proposed a formulation of k-induction for non-recursive programs with loops, based on the following rule, where correctness denotes partial correctness, characterised by assertions appearing in the program:

$$\frac{\begin{array}{ll}\textbf{Base case} & \textbf{Step case}\\ s_\alpha; & \\ \overbrace{\textbf{if}(\phi)\ \{\ s_\beta\ \}\ \ldots\ \textbf{if}(\phi)\ \{\ s_\beta\ \}}^{k\ \text{times}} & \overbrace{\textbf{assume}(\phi);\ s_\beta^{assume};\ \ldots;\ \textbf{assume}(\phi);\ s_\beta^{assume};}^{k\ \text{times}}\\ \textbf{if}(\neg\phi)\ \{\ s_\gamma\ \}\ \text{is correct} & \textbf{if}(\phi)\ \{\ s_\beta\ \}\ \textbf{else}\ \{\ s_\gamma\ \}\ \text{is correct}\end{array}}{s_\alpha;\ \textbf{while}(\phi)\ \{\ s_\beta\ \};\ s_\gamma\ \text{is correct}}$$

where s_β^{assume} is identical to s_β, except that every statement of the form **assert**(ψ) appearing in s_β is replaced with **assume**(ψ) in s_β^{assume}.

Let P be the program in the conclusion of the rule. The base case checks that there are no erroneous traces from the start of P that execute up to k loop iterations. The step

case checks that any trace beginning in an arbitrary state and executing k loop iterations successfully can be extended to either safely execute a further loop iteration (if ϕ holds), or safely execute the tail of P (if $\neg\phi$ holds). Combining these two facts ensures that P admits no erroneous traces. Soundness of the rule is proved formally in [7]. The base case and step case correspond to the antecedents (a) and (b) of the rules for transition systems presented in section §2.

If the statements s_α, s_β and s_γ of P are loop-free, the base and step cases generated by the k-induction rule do not contain any loops. As a result, they can be checked exhaustively using bounded model checking. Starting with a small value of k, we attempt to verify P by checking a base and step case, increasing k until either a base case fails, revealing a bug in P, both base and step case succeed, indicating that P is correct, or a "give up" value for k is reached, in which case the verification attempt is abandoned.

If s_α, s_β or s_γ are not loop free, *i.e.* the program contains sequences and/or nests of loops, P can be transformed (using standard techniques) into an equivalent program P' containing a *single*, "monolithic" while loop, to which k-induction can be applied.

If we restrict our attention to inter-DMA race checking, as in [7], it is often possible to significantly simplify input programs before verification, through program slicing. For example, the process function of Figure 1 does not involve any DMA operations, and does not influence control-flow in double_buffer. Thus, for the purposes of inter-DMA race checking, we can slice away both calls to process in double_buffer. Applying the translation of §3 to the sliced program yields an instrumented program which can be verified successfully using k-induction: verification succeeds with $k = 2$. Experimental results show that, combined with program slicing, k-induction is effective for analysis of inter-DMA races [7].

Suppose we do not wish to restrict our attention to inter-DMA races when verifying the program of Figure 1, and want to check in addition that the array accesses in process do not race with pending DMAs. In this more complex scenario, k-inductive reasoning is more likely to fail. To illustrate this on our example, consider the inner loop of Figure 3 (excluding the lines marked '!', which should be ignored until §5). For any $k < \mathsf{SZ}$, attempts at proving the example program using k-induction will fail. To see this, consider the following partial state σ, which illustrates three problems that we encounter when attempting to prove our example program correct using k-induction: $\sigma = [\ldots, \mathsf{tracker.l} \mapsto \mathsf{bufs[0]}, \mathsf{tracker.s} \mapsto 1, \mathsf{cur\_buf} \mapsto 0, \mathsf{j} \mapsto x, \mathsf{tags} \mapsto \{40, 41\}]$ where $1 \leq x \leq \mathsf{SZ}$.

Problem 1. *The step case fails to establish information about the contents of the tags array.* For any $k < \mathsf{SZ}$, the assertion at line 14 immediately following the loop (and corresponding to the **put** operation at line 17 of Figure 1) will fail because tags[cur_buf] is not less than 32 at state σ. By assuming successful execution of k iterations of the loop, for $k < \mathsf{SZ}$, we do not establish that tags[cur_buf] is within the permitted range.

Problem 2. *The step case fails to establish information about pending DMAs (I).* Let $k = \mathsf{SZ} - x$. The step case for k-induction will fail: it is possible to successfully execute k iterations of the inner loop from any state of the form σ: the values assigned to tracker ensure that the assertion at line 11 cannot fail for $\mathsf{j} > 0$. However, the assertion at line 14 is guaranteed to fail, since tracker.l=bufs[0]=bufs[cur_buf].

To illustrate a further problem, we consider another partial state, σ':

$\sigma' = [\ldots, \mathsf{tracker.l} \mapsto \mathsf{bufs[1]}, \mathsf{tracker.s} \mapsto 1, \mathsf{tracker.t} \mapsto 2, \mathsf{cur\_buf} \mapsto 0,$
$\mathsf{next\_buf} \mapsto 1, \mathsf{j} \mapsto x, \mathsf{tags} \mapsto \{0, 1\}, \mathsf{i} \mapsto 1, \mathsf{num\_chunks} \mapsto 1000]$

where $0 \leq x \leq \mathsf{SZ}$.

Problem 3. *The step case fails to establish information about pending DMAs (II).* Let $k = \mathsf{SZ} - x + 1$. The k-inductive step case assumes k successful loop iterations. From state σ' it is possible to successfully execute $\mathsf{SZ} - x$ iterations of the inner loop: the assertion at line 11 repeatedly succeeds because tracker.l targets bufs[1] which is disjoint from bufs[cur_buf] = bufs[0]. Statements 14–16 and 2–6 of the outer loop can also be executed successfully; with the monolithic approach to k-induction for multiple loops, this counts as one further loop iteration. Control is now at the start of the inner loop, and we have tracker.l = bufs[1], size.s = 1, and cur_buf = 1. As a result, the assertion at line 11 is guaranteed to fail.

The crucial point common to Problems 1–3 is that the inner loop has too many iterations to be fully unwound, and does not contain sufficiently useful assertions to prove the assertion that follows (in the case of Problems 1 and 2), or assertions in the next outer loop iteration (in the case of Problem 3). In the remainder of the paper we show that we can overcome this problem, and achieve full DMA race analysis, by strengthening k-induction using static analysis techniques, as outlined in §2.

5 Strengthening k-induction for DMA race analysis

To overcome the problems discussed in §4, we use a combination of three lightweight static analysis techniques: a simple abstract interpreter, which allows us to infer generic program invariants, and two techniques for inferring program properties specifically related towards DMA races. While none of these techniques can be used in isolation to prove DMA race freedom, their combination allows us to strengthen the inductive hypothesis sufficiently for successful verification using k-induction.

5.1 Abstract interpretation-based static analysis

In our experience, straightforward analysis with general-purpose abstract domains fails to establish program invariants strong enough to establish DMA race freedom of realistic examples. The properties we are interested in depend on precise interrelations between variable values over multiple, possibly nested loop iterations. General-purpose numeric abstract domains can typically not accurately handle such situations.

Instead of designing an abstract domain specifically for DMA race analysis, with potentially little reuse value, we use lightweight, general-purpose abstract domains together with k-induction. From the analysis result of an abstract interpreter, we extract a mapping inv from statements to local program invariants. We then construct a program P' which is identical to P, except that every statement s appearing in P is prepended with **assume**$(inv(s))$. If the computed invariants are strong enough, it may now be possible to prove P' correct using k-induction.

Consider the code of Figure 3. As discussed in §4, this program fragment, without the lines marked '!', cannot be verified using k-induction due to the lack of useful invariants for the inner loop. The condition of the **assume** statement marked '!' (lines 8–9) is

the relevant part of the invariant computed by our abstract interpreter. Merely assuming this invariant does not allow k-induction to succeed, but *does* rule out many of the conditions under which the step case for k-induction fails. For example, the inferred loop invariant establishes that tags[cur_buf] is in the range $\{0, 1\}$, which eliminates Problem 1 discussed in §4.

We now discuss details of our abstract interpreter.

Analysis with interval and (dis)equality domains. We have found invariants generated by the classical *interval domain* to be useful for our purposes. In addition, we have implemented a simple domain that keeps track of *equalities and disequalities* between variables. We use a reduced product [5] of these two domains, which yields an analysis that is more precise than applying the components in isolation.

Trace partitioning. We have found that, for some practical examples, it is necessary to increase the precision of generated invariants. In each of these cases, we are able to sufficiently increase the precision of the invariant by manually applying *trace partitioning* [16], a refinement technique for abstract domains that enables inference of disjunctive invariants using non-disjunctive domains. We performed *value-based trace partitioning*. This involves identifying a critical section of code α, a variable var and a set of distinct values $\{v_1, v2, \ldots, v_k\}$, and applying the following code transformation:

$$\alpha \longrightarrow \textbf{if}(\text{var} == v_1)\ \{\ \alpha\ \}\ \textbf{else if}(\text{var} == v_2)\ \{\ \alpha\ \} \ldots \textbf{else if}(\text{var} == v_k)\ \{\ \alpha\ \}\ \textbf{else}\ \{\ \alpha\ \}$$

Given a partitioning heuristic, trace partitioning can be automated. For our purposes, a simple heuristic that splits on the value of a variable, whenever this variable is in a very small range, would suffice.

5.2 Chunking

The invariants inferred by our abstract interpreter do not solve Problem 2 of §4: unless we choose $k \geq$ SZ, assuming k successful iterations of the inner loop does not rule out the possibility that a pending DMA is accessing bufs[cur_buf].

Iteration j of the inner loop accesses array element bufs[cur_buf][j] (line 12 of Figure 3), and DMA race checking instrumentation generates an assertion that no DMA is targeting this location (line 11). We describe how, from the assertion **assert**(ϕ) at line 11 of Figure 3, where:

$\phi =$ **disjoint**(&bufs[cur_buf][j], **sizeof**(**float**), tracker.l, tracker.s)

we can derive the assertion **assert**(ψ) marked '!' at line 10, where:

$\psi =$ **disjoint**(&bufs[cur_buf][0], SZ$*$**sizeof**(**float**), tracker.l, tracker.s)

A standard loop analysis (*cf.* [1]) establishes that j is an induction variable (a variable whose value increases linearly with the number of loop iterations) for the inner loop, j is used to access contiguous elements of bufs[cur_buf], and j ranges between 0 and constant value SZ, which are both loop-invariant expressions. It is clear that if a pending DMA operation can write to some portion of array bufs[cur_buf] starting at index a, where $0 \leq a <$ SZ, then the assertion at line 11 will fail when j $= a$. Because j is an induction variable and the loop has SZ iterations, we can deduce that either ψ is a loop invariant, or ϕ must be false in some loop iteration. Thus condition ψ is a *required* invariant for

the loop, and can be added as an assertion: a program trace violating ψ on entry to the loop can be extended by at most SZ iterations to a trace violating ϕ.

We term this process of deriving assertions stating required loop invariants as *chunking*, since it involves gluing together many assertions about small, contiguous regions of memory to form a single summarizing assertion about a larger chunk of memory. We have implemented a chunking analysis for regularly structured loops such as the loop in our example. Such regularly structured loops are a common feature in streaming applications for architectures such as the Cell BE.

5.3 Code motion to ease analysis of inner loops

```
1    ...; assert(dma_cond(bufs[next_buf], SZ*sizeof(float), tags[next_buf]));
2    assume(tracker.t != tags[cur_buf]);
3    for(j = 0; j < SZ; j++) {
4  !   assume(dma_cond(bufs[next_buf], SZ*sizeof(float), tags[next_buf]));
5      assume(0 <= cur_buf && cur_buf <= 1 && 0 <= next_buf && next_buf <= 1 &&
6             cur_buf != next_buf && tags[0] == 0 && tags[1] == 1);
7      assert(disjoint(&bufs[cur_buf][0], SZ*sizeof(float), tracker.l, tracker.s));
8      assert(disjoint(&bufs[cur_buf][j], sizeof(float), tracker.l, tracker.s));
9      bufs[cur_buf][j] = bufs[cur_buf][j] + 1.0f;
10   }
11 ! if(*) { tracker = bufs[next_buf], SZ*sizeof(float), tags[next_buf]); }
12   assert(dma_cond(bufs[cur_buf], SZ*sizeof(float), tags[cur_buf])); ...
```

Fig. 4. Part of the fragment of Figure 3 after application of code motion, indicated by '!'

Recall Problem 3 of §4. We illustrated the issue that, for certain potential states such as σ' on page 7, assuming successful iterations of the inner loop in Figure 3 does not eliminate failing assertions then the inner loop is entered on the next iteration of the outer loop. Unfortunately, neither abstract interpretation nor chunking alleviate this problem.

We overcome this issue through a sound program transformation which we term *code motion* (from *loop-invariant code motion* in compilers [1]). We first illustrate how code motion works using the running example.

Recall from Figure 2 that the instrumentation corresponding to a DMA operation consists of two parts: an assertion **assert**(ϕ) over the tracker record, and a nondeterministic conditional statement **if**($*$) { s }, where s is an assignment to tracker. For the **get** operation at line 14 of Figure 1, these appear at lines 4 and 5 of Figure 3, respectively.

For each instrumented DMA, code motion aims to push **if**($*$) { s } as late in the execution schedule as possible, within the confines of the lexical scope in which the statement appears. After this transformation, statements of the form **assume**(ϕ) are inserted into the headers of all loops lying between the **assert**(ϕ) and **if**($*$) { s }.

Returning to our running example, suppose abstract interpretation and chunking have already been applied to obtain the highlighted statements of Figure 3. Let s_{assign}

denote the nondeterministically guarded assignment statement of line 5. Observe that s_{assign} is independent from the **assume** statement at line 8. Observe further that, if the assignment of s_{assign} *is* executed, then the guards of the **assume** and **assert** statements of lines 6, 10 and 11 are guaranteed to evaluate to $true$. This follows because, from the analysis result of the abstract interpreter, we know that cur_buf $\neq$ next_buf. Finally, note that s_{assign} refers to disjoint variables from the loop control statements j = 0 and j++, and modifies no variable appearing in the loop guard j $<$ SZ.

As a result of these observations, we can move s_{assign} after the inner loop without affecting correctness of the program. In Figure 4, this transformation has been applied: s_{assign} now appears at line 11. To complete code motion for this example, observe that the assertion **assert**(**dma_cond**(...)) at line 1 of Figure 4 is guaranteed to be executed before the inner loop is entered, and we can statically deduce that no statement following this assertion until the end of the inner loop can affect the truth value of **dma_cond**(...). Hence, we can insert at line 4 of Figure 4 the statement **assume**(**dma_cond**(...)) without changing correctness of the program.

Code motion eliminates Problem 3 of §4: the **assume** at line 4 of Figure 4 ensures that the assertions at lines 7 and 8 will not fail when the inner loop is executed in the subsequent outer loop iteration. Code motion is frequently applicable to DMA programs: to overlap communication with computation, a DMA operation is often issued long before its results are needed; there is often an intervening processing loop, as in Figure 3. Code motion reverses this optimization, for the purposes of verification.

Formal details of code motion. To make precise the conditions under which code motion can be applied, we regard a program as a sequence of statements defined by the following grammar:

$$\begin{aligned} \mathit{Stmt} ::= {} & x = \mathit{Expr} \mid \textbf{assert}(\mathit{Expr}) \mid \textbf{assume}(\mathit{Expr}) \mid \\ & \textbf{if}(\mathit{Expr})\ \{\ \mathit{Stmt}\ \}\ \textbf{else}\ \{\ \mathit{Stmt}\ \} \mid \textbf{while}(\mathit{Expr})\ \{\ \mathit{Stmt}\ \} \mid \mathit{Stmt};\ \mathit{Stmt} \end{aligned}$$

where $Expr$ denotes a side-effect free expression over a set of variables, which may involve the nondeterministic expression $*$, and the **else** branch of a conditional statement may be omitted. We assume standard trace semantics for programs.

Definition 1. *Statements s_1 and s_2 are* independent *if one of the following holds:*

1. *No variable assigned to by s_1 is referenced (in an assignment or expression) in s_2, and vice-versa*
2. *s_1 is '**if**$(*)$ { $x = \phi_1$ }', s_2 is '**assert**/**assume**(ϕ_2)', and $\phi_2[x/\phi_1] \equiv true$*
3. *s_2 is '$s; s'$' or '**if**(ϕ) { s } **else** { s' }' or '**while**(ϕ) { s }', s_1 and s are independent, s_1 and s' are independent, and s_1 does not assign to any variable referenced in ϕ*

Essentially, independent statements can be re-ordered without affecting program correctness. In this context, the purpose of condition 1 in Definition 1 is obvious. Condition 2 is domain specific, allowing the nondeterministically executed assignment associated with a DMA operation to be moved. Condition 3 allows independence of compound statements to be established via independence of their components.

A statement s of P is *simple* if s is not a loop, assertion or assumption, and contains no loops, assertions or assumptions as sub-statements.

Benchmark	Lines of code	Time	of which AI	k	Max base case	Max step case
single buffer	152	1.70	9.86%	2	5873	178305
single buffer IO	160	4.25	5.21%	3	6781	334915
double buffer	270	8.52	9.06%	2	67418	386705
double buffer IO	284	24.74	3.49%	3	132266	726512
triple buffer	379	44.32	6.46%	3	9208	650404
triple buffer IO	420	54.80	3.96%	3	9224	707592
double buffer TP	359	9.13	15.65%	2	109783	206434
double buffer IO TP	390	42.47	7.18%	3	215385	854164
triple buffer TP	611	138.10	7.13%	3	8813	958183
triple buffer IO TP	1813	422.45	3.39%	3	8824	3377134

Fig. 5. Experimental results for verification of benchmarks from the IBM Cell SDK.

Lemma 1. *Let '$s_1; s_2$' be a statement sequence appearing in program P. Suppose s_1, s_2 are independent, and s_1 is simple. Let P' be identical to P, except that '$s_1; s_2$' in P is replaced with '$s_2; s_1$' in P'. Then P is correct if and only if P' is correct.*

Lemma 2. *Let $s =$ '$s_1; s_2; \ldots; s_n$' be a statement of program P, where $s_1 =$ '**assert**(ϕ)' and s_1, s_i are independent ($2 \leq i \leq n$). Let $s' =$ '$s_1, s'_2; \ldots; s'_n$', where for $2 \leq i \leq n$, s'_i is identical to s_i, except that any statement t appearing in s_i may be replaced with '**assume**(ϕ); t' in s'_i. Let P' be identical to P, except that statement s is replaced with s'. Then P is correct if and only if P' is correct.*

Lemma 1 (suitably extended to support record variables) allows the nondeterministically executed assignment associated with a DMA to be scheduled later in program execution. Lemma 2 establishes correctness of the strategy described above for placing **assume** statements of the form **assume**(ϕ) at program points dominated by **assert**(ϕ), where ϕ is the race checking condition associated with a DMA.

We omit the straightforward proofs of Lemmas 1 and 2, noting only that the loop-, assertion- and assumption-free conditions of Lemma 1 are necessary to ensure that code motion does not lose error traces, or introduce spurious error traces.

6 Experimental evaluation

We have incorporated analyses based on abstract interpretation, chunking and code motion, into SCRATCH, a tool for DMA race analysis in Cell BE programs, first presented in [7]. SCRATCH performs instrumentation of DMA operations using the encoding of §3. Abstract interpretation and chunking are then applied, after which code motion is attempted for each DMA operation in an input program. In the context of C, independence of statements is determined using a SAT solver, incorporating information obtained via abstract interpretation and pointer alias analysis. SCRATCH attempts to verify the resulting program via k-induction, using the CBMC tool [4] for bounded model checking, equipped with MiniSat 2.0 as a back-end SAT solver.

Our focus is on using k-induction and static analysis to prove DMA race freedom of programs that are suspected to be correct. Bounded model checking, or runtime race analysis (e.g. with the IBM Race Check library [13]) are more effective for detecting DMA races in buggy programs. A comparison of SCRATCH with the IBM Race Check library is presented in [7].

Figure 5 shows experimental results applying SCRATCH to a set of benchmark programs provided with the IBM Cell SDK [14], on a 3.2GHz Intel Xeon machine with 48GB RAM, running Ubuntu. The benchmarks consist of six distinct data programs, using single, double or triple buffering for data-movement. For each buffering strategy there are two program variants, one where processing is performed in-place on a local buffer, and one where processing copies results from an input buffer to an output buffer. The latter variant are marked *IO* in Figure 5. In addition, for the double and triple buffering examples, we present semantically equivalent variants where trace partitioning has been manually applied as discussed in section §5.1. These benchmarks are marked *TP* in Figure 5. For the double and triple buffering examples without trace partitioning, a simple strengthening invariant was added manually.

Simplified versions of these benchmarks, where inner loops are manually sliced away, were studied in [7]. The techniques of [7], where k-induction is applied without strengthening, are unable to handle the full versions of the benchmarks which we consider here. Thus an experimental comparison with [7] is not meaningful.

For each benchmark, Figure 5 shows the number of lines of code, after DMA instrumentation and inlining, the total time (in seconds, averaged over multiple runs) taken for k-inductive verification with SCRATCH, the percentage of this time dedicated to abstract interpretation, and the value of k required for verification to succeed. The cost of the chunking and code motion analyses was negligible in all cases. In all cases, verification is performed with an initial k of 1, and increasing k by 1 until both the k-induction base and step case succeed. Our abstract domain is a prototype, and could be made significantly more efficient with a dedicated implementation. Verification of *triple buffer IO TP* takes significantly longer than for *triple buffer in out*. This is due to code blow-up: value-based trace-partitioning leads to nine variations of a function call, repeated at three distinct points in the program.

We also show the number of variables for the largest SAT problems which had to be solved when checking base and step cases. The step case SAT problems are significantly larger than their associated base cases. The reason for this is that all benchmarks have a similar form to Figure 1, consisting of a main outer loop, and a series of calls to a process function containing a loop with SZ iterations, where SZ = 4096. Because of the loop in process, the base case *never* reaches the end of the first iteration of the main loop. As a result, the vast majority of verification work is undertaken in the step case.

The single buffer benchmarks do not require use of code motion. Verification of the double and triple buffering benchmarks requires all three of our static analysis techniques, and even then requires a non-trivial value of k to succeed. This shows that the mixture of techniques discussed in the paper really are working in tandem.

For the *single buffer* and *single buffer IO* benchmarks, verification by straightforward k-induction is possible if inner loops with SZ iterations are unrolled. This is infeasible when SZ is set to 4096. Therefore, we compare our strengthened k-induction

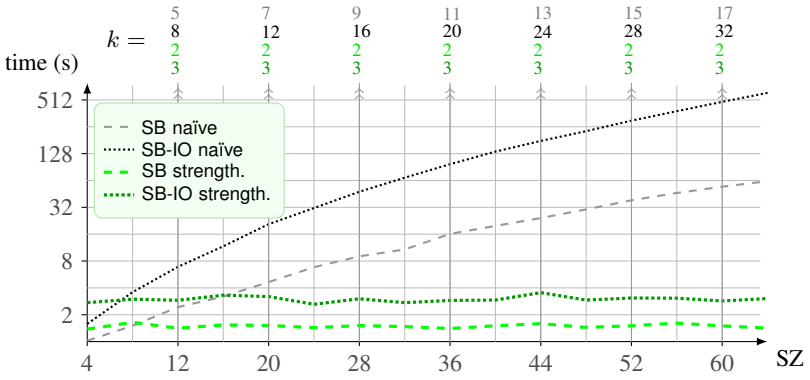

Fig. 6. Verification times for single buffer examples, with and without strengthening

procedure with the naïve approach by replacing SZ with small values. Results are shown in Figure 6, which also indicates the value of k required for verification in each case. The figure illustrates the benefits of strengthened induction: for the largest value of SZ considered, 64, the *single buffer IO* benchmark is verified 170 times faster when strengthening is employed.

7 Related work

Inductive reasoning has played a key role in proofs of program correctness for many years [11]. The k-induction method was introduced in [18]. Variations and applications of k-induction have been extensively studied (see *e.g.* [7, 8, 12]).

The problem of strengthening induction is considered in [3]. A counterexample-driven technique is used to iteratively strengthen the property under consideration in an attempt to make it inductive. The work is primarily concerned with standard induction (1-induction), though experimental results for k-induction are also presented. In contrast, our approach to induction strengthening uses separate static analysis techniques—abstract interpretation, chunking and code motion—to process a program *before* inductive reasoning is applied. We believe our method is complementary to that of [3], and it may be beneficial to combine the approaches.

Techniques for detecting data races in shared memory multithreaded applications have been extensively studied, see *e.g.* [10, 9, 15, 17]. However, none of these methods are applicable to DMA race detection for multicore architectures such as the Cell BE.

8 Summary

The k-induction method is a promising technique for SAT-based verification of programs with loops, and has shown merit in this area in the context of DMA race analysis. The main weakness of k-induction is its sensitivity to inner loops, which may contain

insufficiently strong assertions to build up an inductive invariant that implies program correctness. We have addressed this weakness by using three kinds of static analysis: abstract interpretation, chunking, and code motion. We have incorporated these techniques into SCRATCH, a DMA race analysis tool for the Cell BE architecture, and shown experimentally that the combination of static analysis and k-induction takes a large step towards the goal of fully automated verification of programs in this application domain.

We believe there is potential for generalizing our chunking and code motion analyses to aid inductive verification of programs in other application domains. We also intend to achieve 100% automation through an implementation of trace partitioning. A key challenge will be to design heuristics to automatically decide where trace partitioning can be usefully applied.

References

1. Aho, A., Lam, M., Sethi, R., Ullman, J.: Compilers: Principles, Techniques and Tools. Addison-Wesley (2006)
2. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 118–149 (2003)
3. Bradley, A.R., Manna, Z.: Property-directed incremental invariant generation. Formal Asp. Comput. 20(4-5), 379–405 (2008)
4. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. LNCS, vol. 2988, pp. 168–176. Springer (2004)
5. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL. pp. 269–282 (1979)
6. Donaldson, A.F., Kroening, D., Rümmer, P.: Scratch: a tool for automatic analysis of DMA races. In: PPoPP (to appear) (2011)
7. Donaldson, A.F., Kroening, D., Rümmer, P.: Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In: TACAS. pp. 280–295. Springer (2010)
8. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci. 89(4) (2003)
9. Engler, D., Ashcraft, K.: RacerX: Effective, static detection of race conditions and deadlocks. In: SOSP. pp. 237–252. ACM (2003)
10. Flanagan, C., Freund, S.N.: Type-based race detection for Java. In: PLDI. pp. 219–232. ACM (2000)
11. Floyd, R.W.: Assigning meanings to programs. Mathematical aspects of computer science, Proceedings of Symposia in Applied Mathematics pp. 19–32 (1967)
12. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In: FMCAD. pp. 109–117. IEEE (2008)
13. IBM: Example Library API Reference, version 3.1 (July 2008)
14. IBM: Cell BE resource center (2009), http://www.ibm.com/developerworks/power/cell/
15. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: PLDI. pp. 308–319. ACM (2006)
16. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Trans. Program. Lang. Syst. 29(5) (2007)
17. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. 15(4), 391–411 (1997)
18. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: FMCAD. LNCS, vol. 1954, pp. 108–125. Springer (2000)
19. Tip, F.: A survey of program slicing techniques. J. Prog. Lang. 3(3) (1995)