Location—the other confinement form

(position paper: the Geneva convention in a changed world)

Alan Mycroft

University of Cambridge Computer Laboratory
William Gates Building, 15 JJ Thomson Avenue,
Cambridge CB3 0FD, UK
Firstname.Lastname@cl.cam.ac.uk

Abstract. Historically, concerns about aliasing and confinement in object-oriented languages arose from Software Engineering or Program Verification sources: spaghetti-data resulted in programs too delicate to modify or "... we could prove Q if only we knew that x is not aliased". Nowadays there is a further source of aliasing and confinement issues: performance (and even correctness) concerns due to object placement within memory hierarchies on heterogeneous multicore architectures.

1 Introduction

The "Geneva convention on the treatment of object aliasing" (Hogg, Lea, Wills, de Champeaux and Holt 1992) captured perfectly the worries of the age—programs by-and-large ran on a single CPU and, while object orientation had enabled larger programs for the same maintenance burden, the problems of aliasing and confinement were seen as issues both to Software Engineers and to Program Verifiers.

Issues such as concurrency and physical distribution were seen as rather peripheral—threads ("be careful to use locks") and RPC/RMI ("data is marshalled" without too much worry about effects on aliasing and object identity). Indeed object-oriented languages largely co-evolved contemporaneously with the ever-faster single-core processor epitomised by the x86 with its preservation of binary compatibility during vast architectural upheavals.

Today the world has an additional source of concerns resulting from multicore computing—including potentially non-uniform and non-sequentially-consistent memory, and heterogeneous processors. These bring their own issues of confinement, for example even on a multicore x86 having a common address space, suppose program P having two threads runs correctly on a single core processor:

- P may not even run correctly with its threads running on two separate cores (most multicore architectures allow reads and writes to disjoint locations to overtake one another leading to non-sequentially-consistent behaviour).

– Even if P runs correctly (e.g. by inserting locking or `mfence` instructions in any lock-free data structures used by multiple threads) then it may run slower on two separate cores rather than as two tasks on a single core due to cache effects: writing on one processor to an address in a given cache line invalidates all addresses in the same cache line on all other processors.

The situation is of course far worse when memory is non-uniform: memory addresses valid on one core may not be valid on another core, or accessing data located on one core may be very costly on another core (and copying data requires detailed understanding of aliasing). A GPU/CPU combination is a good example.

It is noteworthy that the issue of data-structure (we should read 'object') *placement* is problematic in the most popular languages for these architectures. OpenMP provides pragmas which *permit* a compiler to parallelise code for homogeneous shared-memory processors but with no safety guarantees (e.g. if a later iteration of a loop writes to a location written or read by an earlier iteration); moreover support for heterogeneity is currently lacking. On the other hand CUDA and OpenCL provide for detailed placement of code and data on a CPU or GPU core, but all memory transfers have to be coded explicitly—and again with little linguistic support.

A classical example of something we understand as ownership transfer (here tied to procedure call and return) is the implementation choice in Fortran of whether parameters are passed to a procedure by reference, or by value-result (Ada `in out`)—see e.g. "Advanced compiler design and implementation" (Muchnick 1997). Entertainingly for us, Fortran requires a conforming program not exploit any aliasing to determine the implementation used. We interpret this, slightly more strictly than Fortran requires, as the value being passed having *ownership of the right to access it* transferred from the the actual parameter to the formal parameter for the interval between call and return.

2 Object-orientation and object migration

Software architectures like CORBA provide for distributed (read as having multiple disjoint address spaces) object-oriented systems which might appear to be useful conceptually for multicore. However, CORBA provides no built-in support for objects to move, and therefore method calls on them suffer the cross-memory-access (possibly network) costs. Software wishing to move objects around for performance reasons faces the same lack of linguistic support as in CUDA and OpenCL above.

What we would really like is to be able to *migrate* objects from one location or address space[1] to another. We are interested in doing so in a way which does not affect object identity—any copying should not be visible semantically;

[1] It suffices to consider the case of migrating an object from one location to another (we can always conceptually view (processor,address) pairs as enriched addresses in a single address space).

this can be controlled by the Geneva-convention features of aliasing: detection, advertisement, prevention and control.

In referentially transparent languages, object identity plays little role: a copy of an object is indistinguishable from an alias. In mainstream object-oriented languages, object identity is captured by location, and copies are distinguished from aliases by the different effects (observed by method calls) which methods have on them. In general, we can move an object from one address to another one provided we simultaneously—effectively atomically—update all references to that object to point to the new address. This process happens without our noticing during copying or compacting garbage collection; it can be seen as changing an object's address without changing its identity.

The question to consider is: under what circumstances may we copy an object (splitting its identity), do some operations on the copy, and eventually recover or recreate a single object which has the same behaviour as if we had done the operations on the object directly.

A trivial case is when an object has a single[2] reference. Suppose class `C` has an `int` field `f` and a copy constructor; consider the methods

```
C m1(C x) { x.f++; return x; }
C m2(C x) { C y = C(x); y.f++; return y; }
```

Normally `m1` and `m2` are distinguishable, but the calls `m1(o)` and `m2(o)` are indistinguishable if `o` holds the only reference to an object and `o` is dead after the call.

3 Caches and Ownership

It is noteworthy that caches work best when used as if they represent "single writer multiple reader" regions. Cache coherency protocols invalidate, on a write by a given processor, that cache line for all other processors (which then must re-read the cache line on a subsequent access). It is instructive to consider this in terms of ownership, here interpreted as performance ownership: logically a write operation on a given processor transfers 'fast access' *ownership* to that processor and removes it from all other processors; By contrast a read operation on a given processor adds that processor to the set of processors sharing ownership of that cache line. We may consider the cache line to *migrate* on writing. This view of 'physical' cache lines does not align perfectly with objects (which typically can cross cache-line boundaries, and indeed parts of two objects may share a single cache line). However, at some cost in fragmentation we may pad objects so that each cache line corresponds to at most one object. Alternatively we may ignore the issue of two objects sharing a cache line; this hits performance in the situation (hopefully rare dynamically) when two logically isolated objects are processed concurrently by separate cores—ownership is extraneously exchanged on the cache line even though the objects are isolated.

[2] From a local variable rather than an instance variable (a.k.a. heap object).

It might appear at first sight that cache-coherent single-address-space architectures are likely to need a different approach from heterogeneous architectures requiring explicit DMA-like transfers between processors. However, while there are often speed differences between message passing in the cache-coherency protocol and in a DMA operation (and therefore the penalties for having data in the wrong place differ), both architectures benefit from the same optimisation of having data being migrated to be close to the processing element. Just because the message passing in a cache-coherency protocol happens behind the scenes does not mean the total loss of performance from a program which uses data inappropriately is proportionately less than for the same program using explicit DMA-style message passing.

4 How useful are existing techniques?

A wide variety of work has addressed the issue of expressing possible run-time linkages between objects, ownership issues, notably taking inspiration both from Software Engineering and from semantics-based theory:

- owners as dominators
- owners as writers
- linear and other substructural types, region-based types and memory management, uniqueness types and separation logic.
- shape-analysis-based techniques

The ideal criterion for temporarily migrating ("borrowing") an object from one address (or address space) to another is that, during the period the object is migrated, no access to the object occurs via any of its aliases.

A promising technique for bridging the gap between detailed hardware cache properties and high-level syntactic representations of cache expectation is a variant of "Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only" (Boyland, Noble and Retert, ECOOP 2001). We suppose a multi-core processor having cores (each with its own cache) ranged over by $\ell \in \mathcal{L}$; cores have a common address space. As before pointers are represented by capabilities (o, r) where o is an object address and r is a set of access rights. Access rights r are extended from the existing $\{\mathbf{R}, \mathbf{W}, \mathbf{I}, \mathbf{O}\}$ to include $\mathbf{H}_\ell$. Both read and write operations on capability (o, r), by a command executing on core ℓ, add $\mathbf{H}_\ell$ to the access rights r' of all capabilities (o, r'); however writes first remove $\{\mathbf{H}_{\ell'} \mid \ell' \in \mathcal{L}\}$ from all such capabilities. Stacks (and hence local-variable addresses) are unshared between processors and so do not need operations on involving $\mathbf{H}_\ell$.

Note that cache occupancy is not strictly an access right in the sense used elsewhere in that paper because it is a property which can be lost and then regained at the semantic level.

4.1 Concrete syntactic proposals

A challenge left for future work is to design static type systems and interface specification to match the capability-style semantics detailed above.

However, note that Kilim (Srinivasan and Mycroft, ECOOP 2008) provides a set of modifiers which enable tree-like object structures to be transferred between processes while respecting the isolation property[3] that each object is only accessible by one processor. This system readily allows read-only objects to be shared between multiple processors, providing a direct match to natural cache-ownership ownership properties discussed in Section 3.

It is worth noting that the discussion "Why we should not add readonly to Java (yet)" (Boyland 2005) on whether high-level capabilities like 'readonly' should be transitive in their effects has its echoes here. Clearly at a low-level each address has its own associated cache line and node in a data structure can be independently present or absent from a cache. However, there is plenty of scope for high-level interfaces to specify syntactically how much of a data structure will be loaded into cache by a procedure, e.g. to enable cache pre-fetch optimisation.

A much wider question is whether we can harness other forms of ownership to facilitate migration. For example while owners-as-writers does not fit well with cache-like properties, owners-as-dominators provides a way to (specify and) transfer an object and all its owned objects to another process.

5 Conclusions

This position paper argued that the problems of exploiting multicore, possibly in heterogeneous-processor or non-uniform-memory forms, provides another input (alongside semantic theory and software engineering) to the problem of designing convenient syntactic means to express aliasing and confinement issues in programing languages.

Acknowledgements

Thanks are due to Dominic Orchard for helpful comments on an earlier draft and to an anonymous referee suggesting the comparison to Boyland's (et al.) "Capabilities for Sharing" work.

[3] Isolation also avoids issues concerning data races and hence removes the possibility, discussed in the Introduction, of non-sequentially-consistent behaviour in modern multicore CPUs.