Safe Asynchronous Multicore Memory Operations

Matko Botinčan, Mike Dodds
University of Cambridge
{matko.botincan, mike.dodds}@cl.cam.ac.uk

Alastair F. Donaldson
Imperial College London
afd@doc.ic.ac.uk

Matthew J. Parkinson
Microsoft Research Cambridge
mattpark@microsoft.com

***Abstract*—Asynchronous memory operations provide a means for coping with the memory wall problem in multicore processors, and are available in many platforms and languages, e.g., the Cell Broadband Engine, CUDA and OpenCL. Reasoning about the correct usage of such operations involves complex analysis of memory accesses to check for races. We present a method and tool for proving memory-safety and race-freedom of multicore programs that use asynchronous memory operations. Our approach uses separation logic with permissions, and our tool automates this method, targeting a C-like core language. We describe our solutions to several challenges that arose in the course of this research. These include: syntactic reasoning about permissions and arrays, integration of numerical abstract domains, and utilization of an SMT solver. We demonstrate the feasibility of our approach experimentally by checking absence of DMA races on a set of programs drawn from the IBM Cell SDK.**

***Index Terms*—Software verification; Concurrent programs; Abstract interpretation; Automated theorem proving**

I. Introduction

Asynchronous memory operations are an important feature of modern multicore systems. They provide a means for coping with the high cost of shared-memory access (the so-called *memory wall* problem). Using asynchronous operations, cores can delegate data-movement to dedicated hardware, and continue processing on private memory, to which they have fast, contention-free access. These operations are widely available, e.g. direct memory access (DMA) operations in the Cell Broadband Engine, asynchronous copying in OpenCL, and one-sided operations in MPI-2.

The high performance permitted by asynchronous memory operations comes at a price: increased programming complexity. Erroneous synchronisation within a thread can lead to data races, for example when copying a section of memory by an asynchronous operation and then writing to or deallocating it before the operation completes. The deallocation of the memory can happen implicitly when a function returns, or when a thread is joined. These problems are compounded in a multithreaded setting: incorrectly managed asynchronous operations can lead to *inter-core* data races, where a thread running on core i issues an operation to copy data to the local memory of core j, while a thread running on core j simultaneously accesses this memory.

Asynchronous operations run concurrently with other threads, and have undefined behaviour over written memory until synchronised. Consequently, incorrectly-managed asynchronous copying can lead to highly nondeterministic bugs that are extremely difficult to diagnose through simulation and testing. While data races in shared-variable concurrent programs are often benign, or intentional, this is virtually never the case for races involving asynchronous operations, and we regard such races as bugs. Since buggy programs may behave entirely correctly on some implementations, while failing drastically on others, there is an urgent need for formal verification techniques in this area.

In this paper, we present a method and a tool for proving safety of multicore programs with fork/join thread-spawning and asynchronous memory operations. Our deductive, proof-based method uses the following intuition: (1) to successfully perform an asynchronous memory operation to move data from A to B, a thread must have permission to read from A, and permission to write to B; and (2) upon issuing such an operation, the thread loses these permissions until it issues a corresponding synchronisation operation that waits until the transfer has completed, at which point permissions for A and B are restored. We present program logic rules for asynchronous memory operations that formalise these intuitions. If it is possible to derive a proof for a multicore program using our proof system, then the program is guaranteed to be memory-safe and race-free.

We have developed techniques for automating our verification method, which involves complex flow-, path- and context-sensitive analysis. Our approach hinges on symbolic execution over an abstract domain of separation logic assertions [3] with permissions [4] and array predicates. Using the rules of our proof system for entailment checking and abstraction, symbolic execution is guided towards a fix-point, yielding a safety proof if the program is correct. If the program is buggy, the proof attempt will fail, providing feedback which can be used to locate the source of the bug.

We have implemented our method for a C-like language in a prototype tool, asyncStar, built on coreStar [5], a modular back-end for separation logic analysis and verification. Standard separation logic abstraction techniques [10] are used to automatically infer loop invariants describing heap resources. To handle numeric constraints relevant to array manipulation, we extend coreStar with a novel technique allowing integration of arbitrary numerical domains [9] with separation logic abstraction.

We present an experimental evaluation demonstrating the feasibility of the abstraction approach on a number of hand-crafted examples, and of the whole method using a set of benchmark programs drawn from the IBM Cell Broadband Engine SDK [21].

In summary, the contributions of the paper are:

- A program logic for multicore programs with threads and asynchronous memory operations.
- Solutions to a number of challenges arising when automating the reasoning with such a logic, including: syntactic reasoning about permissions and arrays, integration of numerical abstract domains, and utilization of an SMT solver.
- asyncStar, a prototype implementation of our method, evaluated on a set of industrial benchmarks.

Asynchronous memory operations are increasingly widely used, for example in languages such as CUDA and OpenCL. While we have focussed in this paper on the Cell BE architecture, the techniques we propose could form the basis of a correctness analysis for any similar domain.

II. Background

We now introduce asynchronous memory operations via a running example, and recap the definition of a data race in a concurrent program. We then motivate the need for techniques capable of analysing data races caused by asynchronous memory operations. We use our running example to demonstrate the subtle nature of bugs that can arise due to this type of race, and argue that data races caused by asynchronous operations are almost never benign.

Asynchronous memory operations. Our target memory architecture closely resembles the IBM Cell BE. We assume a single central CPU core responsible for coordinating and distributing tasks (the *master*, or *host*), and a number of subsidiary cores that can be used to perform asynchronous computations (*slave* or *accelerator* cores). Each core has its own local memory, disjoint from other cores. The host core has regular access to the system's main memory, which we call *host memory*, while each accelerator core has its own *scratchpad memory*, also referred to as *local memory*.

An accelerator core can access its local memory in a fast, contention-free manner using standard load and store instructions. The local memory of one core is not directly visible to any other cores, including the host. Accelerator cores cannot access host memory using standard load and store instructions. Instead, the accelerator cores access host memory using special **get** and **put** operations, which copy a segment of memory from and to from the core's local memory, respectively. **get** and **put** are *asynchronous* operations; they each *initiate* a memory transfer, but they do not wait for the transfer to complete. In order to be sure that a transfer has finished, a thread must call **wait**. Each **get** and **put** is associated with a *tag*. A **wait** call takes a tag as an argument and blocks until all operations associated with that tag have completed.[1] In the Cell architecture, asynchronous memory operations are implemented using direct memory access (DMA).

To see how this works in practice, consider the double-buffering algorithm dub_buf shown in Fig. 1, which is to be executed by an accelerator core and is spawned by a host thread master. The algorithm reads an array of L·M values from host memory, processes the values locally, and copies the result to an output array in host memory. The algorithm reads chunks from the input array using **get**, processes them in local memory (we do not show the actual processing code, assuming that it does not issue any asynchronous memory operations), then copies the result to the output array using **put**. The arguments to **get** and **put** are a local address x, a host address y, the number of bytes s of data to be copied, and a tag t.

Two internal buffers of size L are used to store copied chunks of the shared array. At any point during execution of the while loop, one buffer is being filled with data via a **get** operation, while the other buffer is being processed in-place, or its contents are being transferred back to shared memory via a **put** operation. Because local computations and asynchronous operations execute in parallel, the algorithm achieves efficiency by overlapping computation with communication.

Data races. We follow the definition of a data race given in [18]. A program has a *data race* if the program can be executed on a multiprocessor such that two memory accesses are performed simultaneously, and:

- the accessed regions of memory overlap
- at least one the accesses is a write
- the accesses are not *both* for synchronisation purposes

In general concurrent programs, data races often lead to subtle bugs which occur nondeterministically, depending on particular thread or process interleavings. Data races can often be benign. For instance, it is common for threads to race when updating shared variables used for logging [18]. Distinguishing between dangerous and benign races is a difficult problem. However, we will argue that in an asynchronous setting data-races are almost never benign.

Asynchronous memory operations and data races. In the context of asynchronous memory operations, a **get** operation issued by core i can race with:

- a regular read or write by core i to the associated region of local memory
- a **get** or **put** issued by core i, accessing an overlapping region of local memory
- a regular write access by the host core to the associated region of host memory
- a **put** issued by a core j (possibly equal to i), accessing an overlapping region of host memory

The race scenarios for a **put** operation are analogous.

High latency asynchronous operations are typically used to transfer large chunks of data between memory spaces. Thus, in contrast to data races in shared-variable concurrent programs, races caused by asynchronous memory operations are almost *never* benign. Furthermore, the asynchronous nature of **get** and **put** can lead to particularly subtle bugs. For example, if

[1] In the case of Cell BE, a **wait** operation only blocks until all operations associated with the tag *and* issued by the core that calls **wait** have completed. This can be realised in our setting by equipping each accelerator core with a distinct set of tags.

```
#define N ...  // Num threads.
#define M ...  // Num chunks of data to
               //   be processed per thread.
#define L ...  // Size of chunk, in bytes.

master(char src[len], char dst[len]) {
  // 'master' runs on host.
  tid* t[N]; int i;
  for (i=0; i<N; i++) {
    t[i] = fork(
      dub_buf,
      2*i,
      src+i*M*L,
      dst+i*M*L);
  }
  for (i=0; i<N; i++) {
    join(t[i]);
  }
}
```

```
dub_buf(int t, char *ihead, char *ohead) {
  char buf[2][L];     // Two buffers, each of size 'L' bytes.
  int cur = 0;        // Indices recording which buffers
  int nxt = 1;        //    are being used for input / output.
  int i = 1;
  char *in = ihead;   // Pointers to data in host memory.
  char *out = ohead;
  get(buf[cur], in, L, t); in += L;
  while (i < M) {
    wait(t^nxt);                     // Wait for previous 'put' to complete.
    get(buf[nxt], in, L, t^nxt);     // Get data to process next iteration.
    in += L;
    wait(t^cur);                     // Wait for requested data to arrive.
    // ⟨process data⟩
    put(buf[cur], out, L, t^cur );// Put results back to host memory.
    out += L;
    cur = cur^1;  nxt = nxt^1;       // Switch buffers and
    i += 1;                          //   increase chunk count.
  }
  wait(t^cur); wait(t^nxt)
  // ⟨process data⟩
  put(buf[cur], out, L, t^cur);      // Put results back to host memory.
  wait(cur);
}
```

Fig. 1. Double-buffering algorithm, adapted from the Cell SDK [21].

a **get** operation targeting a region of the stack allocated by a function f is not properly synchronised, the operation may still be pending after f returns. This can lead to corruption of the stack during subsequent calls. Asynchronous operations may complete quickly enough that stack corruption is unlikely, occurring only rarely, e.g. if bus traffic is sufficiently high. This means that bugs can be extremely difficult to reliably reproduce.

Patterns of copying can be complex even for simple examples, making mistakes hard to avoid. Consider again the double-buffering algorithm. The algorithm uses binary variables cur and nxt to record which buffer is incoming and which is outgoing. At each loop iteration the variables swap values. Furthermore, at the start of each loop iteration, data from one local buffer is being copied back to the input buffer, while data in adjacent cells in the input buffer is being copied into the other local buffer. At the end of the loop, one local buffer is the target of a copy from the input buffer, while the other is the source of a copy to the output buffer. The potential for confusion is enormous (the authors of this paper found avoiding bugs rather tricky when adapting the algorithm for presentation here).

To avoid undefined behaviour caused by races with asynchronous memory operations, we must be able to ensure that the source of an asynchronous operation is not written to, and the target neither written to nor read from, until the operation is explicitly synchronised via a **wait** call. For instance, if there was no call to **wait** at the first line in the while-loop of dub_buf then the example would exhibit a race which we can observe by logging first five asynchronous operations:

```
(1)  get(buf[cur], in, L, t)         // cur=0
(2)  get(buf[nxt], in, L, t^nxt)     // i=0, nxt=1
(3)  wait(t^cur)                     // i=0
(4)  put(buf[cur], out, L, t^cur)    // i=0, cur=0
(5)  get(buf[nxt], in, L, t^nxt)     // i=1, cur=0
```

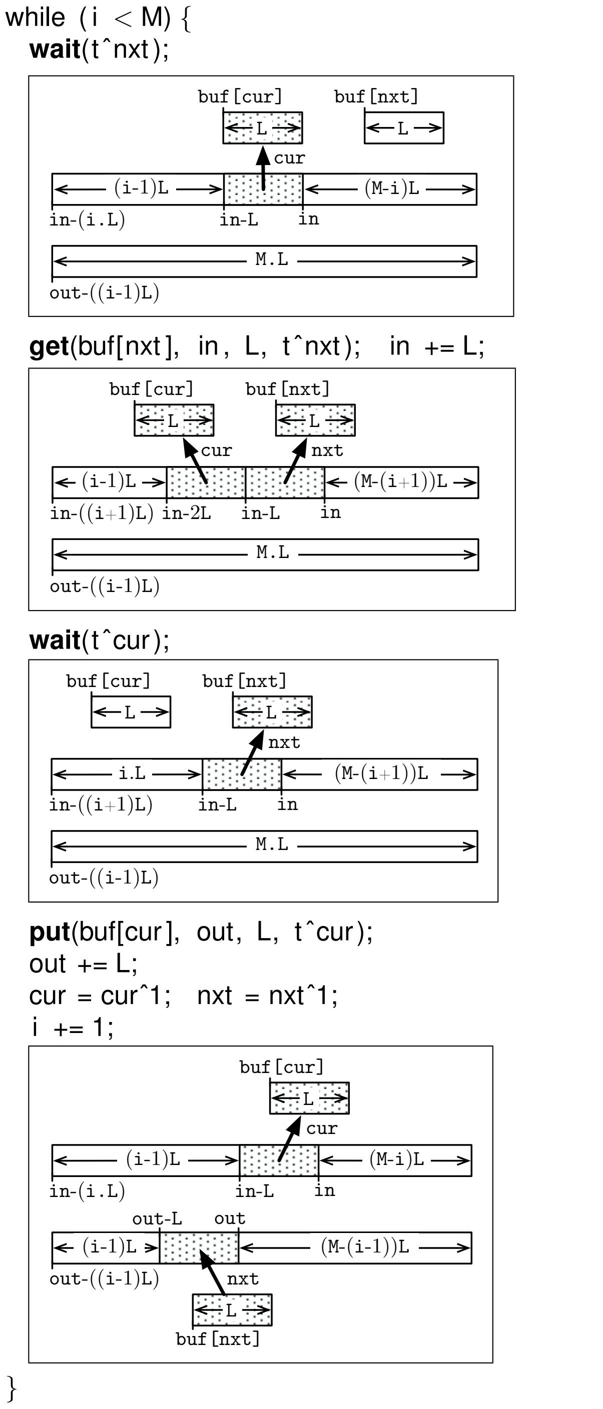
The operation **put** at (4) is in race with the operation **get** at (5) because both of them are accessing the same portion of the internal buffer.

Existing techniques for dynamic race detection could be readily adapted to apply in the context of asynchronous memory operations, and runtime monitoring [20] and system simulation [23] have been used to find races in Cell BE programs. However, such techniques can only *detect* data races. In the remainder of the paper, we describe a method which can be used to *prove absence* of data races due to asynchronous memory operations in concurrent programs.

III. Reasoning about asynchronous operations

We now use the double buffering example of Fig. 1 to provide an overview of our technique for reasoning about race freedom of asynchronous memory operations in concurrent programs. For reasons of space, we do not present full formal details of our approach, and justify its soundness in an informal manner.

By careful analysis of the dub_buf function of Fig. 1, we can determine precisely the pattern of copying between the



```
while ( i < M) {
  wait(t^nxt);
  get(buf[nxt], in, L, t^nxt);  in += L;
  wait(t^cur);
  put(buf[cur], out, L, t^cur);
  out += L;
  cur = cur^1;  nxt = nxt^1;
  i += 1;
}
```

Fig. 2. Sequence of memory reads and writes in the main loop of the double-buffering algorithm.

input and output arrays, and the two buffers local to each thread. This pattern of behaviour is illustrated in Fig. 2. In this diagram, boxes denote contiguous chunks of memory. Labelled arrows inside boxes are used to indicate the size of each chunk, while external labels indicate addresses. Grey-filled boxes denote chunks that are subject to pending memory-transfer operations, while arrows between grey boxes indicate the direction of copying. Labels on these arrows denote the tag of the operation.

Reasoning using separation logic. Diagrams such as Fig. 2 would be laborious and difficult to annotate correctly by hand. However, precisely capturing information about shifting patterns of reading and writing is essential in establishing correctness of an algorithm involving asynchronous memory operations.

We use a program logic, based on *separation logic* [26], to capture precisely the information presented intuitively in Fig. 2. Separation logic is a Hoare-style logic for verifying programs. Reasoning in it depends on a new logical connective – the *separating conjunction*, '$*$'. Formula $P_1 * P_2$ asserts that memory can be split into two disjoint parts, one satisfying P_1 and the other P_2. Reasoning in separation logic is *local*, meaning that a specification must express all the resources a program needs to execute without faulting. For example, the specification

$$\{P\}\,C\,\{Q\}$$

says that, if the program C executes to termination starting with a resource satisfying P, the result will be a resource satisfying Q, **and** that all resources used by C are either specified by P or acquired through explicit resource transfer.

Locality means that a program can be verified within a small resource, and then substituted into a larger context. This property is expressed by the *frame* and *parallel* rules[2]:

$$\textsc{Frame}\ \ \frac{\{P\}\ C\ \{Q\}}{\{P * F\}\ C\ \{Q * F\}}$$

$$\textsc{Parallel}\ \frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}$$

The frame rule allows any specification $\{P\}\,C\,\{Q\}$ to be extended by an arbitrary frame F that is unchanged by the C. The parallel rule allows the specifications for two threads to be combined, provided they access disjoint resources.

Reasoning in separation logic is performed in terms of a single thread at once – the *local thread*. Reasoning is therefore said to be *thread-local*. We can think of the pre- and post-condition as denoting the portion of the state of which the local thread has *ownership*. Other running threads may own other portions of the state, but the parallel rule ensures that the global pattern of ownership is consistent. Note that ownership does not preclude sharing; two threads can share a resource as long as both only read from it.

Separation logic is an appropriate approach to verifying asynchronous memory copying operations because it deals gracefully with ownership and with resource-transfer between threads. Separation logic also has a proven history in verifying complex algorithms with dynamic patterns of ownership, and good tool-support for symbolic execution.

Representing arrays and pending operations. To reason about examples such as the double-buffering algorithm, we

[2] Below, we also give rules for thread manipulation using **fork** and **join**.

must be able to precisely capture patterns of resource ownership and transfer. To do this, we extend separation logic with two new assertions, arr and pend.

- $\mathsf{arr}_c(x, s, p, vs)$ denotes an array of s bytes, starting at address x. Parameter p is a *permission* giving the level of access the local thread holds over the array. The final parameter records that the array holds values vs. Subscript c records where the array is stored: either h for host memory, or ℓ for local scratchpad memory.
- $\mathsf{pend}(t, \mathcal{O})$ denotes a set $\mathcal{O}$ of pending asynchronous memory operations with associated tag t. Each element of $\mathcal{O}$ is a tuple recording a particular operation. As well as recording the existence of a set of pending operations, a thread owning this predicate can access the associated resources after the operations complete.

Using these predicates and the $*$ operator, we can represent complex states in resource-transferring algorithms.

Permissions. Permissions are used in separation logic to support race-free sharing between threads [4]. They can be represented by fractions[3] in the interval $(0, 1]$. A permission 1 denotes that the thread has exclusive write access to the array, while a permission $p \in (0, 1)$ records that it has non-exclusive read access. Soundness is ensured by the guarantee that the sum of all permissions is never more than 1. The permission 1 can be split into read permissions, which may themselves be split further, and split permissions may be joined back together. Permissions in array predicates are split and joined according to the following rule:

$$p \leq q \leq 1 \quad \Rightarrow \quad \begin{array}{l} (\mathsf{arr}_c(x, s, q, vs) \Leftrightarrow \\ \quad \mathsf{arr}_c(x, s, p, vs) * \mathsf{arr}_c(x, s, q - p, vs)) \end{array}$$

In addition to splitting and joining of permissions, we can also split and join array predicates with respect to length:

$$x \leq y < (x + s) \implies \left(\begin{array}{l} \mathsf{arr}_c(x, s, p, vs) \iff \\ \left(\begin{array}{l} \exists vs_1, vs_2.\, vs = vs_1 \mathbin{@} vs_2 \wedge \\ \mathsf{arr}_c(x, y - x, p, vs_1) * \mathsf{arr}_c(y, x + s - y, p, vs_2) \end{array} \right) \end{array} \right)$$

(We use @ to denote concatenation of array values.)

Reasoning about get, put and wait. We reason about **get** and **put** by giving them specifications based on the arr and pend predicates. The specification for **get** is as follows.

$$\begin{array}{l} \{\mathsf{arr}_\ell(x, s, 1, xs) * \mathsf{arr}_h(y, s, p, ys) * \mathsf{pend}(t, \mathcal{O})\} \\ \quad \mathbf{get}(x, y, s, t) \\ \{\mathsf{pend}\left(t, \{\langle y_h, x_\ell, s, p, ys\rangle\} \cup \mathcal{O}\right)\} \end{array}$$

Before calling **get**, the thread must have read access to the host array and write access to the local array. After **get** completes, the thread loses the permissions it held for both arrays – it cannot safely write to x, as it may be in an inconsistent state. It *can* continue to safely read from y, as long as it holds an additional read permission.

$$\begin{array}{l} \left\{\mathsf{arr}_\ell(x, s, 1, xs) * \mathsf{arr}_\ell(z, s, 1, zs) * \mathsf{arr}_h(y, s, \tfrac{1}{2}, ys) * \mathsf{pend}(t, \emptyset)\right\} \\ \quad \mathbf{get}(x, y, s, t); \\ \left\{\mathsf{arr}_\ell(z, s, 1, zs) * \mathsf{arr}_h(y, s, \tfrac{1}{4}, ys) * \mathsf{pend}\left(t, \{\langle y_h, x_\ell, s, \tfrac{1}{4}, ys\rangle\}\right)\right\} \\ \quad \mathbf{get}(z, y, s, t); \\ \left\{\mathsf{pend}\left(t, \{\langle y_h, x_\ell, s, \tfrac{1}{4}, ys\rangle, \langle y_h, z_\ell, s, \tfrac{1}{4}, ys\rangle\}\right)\right\} \end{array}$$

Fig. 3. An example proof outline illustrating splitting of permissions and framing.

Intuitively, the arrays are held by the memory controller until the operation completes. The pend predicate is updated with a tuple recording the new operation (the source address, the target address, length, the permission on the source and the value of the source; the subscripts ℓ and h indicate in which memory spaces the arrays are stored).

The specification for **put** is analogous to that for **get**:

$$\begin{array}{l} \{\mathsf{arr}_\ell(x, s, p, xs) * \mathsf{arr}_h(y, s, 1, ys) * \mathsf{pend}(t, \mathcal{O})\} \\ \quad \mathbf{put}(x, y, s, t) \\ \{\mathsf{pend}\left(t, \{\langle x_\ell, y_h, s, p, xs\rangle\} \cup \mathcal{O}\right)\} \end{array}$$

In Fig. 3 we illustrate how splitting of permissions and framing works when used with specifications of **get** and **put**. Suppose that we want to transfer s bytes from the host array y to local arrays x and z. To do this we need a write permission for arrays x and z and a read permission for array y (which can be any fraction greater than 0 and less than or equal to 1). Assume that we start with the permission $\frac{1}{2}$ given to the array y. To issue the first **get** transferring from y to x we can split the fraction $\frac{1}{2}$ into quarters and then frame with $\mathsf{arr}_\ell(z, s, 1, zs)$ and $\mathsf{arr}_h(y, s, \frac{1}{4}, ys)$ in order to apply the specification. To apply the specification for the second **get** we frame with the predicate $\mathsf{pend}\left(t, \{\langle y_h, x_\ell, s, \frac{1}{4}, ys\rangle\}\right)$.

The **wait** function ensures that all memory operations associated with a particular tag t have completed. All the arrays that were sources or targets of these pending operations can be safely accessed once the memory operations have completed. Consequently, the specification of **wait** returns the arrays held by the memory controller to the thread.

$$\begin{array}{l} \{\mathsf{pend}(t, \mathcal{O})\} \\ \quad \mathbf{wait}(t) \\ \left\{ \begin{array}{l} \mathsf{pend}(t, \emptyset) * \\ \circledast_{\langle x_c, y_{c'}, s, p, vs\rangle \in \mathcal{O}}.\, \mathsf{arr}_c(x, s, p, vs) * \mathsf{arr}_{c'}(y, s, 1, vs) \end{array} \right\} \end{array}$$

In the post-condition of **wait**, the set of pending operations for the tag t is empty, and for each pending operation recorded in the precondition, a pair of arrays has been returned to the thread. Note that for each pair of host and scratchpad arrays, the values in the arrays are identical, as the copying operation has completed. We use $\circledast$, the iterated separating conjunction,[4] to say that the thread holds a pair of arrays for *each* pending operation represented in the precondition.

[3] In order to avoid reasoning about fractional arithmetic, in our tool we represent permissions by trees. See §IV-B for details.

[4] That is, $\circledast_{i \in \{x, y, z..\}}.\, P \;\triangleq\; P[x/i] * P[y/i] * P[z/i] \ldots$

Verifying the double-buffering algorithm. The dub_buf function can be specified as follows, using predicates from our assertion language:

$$\left\{\begin{array}{r}\mathsf{arr}_h(\mathsf{ihead}, \mathsf{M}\cdot\mathsf{L}, p, is) * \mathsf{arr}_h(\mathsf{ohead}, \mathsf{M}\cdot\mathsf{L}, 1, \_) \\ * \,\mathsf{pend}(\mathsf{tag}, \emptyset) * \mathsf{pend}(\mathsf{tag}+1, \emptyset)\end{array}\right\}$$

```
dub_buf(tag, ihead, ohead)
```

$$\left\{\begin{array}{r}\mathsf{arr}_h(\mathsf{ihead}, \mathsf{M}\cdot\mathsf{L}, p, is) * \mathsf{arr}_h(\mathsf{ohead}, \mathsf{M}\cdot\mathsf{L}, 1, \_) \\ * \,\mathsf{pend}(\mathsf{tag}, \emptyset) * \mathsf{pend}(\mathsf{tag}+1, \emptyset)\end{array}\right\}$$

(Here, and elsewhere, we use underscore, '_', to denote a fresh existentially-quantified variable that is used only once.)

The pre- and post-conditions both assert that a pair of arrays exist at addresses ihead and ohead, each of length M·L. The sets of pending operations on tags tag and tag + 1 are empty. The ohead array must have a permission argument 1, meaning the algorithm must be able to write to this array. However, the ihead array can have an arbitrary permission $p \in (0, 1]$ as its argument, meaning that the thread only requires the ability to read. The separating conjunction '$*$' is essential in assigning the right meaning to this specification. Conjoining the array predicates with $*$ ensures that they occupy disjoint portions of memory (meaning that the ohead array can be modified without affecting the ihead array).

The assertions arr and pend can now be used to represent invariants of the double-buffering algorithm. Fig. 4 shows the invariants for the main loop of the double-buffering algorithm. These invariants correspond directly to the intuitive reading of the algorithm given by the diagrams in Fig. 2. Furthermore, these invariants can be generated automatically by our tool.

The following assertion (the first invariant in Fig. 4) corresponds to the first diagram in Fig. 2:

$$\begin{array}{ll|l}
 & \mathsf{buf}[\mathsf{cur}] \mapsto b^c * \mathsf{buf}[\mathsf{nxt}] \mapsto b^n & 1 \\
* & \mathsf{arr}_\ell(b^n, \mathsf{L}, 1, \_) & 2 \\
* & \mathsf{arr}_h(\mathsf{in}-(\mathsf{i}\cdot\mathsf{L}), (\mathsf{i}-1)\cdot\mathsf{L}, p, is_1) & 3 \\
* & \mathsf{arr}_h(\mathsf{in}, (\mathsf{M}-\mathsf{i})\cdot\mathsf{L}, p, is_2) & 4 \\
* & \mathsf{pend}(\mathsf{t\hat{}\,cur}, \{\langle \mathsf{in}_h - \mathsf{L}, b^c_\ell, \mathsf{L}, p, is_p\rangle\}) * \mathsf{pend}(\mathsf{t\hat{}\,nxt}, \emptyset) & 5 \\
* & \mathsf{arr}_h(\mathsf{out}-((\mathsf{i}-1)\cdot\mathsf{L}), \mathsf{M}\cdot\mathsf{L}, 1, \_) & 6 \\
\wedge & (is_1 \mathbin{@} is_p \mathbin{@} is_2) = is & 7
\end{array}$$

Line 1 asserts that buf[cur] and buf[nxt] point to addresses b^c and b^n. Variables cur and nxt are used to choose between elements of an array of two buffers, where each buffer has size L. Line 2 denotes the unused buffer array at b^n. Lines 3, 4 and 6 denote the in and out arrays. The in array has a chunk of size L missing – this is held by the asynchronous copying operation and will be returned when the operation completes. Line 5 asserts that the t^cur tag is being used to copy from the host in array to the local array at b^c, but that the t^nxt tag is unused (it is associated with an empty set of operations). Line 7 asserts that the concatenation of the two input arrays and the values in the pending copy results in the original contents of the input array.

```
while (i < M) {
    wait(t^nxt);
```

$$\left\{\begin{array}{r}
\mathsf{buf}[\mathsf{cur}] \mapsto b^c * \mathsf{buf}[\mathsf{nxt}] \mapsto b^n * \mathsf{arr}_\ell(b^n, \mathsf{L}, 1, \_) \\
* \,\mathsf{arr}_h(\mathsf{in}-(i\cdot\mathsf{L}), (\mathsf{i}-1)\cdot\mathsf{L}, p, is_1) \\
* \,\mathsf{arr}_h(\mathsf{in}, (\mathsf{M}-\mathsf{i})\cdot\mathsf{L}, p, is_2) \\
* \,\mathsf{pend}(\mathsf{t\hat{}cur}, \{\langle \mathsf{in}_h - \mathsf{L}, b^c_\ell, \mathsf{L}, p, is_p\rangle\}) * \mathsf{pend}(\mathsf{t\hat{}nxt}, \emptyset) \\
* \,\mathsf{arr}_h(\mathsf{out}-((i-1)\cdot\mathsf{L}), \mathsf{M}\cdot\mathsf{L}, 1, \_) \\
\wedge (is_1 \mathbin{@} is_p \mathbin{@} is_2) = is
\end{array}\right\}$$

```
    get(buf[nxt], in, L, t^nxt);
    in += L;
```

$$\left\{\begin{array}{r}
\mathsf{buf}[\mathsf{cur}] \mapsto b^c * \mathsf{buf}[\mathsf{nxt}] \mapsto b^n \\
* \,\mathsf{arr}_h(\mathsf{in}-((i+1)\cdot\mathsf{L}), (\mathsf{i}-1)\cdot\mathsf{L}, p, is_1) \\
* \,\mathsf{arr}_h(\mathsf{in}, (\mathsf{M}-(\mathsf{i}+1))\cdot\mathsf{L}, p, is_2) \\
* \,\mathsf{pend}(\mathsf{t\hat{}cur}, \{\langle \mathsf{in}_h - 2\mathsf{L}, b^c_\ell, \mathsf{L}, p, is_p\rangle\}) \\
* \,\mathsf{pend}(\mathsf{t\hat{}nxt}, \{\langle \mathsf{in}_h - \mathsf{L}, b^n_\ell, \mathsf{L}, p, is'_p\rangle\}) \\
* \,\mathsf{arr}_h(\mathsf{out}-((i-1)\cdot L), \mathsf{M}\cdot\mathsf{L}, 1, \_) \\
\wedge (is_1 \mathbin{@} is_p \mathbin{@} is'_p \mathbin{@} is_2) = is
\end{array}\right\}$$

```
    wait(t^cur);
```

$$\left\{\begin{array}{r}
\mathsf{buf}[\mathsf{cur}] \mapsto b^c * \mathsf{buf}[\mathsf{nxt}] \mapsto b^n * \mathsf{arr}_\ell(b^c, \mathsf{L}, 1, \_) \\
* \,\mathsf{arr}_h(\mathsf{in}-((i+1)\cdot\mathsf{L}), \mathsf{i}\cdot\mathsf{L}, p, is_1) \\
* \,\mathsf{arr}_h(\mathsf{in}, (\mathsf{M}-(\mathsf{i}+1))\cdot\mathsf{L}, p, is_2) \\
* \,\mathsf{pend}(\mathsf{t\hat{}cur}, \emptyset) * \mathsf{pend}(\mathsf{t\hat{}nxt}, \{\langle \mathsf{in}_h - \mathsf{L}, b^n_\ell, \mathsf{L}, p, is_p\rangle\}) \\
* \,\mathsf{arr}_h(\mathsf{out}-((i-1)\cdot\mathsf{L}), \mathsf{M}\cdot\mathsf{L}, 1, \_) \\
\wedge (is_1 \mathbin{@} is_p \mathbin{@} is_2) = is
\end{array}\right\}$$

```
    put(buf[cur], out, L, t^cur);
    out += L;
    cur = cur^1; nxt = nxt^1;
    i += 1;
```

$$\left\{\begin{array}{r}
\mathsf{buf}[\mathsf{cur}] \mapsto b^c * \mathsf{buf}[\mathsf{nxt}] \mapsto b^n \\
* \,\mathsf{arr}_h(\mathsf{in}-(i\cdot\mathsf{L}), (\mathsf{i}-1)\cdot\mathsf{L}, p, is_1) \\
* \,\mathsf{arr}_h(\mathsf{in}, (\mathsf{M}-\mathsf{i})\cdot\mathsf{L}, p, is_2) \\
* \,\mathsf{pend}(\mathsf{t\hat{}nxt}, \{\langle b^n_\ell, \mathsf{out}_h - \mathsf{L}, \mathsf{L}, 1, \_\rangle\}) \\
* \,\mathsf{pend}(\mathsf{t\hat{}cur}, \{\langle \mathsf{in}_h - \mathsf{L}, b^c_\ell, \mathsf{L}, p, is_p\rangle\}) \\
* \,\mathsf{arr}_h((\mathsf{out}-((i-1)\cdot\mathsf{L}), (\mathsf{i}-2)\cdot\mathsf{L}, 1, os) \\
* \,\mathsf{arr}_h(\mathsf{out}, (\mathsf{M}-(i-1))\cdot\mathsf{L}, 1, os) \\
\wedge (is_1 \mathbin{@} is_p \mathbin{@} is_2) = is
\end{array}\right\}$$

```
}
```

Fig. 4. Invariants in the main loop of the double-buffering algorithm.

Reasoning about fork and join. Verifying master (Fig. 1) requires rules for **fork** and **join**.

$$\frac{\mathsf{f}(\bar{\mathsf{x}}) : \{P\}\_\{Q\} \in \Gamma}{\Gamma \vdash \{P[\bar{\mathsf{e}}/\bar{\mathsf{x}}]\}\ \ \mathsf{t} = \mathbf{fork}(\mathsf{f}, \bar{\mathsf{e}})\ \ \{\mathsf{thr}(\mathsf{t}, \mathsf{f}, \bar{\mathsf{e}})\}}$$

$$\frac{\mathsf{f}(\bar{\mathsf{x}}) : \{P\}\_\{Q\} \in \Gamma}{\Gamma \vdash \{\mathsf{thr}(\mathsf{t}, \mathsf{f}, \bar{\mathsf{e}})\}\ \ \mathsf{v} = \mathbf{join}(\mathsf{t})\ \ \{Q[\bar{\mathsf{e}}; \mathsf{v}/\bar{\mathsf{x}}; \mathrm{ret}]\}}$$

The rules for **fork** and **join** essentially extend the PARALLEL rule to handle threads dynamically. Γ is an environment associating functions with their specifications. Upon forking a new thread, the parent thread obtains the assertion thr that stores information about passed arguments for program variables and gives up ownership of the precondition of the function. Joining requires that the executing thread owns the thread handle which it then exchanges for the function's post-condition.

Soundness. We have defined a formal semantics for multicore programs with asynchronous memory operations and used this to establish the soundness of our method. In particular, if it is possible to derive a proof for a multicore program using our system then the program is guaranteed to be free of data races and memory faults. For reasons of space, we defer a full formal presentation of this result to future work.

IV. Automation

We have built a prototype tool, asyncStar, automating our approach. asyncStar can check proofs written in our logic, and, if supplied with pre- and post-conditions, it can synthesise proofs automatically for many examples, including the double-buffering algorithm of Fig. 1.

Approach. The asyncStar tool uses *symbolic execution* combined with *shape analysis* for separation logic [2], [10]. States in a program are represented symbolically by disjunctions of separation logic assertions. Statements are then executed symbolically over these assertions – that is, assertions are updated to reflect the abstract effect of the statement. The analysis executes until it reaches a fix-point where no new symbolic states can be reached.

Symbolic execution alone does not converge in many cases. To ensure termination, symbolic states are abstracted at the heads of loops. In asyncStar, abstraction of separation logic assertions is based on syntactic summarisation of predicates. For example, a linked list of five nodes might be abstracted by a predicate representing a list of unknown length. This approach, however, is not flexible enough to abstract away numeric constraints arising from loop iterations and array manipulation. To deal with such properties we have developed a novel technique that allows integration of standard abstract interpretation tools.

A. Tool architecture

asyncStar is built on top of coreStar [5], a language-independent verification tool for separation logic, consisting of a symbolic execution engine and a separation logic theorem prover. As input, asyncStar takes a program written in VMC (Verified Multicore C), a fragment of C enriched with user-supplied pre- and post-conditions for functions, and (optionally) loop invariants. The input program is translated into the coreStar intermediate language coreStarIL. asyncStar invokes coreStar's core engine, which symbolically executes the generated coreStarIL program, indicates whether verification succeeded, and returns any inferred invariants. The overall architecture of asyncStar is illustrated in Fig. 5.

coreStar has a language-agnostic internal representation. Support for new languages and abstract domains can be added by providing logic rules and abstraction rules through text input files. Unlike most tools for shape analysis, coreStar does not just deal with heap data. Rather, its analysis allows automatic reasoning about abstract objects such as threads, pending memory operations and so on. By building on coreStar, we were able to develop and test asyncStar rapidly, avoiding re-development of existing features.

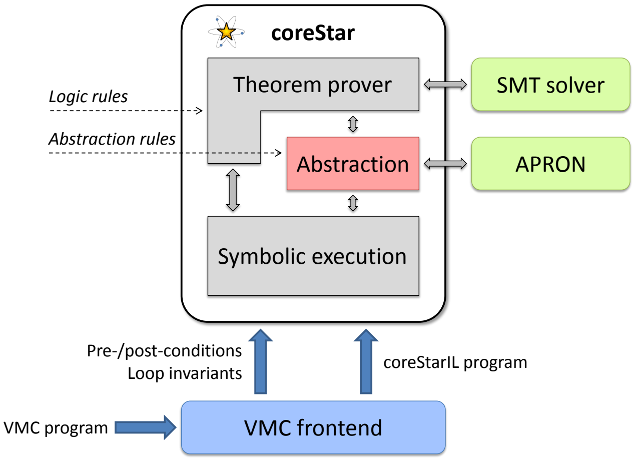


Fig. 5. Architecture of the asyncStar tool

To implement our approach using coreStar, we developed a front-end which accommodates VMC programs and our logic for reasoning about asynchronous memory operations, and extended coreStar's abstraction engine. Developing the front-end involved three major efforts: translating VMC programs into coreStar's intermediate representation (time consuming but straightforward), developing a syntactic encoding of permissions (see §IV-B) and writing logic rules for reasoning about arrays with permissions and pending asynchronous memory operations (technically involved and thus omitted in this paper). Extending coreStar's core engine required two major pieces of work: extending coreStar's symbolic execution with abstraction for first-order domains such as arithmetic (see §IV-C), and adding support for external SMT solvers (see §IV-D).

B. Reasoning syntactically about permissions

Permissions are fundamental to our approach, allowing control over reading and writing for memory shared between threads and pending operations. In §III, we represented permissions by fractions from the interval $(0, 1]$ and permission joining by addition. However, arithmetic reasoning is expensive in coreStar, requiring calls to an SMT solver (see §IV-D). Hence, in our implementation we actually use the binary tree share model [12]. Permission splitting and joining can be reasoned about syntactically using coreStar's rewrite engine, without recourse to arithmetic.

In [12], permissions are represented by binary trees with boolean-valued leaves and unlabelled internal nodes. Maximum permission (1 in the numerical model, corresponding to write permission) is represented by a single true-valued leaf. Splitting maximum permission results in a pair of two-leaf trees, representing read permissions. One tree has a true left and false right leaf, while the other is the other way round, and further splits result in larger trees. Minimum permission is represented by a single false-valued leaf.

We have encoded these binary trees using coreStar's expression language. For example, we might have the following expression representing a tree with three leaves:

$$\mathsf{branch}(\mathsf{leafT}, \mathsf{branch}(\mathsf{leafT}, \mathsf{leafF}))$$

Note that branch, leafT and leafF have no semantics in coreStar; they are treated syntactically, as uninterpreted functions.

C. Abstraction in pure domains

In asyncStar, we extend coreStar's syntactic abstraction with support for abstraction over numerical properties. Our approach uses external tools to implement standard abstract interpretation [9], in which consecutive abstract assertions are joined (and if necessary, widened).

Approach to abstraction in coreStar. In coreStar, the symbolic execution is performed on the control-flow graph of the input program. Each node in a program's control-flow graph is associated with a set of separation logic assertions. Each set represents an abstract state consisting of the disjunction of its elements. By symbolically executing an iteration of a loop, starting from the current abstract state, coreStar constructs a new set of *candidate* assertions. coreStar checks whether each candidate is contained within any of the existing assertions and if not, adds the candidate assertion to the set. Containment is defined via separation logic entailment: an assertion Δ_c is contained within an assertion Δ_a if $\Delta_c \vdash \Delta_a$.[5]

Abstraction is applied to candidate assertions before checking for containment. Prior to asyncStar, the abstraction in coreStar was entirely based on syntactic rewriting of predicates. For example, suppose we had a predicate $\mathsf{node}(x, y)$ representing a node at address x with next pointer y, and a predicate $\mathsf{lseg}(x, y)$ representing a linked list starting at y and ending with a pointer to y. We might write the following rules for abstraction:

$$\begin{aligned}\mathsf{node}(x, x') * \mathsf{node}(x', \mathsf{nil}) &\rightsquigarrow \mathsf{lseg}(x, \mathsf{nil})\\ \mathsf{lseg}(x, x') * \mathsf{node}(x', \mathsf{nil}) &\rightsquigarrow \mathsf{lseg}(x, \mathsf{nil})\\ \mathsf{lseg}(x, x') * \mathsf{lseg}(x', \mathsf{nil}) &\rightsquigarrow \mathsf{lseg}(x, \mathsf{nil})\end{aligned}$$

By applying these rules, the infinite family of assertions representing lists of finite length will be collapsed into a single assertion. This approach works well for reasoning about heap resources, but often fails for concrete values in the first-order, or *pure*, part of the assertion.

For example, consider the following code fragment which sets all the elements of an array a of size n to 0.

```
int i = 0;
while (i < n) {
  *(a+i) = 0;
  i++;
}
```

After the k-th iteration of the loop, symbolic execution will generate the candidate assertion

$$\Delta_k \overset{\mathsf{def}}{=} \exists vs \,.\, \mathsf{i} = k \wedge \mathsf{arr}(\mathsf{a}, \mathsf{n}, 1, vs).$$

The prior abstract state will consist of the set of assertions $\{\Delta_1, \dots, \Delta_{k-1}\}$. The analysis will not terminate, because there exists no $j < k$ such that $\Delta_k \vdash \Delta_j$. Achieving convergence requires abstraction of numerical properties.

Pure abstraction. In asyncStar, we handle such numerical abstraction by using abstract interpretation. As in abstract fix-point calculation, the pure parts of the existing and the candidate assertion are joined and (optionally) employed to widen the existing pure assertion. The obtained abstraction is then used to replace the original pure parts of the assertions. In our example, joining the pure parts of Δ_j and Δ_k (for $j < k$) in the polygonal domain yields $j \leq \mathsf{i} \leq k$. After widening and intersecting with the upper bound we obtain $j \leq \mathsf{i} \leq n$. After replacing the original pure parts of Δ_j and Δ_k the following entailment holds and thus the analysis converges:

$$k \leq \mathsf{i} \leq n \wedge \mathsf{arr}(\mathsf{a}, \mathsf{n}, 1, vs) \;\vdash\; j \leq \mathsf{i} \leq n \wedge \mathsf{arr}(\mathsf{a}, \mathsf{n}, 1, vs)$$

Our approach applies to abstracting the pure part of separation logic assertions in *any* abstract domain. The join and widening operators are assumed to be provided by an external tool for abstract interpretation. asyncStar has a modular interface allowing the calling of such tools during the fixed point calculation. In our experiments, we used the APRON tool [22].

D. SMT utilisation

coreStar's theorem prover was not designed to reason about pure domains such as arithmetic. It reasons purely syntactically, knowing nothing about the semantics of interpreted symbols such as $+$, and thus cannot establish even simple facts like $(a + b) = (b + a)$. For arithmetic reasoning, and to reason about arrays, we call an external SMT solver following the approach proposed in [7]. Calling the SMT solver is expensive, as we transfer the entire proof-state to the solver using the SMT-LIB2 format [1]. We only call the solver in three scenarios: (1) when only pure assertions remain to be proved, in which case they are sent to the solver; (2) when proof search is stuck, in which case the solver is invoked to establish new equalities between terms; (3) when we wish to check whether proof rules with pure guards apply. Queries are memoised to further reduce the expense of solver calls.

V. Experiments

We evaluated asyncStar using a set of benchmarks drawn from the Cell Broadband Engine SDK, which represent prototypical patterns used in Cell programs. We also considered a selection of hand-crafted programs. The Cell benchmarks comprise Euler integration-based particle simulation (*particle-sim*), array processing algorithms using single, double and triple buffering with shared buffers for input and output as in the example of §II (*1-/2-/3-buffer*) and with separate buffers (*1-/2-/3-buffer-IO*). All benchmarks consist of a master thread

[5] Such queries are decided by the separation logic theorem prover.

running on the host that divides array processing among slave threads running on accelerator cores.

The hand-crafted benchmarks do *not* exhibit asynchronous memory operations and are instead designed to assess the effectiveness of our abstraction framework in isolation. Benchmarks with prefix ‘array’ perform an operation on elements of an input array. We consider left-to-right (‘countup’) and right-to-left (‘countdown’) processing of the array, and where elements to be processed are contiguous (suffix ‘1’), or separated by n bytes (suffix ‘n’). Benchmarks with prefix ‘control-flow’ exhibit the same control-flow as the buffering benchmarks but do not contain transfers or processing.

Figure 6 shows experimental results obtained on a personal laptop with 2.8GHz Intel Core2 Duo CPU and 4GB RAM under the Windows 7 operating system. We used the SMT solver Z3 and the polygonal abstract domain from the APRON numerical abstract library. For each benchmark, we give the number of symbolic states, the total execution time (in seconds), and the percentage of execution time spent on computations by the abstract domain library (%AI) and the SMT solver (%SMT). For Cell benchmarks we also apply asyncStar to buggy versions obtained by removing a wait operation. asyncStar fails to find a proof for these examples and prints details of the failed proof attempt, from which the bug can be recovered.

For all benchmarks the user is only required to specify function pre- and post-conditions; loop invariants and intermediate assertions are synthesised automatically. If the analysis succeeds, it proves memory-safety and race-freedom. In the failing case, the user can examine the proof state and look for information exposing the source of the problem.

The results demonstrate that our technique can prove or refute our benchmark set within reasonable time bounds. We observe that spatial reasoning is not a bottleneck *per se* (nor is numerical abstraction), but the arithmetical constraints (arising from proof rules guards and arithmetic in the code) discharged to the SMT solver are. We anticipate that in future the expense of SMT utilisation can be significantly reduced by optimisations in our implementation, such as better caching of queries.

VI. Related Work

In prior work, we presented a preliminary outline of our verification technique [6].

asyncStar is built on coreStar [5], which inherits its approach from jStar [11] and a series of prior tools: syntactic reasoning in separation logic was pioneered in SmallFoot [2]; coreStar’s generation of loop invariants uses on shape analysis based on separation logic [10]. Our approach to pure abstraction is similar to that of [25]. These prior tools achieved performance by hard-coding their domain into the tool. In contrast, coreStar’s core is designed as a modular back-end, intended for use in experiments with verification. This approach is validated by our success in quickly building a prototype on top of coreStar.

An approach to checking DMA races for Cell BE using bounded model checking (BMC) and k-induction has been proposed in [14], [15] and implemented as the Scratch tool [16]. The technique has been extended with abstract interpretation to automatically infer loop invariants [13]. Unlike our method, this technique can only be applied to *sequential* software: safety of DMA races is checked with respect to one thread. Extending BMC to concurrent programs would not scale well as it would involve thread interleaving at a global scope. In contrast, our technique verifies threads in a modular fashion and can be used to prove safety in the presence of dynamic thread creation.

General-purpose techniques for race detection (*e.g.* [17], [24], [27]) could be adapted to handle asynchronous memory operations, and runtime monitoring [20] and system simulation [23] have been applied directly in this setting. However, none of these techniques can *prove* absence of races, which is the contribution of our work.

In [19], session types are used for synthesis of low-level data-movement code, allowing the generation asynchronous memory operations that are race-free by construction. We view this approach as complimentary to our approach.

VII. Conclusions and future work

We have presented a novel method and tool for verifying the memory safety of multicore programs that use asynchronous memory operations. Our approach allows full verification of industrial benchmarks from the IBM Cell SDK, which is beyond the reach of current techniques based on model checking.

We plan to extend asyncStar to take into account alignment restrictions associated with data transfers. In the Cell BE architecture, DMA operations must operate on 16-byte aligned pointers, otherwise behaviour is undefined. Misalignment can lead to subtle errors, which are an ideal candidate for verification. We also plan to evaluate our technique on benchmarks from other domains where asynchronous memory operations are important.

We are exploring the use of *abduction* [8] to reduce the need to annotate functions with specifications. Our analysis uses *frame inference* to divide resources into operation preconditions and frames unaffected by the operation. Abduction generates *antiframes*, consisting of the minimal extra resource needed for the operation to execute without error. Antiframes can be pushed to the start of a function, automatically generating function preconditions. Our preliminary experiments using this technique can discover a full specification for the single-buffer benchmark.

Acknowledgements. This work was supported by the EPSRC, RAEng and the Gates trust. We are grateful to George Russell, John Wickerson and the anonymous reviewers for their insightful comments on an earlier draft of this work.

Benchmark	Correct				Buggy			
	Symbolic states	Total time	%AI	%SMT	Symbolic states	Total time	%AI	%SMT
particle-sim	564	331	< 1	98	58	27	2	97
1-buffer	67	13	< 1	89	32	7	< 1	92
1-buffer-IO	80	31	< 1	94	36	16	< 1	96
2-buffer	259	1268	< 1	> 99	82	318	< 1	> 99
2-buffer-IO	286	1871	< 1	> 99	88	389	< 1	> 99
3-buffer	412	7681	< 1	> 99	113	618	< 1	> 99
3-buffer-IO	443	8416	< 1	> 99	121	663	< 1	> 99

Benchmark	Correct			
	Symbolic states	Total time	%AI	%SMT
array-countup-1	23	0.42	3	28
array-countup-n	27	0.49	1	34
array-countdown-1	31	1.08	3	53
array-countdown-n	34	1.17	2	56
control-flow-sb	21	0.44	3	11
control-flow-db	58	1.56	4	14
control-flow-tb	138	6.85	18	21

Fig. 6. Experimental results obtained using asyncStar to analyse correct and buggy benchmarks

References

[1] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard: Version 2.0. Technical report, 2010.

[2] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2005.

[3] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, 2005.

[4] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *POPL*, 2005.

[5] M. Botinčan, D. Distefano, M. Dodds, R. Griore, Naudžiūnienė, and M. Parkinson. coreStar: The core of jstar. In *Boogie: First International Workshop on Intermediate Verification Languages*, 2011.

[6] M. Botinčan, M. Dodds, A. F. Donaldson, and M. J. Parkinson. Automatic safety proofs for asynchronous memory operations. In *PPOPP*, pages 313–314. ACM, 2011.

[7] M. Botinčan, M. J. Parkinson, and W. Schulte. Separation logic verification of C programs with an SMT solver. *ENTCS*, 254, 2009.

[8] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[10] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.

[11] D. Distefano and M. J. Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*, 2008.

[12] R. Dockins, A. Hobor, and A. W. Appel. A fresh look at separation algebras and share accounting. In *APLAS*, 2009.

[13] A. F. Donaldson, L. Haller, and D. Kroening. Strengthening induction-based race checking with lightweight static analysis. In *VMCAI*, 2011.

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

[15] A. F. Donaldson, D. Kroening, and P. Rümmer. Automatic analysis of DMA races using model checking and -induction. *Formal Methods in System Design*, 39(1):83–113, 2011.

[16] A. F. Donaldson, D. Kroening, and P. Rümmer. Scratch: a tool for automatic analysis of dma races. In *PPOPP*, pages 311–312. ACM, 2011.

[17] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.

[18] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *OSDI*, 2010.

[19] K. Honda, V. T. Vasconcelos, and N. Yoshida. Type-directed compilation for multicore programming. *ENTCS*, 241, 2009.

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

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

[22] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, 2009.

[23] M. Kistler and D. Brokenshire. Detecting race conditions in asynchronous DMA operations with full system simulation. In *ISPASS*. IEEE, 2011.

[24] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI*. ACM, 2006.

[25] S. Qin, G. He, C. Luo, and W.-N. Chin. Loop invariant synthesis in a combined domain. In *ICFEM*, 2010.

[26] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[27] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACMTOCS*, 15(4), 1997.