Advanced Topics in Computer Systems, CS262

Brewer/Hellerstein

Emerald Heterogeneous Migration

April 9, 2001

Motivation: why migrate?

- fault tolerance? restart task in a new place, typically from checkpoint
- load balancing? move objects off of overloaded nodes
- performance? reduce commnunication via migration
  idea: replace lots of RPCs with one expensive migration (possibly two, to move it back)
- main alternative: careful placement of new objects to balance load
- overall: advantages are not that clear and there is great deal of complexity...

Easy migration:

- use only machine independent form, such as Java byte codes. Interpret or JIT compile threads as needed -- this seems like a viable solution given JIT compilation to address performance
- interpreted languages (eg. tcl/perl) are relatively easy to migrate, but still need to package up all of the state. (why easier in tcl than perl? tcl uses only strings)

(homogeneous) native code migration:

- has performance win: threads run at full speed
- big challenge: how to package up state
- code and function pointers is easy in the homogeneous case, since code base is the same
- hard part: registers, stack, objects
- challenge: which values are scalars (e.g. ints) and which are pointers? Pointers must be converted to equivalent values on new host. implies: 1) must know for every register, stack frame location, and heap location whether something is a pointer (at the time of migration!), and 2) must be able to tell where the corresponding object is...
- pointers into local objects? either convert to global references or make a stub object on the new host and point to it.
- unions are very hard. Emerald disallows unions that are not all scalar or all pointers, otherwise can't tell if you need to change the pointer. Unlike conservative GC, must have a 100% answer (or don't migrate the object)
- indirect addressing (e.g. via stack pointer) is helpful and needs no translation
- compiler produces template for every object and every stack frame to indicate the pointers
- also need a convetion for registers... easiest one is to partition registers into pointers and scalars and don't mix them. more complex: have stack template define register use as well

heterogeneous native code migration:

- hard problem 1: code pointers and function pointers now need translation
- basic solution: map code pointers to machine independent locations then map them back to machine dependent on the new archictecture
- hard problem 2: no all code locations HAVE a viable translation. Examples: reordered code, code that is hoisted on one platfrom but not on another, atomic instructions that correspond to many instructions on another platform means that you can't map from the middle of the sequence back to the host with the atomic instruction
- general solution: define bus stops: points at which code maps 1-1. for example, at bus stop, both versions have executed the same semantic set of instructions.
- Key insight: between bus stops, the programs need not correspond 1-1 -- compilers are free to optimize as they like.
- Typical bus stop locations: procedure/system call boundaries, the bottom of loops
- required compiler support: bus stop mappings, use of register, temp variables, etc.

Bridging code (not implemented):

- idea: migrate at points with no direct mapping
- solution: 1) define location in UNoptimized code, 2) bridge to that point, 3) migrate, 4) bridge from there to optimized location on target host
- not implemented or evaluated

other tidbits:

- when do you need a network format? only if the two sides differ and you don't want O(n^2) conversions (for n architectures)
- code is immutable, which means it need not be moved
- how to gain control upon next method call? reset stack pointer so that stack allocation fails (check has to be there anyway) Note: can't really use page faults or modify the code; in theory could modify the return address of the active procedure, but it may not return for a long time, and there is a race condition
- must track the USE of variables/registers at every bus stop: different types may require different conversions (not true in homogeneous case)
- need to make sure different compilers produce the same bus stops (in the same order) -- this was done by hand
- struct conversion is by recursive descent (much like RMI marshalling)
- very hard to debug!! errors show up only during migration and may be path or machine dependent; also very hard to test every bus stop
- stack frames are tranlated at the receiver yougest to oldest, which is the opposite of how they need to be allocated; not clear why this is required. They thus have to move frames after everything is translated. In theory, they could avoid the movement using two phases, one for sizing (quick) and one for translation...
- could you migrate C threads? probably not... unless you use a type-safe subset -- otherwise you can't tell how to translate something (not even homogeneous migration)

- how to tranfers objects that use local resources (such as open TCP connections or the CD-ROM)? probably best to use a proxy object on the remote host
- migration in Java in the presence of JIT compilation? (easy without JIT, since JVM is machine independent) need to migrate only at byte code boundaries, which may require bridging code (to get to the next one), or at least identification of which points map to such boundaries
- de-optimization is also very useful for debugging