Automatic Safety Proofs for Asynchronous Memory Operations *

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

Alastair F. Donaldson
University of Oxford, UK
alastair.donaldson@comlab.ox.ac.uk

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

Abstract

We present a work-in-progress proof system and tool, based on separation logic, for analysing memory safety of multicore programs that use asynchronous memory operations.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying, Verifying & Reasoning about Programs

General Terms Languages, Theory, Verification

Keywords Separation logic, memory safety, concurrency

1. 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: cores can delegate data-movement to dedicated hardware, and continue processing on fast, private memory, without contention. Asynchronous memory operations are widely available, e.g. DMA operations in the Cell BE, I/O Acceleration Technology in Intel Xeon cores, and asynchronous memory copying in CUDA/OpenCL.

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 it before the operation completes. If a function returns while an asynchronous operation on a local variable is still pending (perhaps as a part of a speculative pre-fetch), the operation may corrupt the stack frame of subsequently called functions. These problems are compounded in a multithreaded setting: incorrectly managed asynchronous operations can lead to inter-core data races. Asynchronous operations run concurrently with other threads, and they have undefined behaviour over written memory until synchronised. The defects described above thus lead to highly nondeterministic behaviour, thus buggy programs may behave entirely correctly on some implementations, while failing dramatically on others.

We report work in progress on a methodology and tool for automatically proving safety of multicore programs that use asynchronous memory operations. Reasoning about such operations involves complex flow-, path- and context-sensitive analysis of accesses to possibly overlapping regions of memory. We propose a deductive, proof-based approach that enables automation of this analysis. If a proof can be found via our method then the program is guaranteed to be race-free and memory-safe. We show how proofs are performed with an example drawn from the IBM Cell SDK [4]. We then outline our approach to automating this proof strategy.

```
void master(char src[length], char dst[length]) {
  tid* t[N]; int i;        // 'master' runs on host core.
  for (i=0; i<N; i++)
    { t[i] = fork(slave, src+i*M*L, dst+i*M*L, M, i); }
  for (i=0; i<N; i++) { join(t[i]); }
}
void slave(char* in, char* out, int M, int t) {
  char buf[L]; int i = 0;
  while (i < M) {
    get(buf, in, L, t);    // 'slave' runs on accelerator cores, e.g. SPEs
    wait(t);               // 'in' and 'out' point to host memory
    in = in + L;           // 'buf' allocated in scratch−pad memory
    // Process data in the array
    put(buf, out, L, t);
    wait(t);
    out = out + L; i = i + 1;
} }
```

Figure 1. Data processing example using single buffering.

2. Proving safety: a worked example

Single buffering. In the example of Fig. 1, a master thread divides array processing among several slave threads. The master thread runs on the host core, while slave threads run on accelerator cores equipped with scratch-pad memory, e.g. SPEs in the Cell BE architecture. In the slave function, operation **get/put** initiates an asynchronous copy to/from the thread's private memory, respectively. Operations are parameterised by associated tags; calling **wait** on a tag t blocks until all operations associated with t have completed. The function slave uses these operations to implement single buffering. An internal buffer of size L is used to store chunks of the shared array. Operations **get** and **put** are used to fill the buffer and push the results of processing (details of which are omitted) back to the main program. More complex algorithms, using double- or triple-buffering, exploit asynchrony to improve performance by overlapping computation with communication.

Proving safety. Our system is based on separation logic [5], which extends Hoare logic to permit reasoning about dynamically allocated data structures. This is achieved via a new *spatial conjunction* logical connective: $P_1 * P_2$ means that there exists a splitting of the

* This work was supported by the EPSRC, RAEng and the Gates trust.

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.
ACM 978-1-4503-0119-0/11/02.

memory into two disjoint parts, one satisfying P_1 and the other P_2. Separation logic also has a new rule, the frame rule, which allows us just consider the memory that a command changes, and know that the rest is unchanged implicitly.

To reason about the single-buffer example, we introduce two new assertions: $\mathsf{arr}(x, l)$ denotes that x is the address of the first element of an array segment of length l; $\mathsf{MFC}(\mathsf{t}, \mathcal{O})$ (for '*memory flow controller*'), denotes pending asynchronous memory operations. The argument t is the tag controlling the operations, while $\mathcal{O}$ is the set of pending operations. When a thread calls **get** or **put**, the arguments of the operation are added to the set, as follows:

$$\left\{\begin{array}{l}\mathsf{arr}(\mathsf{l},\mathsf{s}) * \mathsf{arr}(\mathsf{h},\mathsf{s}) \\ \quad * \mathsf{MFC}(\mathsf{t},\mathcal{O})\end{array}\right\} \mathbf{get}/\mathbf{put}(\mathsf{l},\mathsf{h},\mathsf{s},\mathsf{t}) \left\{\mathsf{MFC}\,(\mathsf{t}, \{\langle \mathsf{l},\mathsf{h},\mathsf{s}\rangle\} \cup \mathcal{O})\right\}$$

Note that the source and target arrays disappear in the operation post-conditions. Intuitively, these resources are given away to the memory flow controller until the operation finishes. This transfer of resource is essential to ensure race-freedom of the client program.

The above specifications require that each operation can write to both local and shared arrays. In our real proof system, we additionally parameterise arrays by a permission level π, controlling read and write access. If $\pi \in (0..1)$ then the elements of the array can only be read by the thread, while if $\pi = \mathbf{1}$ then the elements can also be written to. The specification for **get** is then as follows (the specification for **put** is dual):

$$\left\{\begin{array}{l}\mathsf{arr}(\mathbf{1},\mathsf{l},\mathsf{s}) * \mathsf{arr}(\pi,\mathsf{h},\mathsf{s}) \\ \quad * \mathsf{MFC}(\mathsf{t},\mathcal{O})\end{array}\right\} \mathbf{get}(\mathsf{l},\mathsf{h},\mathsf{s},\mathsf{t}) \left\{\mathsf{MFC}\,(\mathsf{t}, \{\langle \mathsf{l},\mathsf{h},\mathsf{s}\rangle\} \cup \mathcal{O})\right\}$$

When a thread calls **wait**(), the resources held by the memory flow controller are returned. Here $\circledast$ is the iterated version of $*$:

$$\{\mathsf{MFC}(\mathsf{t},\mathcal{O})\} \quad \mathbf{wait}(\mathsf{t}) \quad \left\{\begin{array}{l}\mathsf{MFC}(t,\emptyset) * \\ \circledast_{\langle \mathsf{l},\mathsf{h},\mathsf{s}\rangle \in \mathcal{O}}.\ \mathsf{arr}(\mathsf{l},\mathsf{s}) * \mathsf{arr}(\mathsf{h},\mathsf{s})\end{array}\right\}$$

The precondition to slave must ensure that the function can read elements of the in array and write to elements of the out array:

$$\mathsf{arr}(\mathsf{in}, \mathsf{M}\cdot\mathsf{L}) * \mathsf{arr}(\mathsf{out}, \mathsf{M}\cdot\mathsf{L}) \tag{1}$$

The proof for the body of the **while** loop in slave is given in Fig. 2; it uses the array segment splitting rule that allows $\mathsf{arr}(x, l)$ to be split as $\mathsf{arr}(x, i) * \mathsf{arr}(x{+}i, l{-}i)$ for $0 < i < l$. This rule is also applied backwards to combine smaller array segments into a bigger one. At slave's exit point we end up with the postcondition having the same symbolic representation as the precondition.

Verifying master (Fig. 3) requires rules for **fork** and **join**. Here Γ is an environment associating threads with specifications.

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

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

Upon forking a new thread, the parent thread obtains the assertion thread 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 postcondition.

3. Automating the proof system

We have built a prototype tool for automating proofs of the kind illustrated in §2; the proof annotations in Figs. 2 and 3, marked in blue, were synthesized by our tool. The tool accepts Cell BE programs written in a C fragment, where asynchronous memory operations correspond to DMA requests. An input program is translated

```
while (i < M) {
 {arr(in−(i·L), L·M) * arr(out−(i·L), L·M)
                 * arr(buf, L) * MFC(t, ∅) ∧ i < M}
 get(in, buf, L, t);
 {arr(in−(i·L), i·L) * arr(in+L, L·(M−i−1)) *
  arr(out−(i·L), L·M) * MFC(t, {⟨in, buf, L⟩}) ∧ i < M}
 wait(t);
 {arr(in−(i·L), L·M) * arr(out−(i·L), L·M)
                 * arr(buf, L) * MFC(t, ∅) ∧ i < M}
 in = in + L;
 // Process data in the array.
 put(buf, out, L, t);
 {arr(out−(i·L), i·L) * arr(out + L, L·(M−i−1)) *
  arr(in−((i+1)·L), L·M) * MFC(t, {⟨buf, out, L⟩}) ∧ i < M}
 wait(t);
 {arr(in−((i+1)·L), L·M) * arr(out−(i·L), L·M) *
      arr(buf, L) * MFC(t, {⟨buf, out, L⟩}) ∧ i < M}
 out = out + L; i = i + 1;
}
```

Figure 2. Proof of the main loop of the slave function.

```
{arr(src, N·M·L) * arr(dst, N·M·L)}
 for(i=0; i<N; i++){ t[i] = fork(slave, src+i*M*L, dst+i*M*L, M, i)}
{⊛_{0≤i<N}. thread(t[i], slave, ⟨src+i·M·L, dst+i·M·L, M, i⟩)}
 for(i=0; i<N; i++){ join(t[i])}
{arr(src, N·M·L) * arr(dst, N·M·L)}
```

Figure 3. Proof of the master's body.

into the internal representation of jStar, a theorem prover for separation logic [2]. Using jStar, we have built a fully-fledged proof system for race-freedom and memory safety in presence of thread concurrency, incorporating the proof rules of §2. Asynchronous memory operations are defined at the byte level but are usually applied to structured data. Our logic uses rewrite rules to marshal data between structured and byte-level representations, according to the C memory model and Cell alignment rules. Arithmetic obligations are discharged to an SMT solver. Currently, the user is required to manually specify pre- and post-conditions for functions (e.g., like the one in Eq. 1 for the slave function). To avoid this, we plan to use *abduction* [1]. Abduction automatically generates (approximations of) resources needed for an operation to execute without error. They can be pushed to the start of a function, automatically generating (partial) preconditions, and from that function specifications.

Unlike an approach to solving this problem based on model checking [3], our method applies to concurrent programs that use dynamic thread creation. Our implementation focuses on Cell BE programs, but could be applied to other platforms (e.g., CUDA and OpenCL). We are intentionally building our logic and our tools to be generic and modular, thus adaptable to these other settings.

References

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

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

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

[4] IBM. Cell BE, 2009. http://ibm.com/developerworks/power/cell.

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