Fine-grained mobility in the *Emerald* system

Jul et al. TOCS 1988

538B projects

- Proposals due Friday (tomorrow) at 6pm Vancouver time

Emerald in a nutshell

- PL + Distributed systems
- “Mobility” taken to an extreme
- Object oriented lang: everything is an object. Unit of distribution/migration is an object.
- A process lives inside an object (also mobile)
- Language level support
- Compiler generates code that considers mobility (to target local/remote systems)
- Advantages
  - Can move data without worrying about specifics
  - Objects (not refs) move to remote location to co-locate data and computation
  - Garbage collection to clean up unreachable objects

Emerald

- What are the advantages of mobility? (Section 1: the sell)
    - Load sharing / load balance : move computation/data at runtime, based on programmer's directives (with a static view). Has huge flexibility for moving load. But has no capacity for measuring load or dynamically responding to load.
    - Communication performance: move computation to data (or data to computation). All big data frameworks (pin computation to data and retain it there for a long time). Emerald is highly sequential (lack of parallelism), very fine grained.
    - Availability: Move objects to different nodes to survive failures. What happens on failure? Predict the future (planned outage)? They never discuss replication. My system gets to run even though some nodes are down (because a node has no direct exec dependencies on another node). They provide checkpointing mechanism (store a freezer version of your objects to disk; and then reconstitute them). True to partial failure.
    - Reconfiguration: Move objects on demand (planner outages). Introduce objects as the system executes (no need to shut everything down: not the Java model). This is everywhere.
    - Special capabilities: Move objects (data/compute) to machines with special resources. Object may require 2 resources, each resource on different machine (x86 and ARM processors). Web proxies? VPN? TensorFlow GPUs/TPUs?
- Mobility today
    - ACM SIGMOBILE : emphasis on devices, *not objects*
    - Well, maybe the smartphone revolution has had something to do with this (it's the hardware!)
    - Cyberforaging: https://en.wikipedia.org/wiki/Cyber_foraging (early mobile offloading papers)
    - Finn: *"Deployment engineering; DevOps" ?*

Emerald : trust the compiler

- What are the mobility primitives? (Section 2.3)
  - **Locate** X : returns explicit location of X
  - **Move** X to Z : co-locate objects of nodes X, Z (or explicitly mode to nodeX)
    - Note: kernel not obliged to move (Absurd? Suspicious? Easy?)
    - May not be satisfiable — e.g., Z cannot be found.
  - **Fix** X at nodeY : perf heuristic to pin obj at node
  - **Unfix** X
  - **Refix** X at nodeZ : unfix, move, fix at nodeZ
    - A forcing function (a stricter *move*)
  - **Attach** objects to other objects : transitive, not symmetric (graph of objects) — this provides automatic migration/ mobility of subgraphs; optimization as well
- Why have an explicit location?
- Attachment is defined statically (during definition)
- Heuristics essential for good performance
- “World’s most intelligent serializer” — package and send only what is necessary, with hints!
- Split control of distribution between programmer and the compiler : who gets to control what?

Emerald : trust the compiler

- Calling semantics
- RPC: **call by value** (too difficult to figure out references)
- Argus : **call by value** (…)
- Emerald : call by reference ok! In fact, everything is call by reference (except for small things).

  - **Call-by-move** : move object to callee and keep it there
    - Z.foo(move X) == Move X to Z ; Z.foo(X)
  - **Call-by-visit** : move object to callee and move back to caller
    - Z.foo(X) == Move X to Z ; Z.foo(X) ; Move X back

RPC-Argus-Emerald

- What's the right level of integration with a PL?
  - RPC v Argus+Emerald different levels of integration
  - Programmer convenience?
  - What about programmer **productivity**?
- How much distribution should you hide/expose to the programmer?
  - Argus v Emerald exposure of distributed objects
  - Give developer <u>control</u> over implementation of distribution (mobility / atomicity of distributed exec)
- *MapReduce / Spark* abstractions for distributed "big data" compute
  - "Products" for a specific job : more targeted computation. Huge market for these! It's what people need.
  - Hide distribution/have many features. Not PLs. Easier to learn an API than a PL?
    - APIs work well for external data
    - PLs are hard to learn (especially DSL)

Emerald

- Environment assumptions
  - Homogeneous nodes/machines : simplifies compilation (same code/ISA, stack layout, register set); simplifies translation.
  - Local area network : low latency (UDP for networking)
  - Single administrative domain (all nodes trust all other nodes)
  - Kernel support (Kernel is Emerald-aware)
  - Reliability assumptions? Not in this paper. Though they do discuss checkpointing features in other work.
- Resembles a datacenter… (many of these ideas are being revisited today)

Emerald locating obj

- Locating objects (ARP variant)
  - Check local kernel mapping first
  - If not, and exists forwarding address, ask the node at the address
  - If the node knows another node, as that node
    - If no nodes have the object, then broadcast
    - Timeout, broadcast again
    - If not all nodes replied, ask the nodes that did not reply
    - If timeout, then assume node with object is unavailable
- UOI : unique object identifier
- *Have to know all the nodes (broadcast is going to target all nodes; reason about “node availability” => object availability)*

Emerald GC

- (Distributed) Garbage collection
  - Mark-and-sweep
  - Concurrent with execution
  - Lazy marked : on move, mark an object as “black” (referenced)
  - Reference counting on moves/relocations
- Their system doesn’t have a working version of a GC

Next: Mace PL

- More PL
- This time PLDI 2007 (20 yrs after Emerald!)
- Will transition us into papers on distributed system correctness: verification and model checking