My Goal

- Architecture exploration of a new memory subsystem within a large-scale (64 - 1024 cores) tiled multicore

![Baseline architecture](http://people.csail.mit.edu/devadas/pubs/em2-detailed.pdf)

![Conceptual architecture](http://people.csail.mit.edu/devadas/pubs/em2-detailed.pdf)
Common Goals

• Explore many pthread apps (SPLASH-2, PARSEC etc.)
• Simulate large-scale multicores: 64 => 256 => 1024 cores
• Number/location of memory controllers
• Single shared simulated address space
  – Graphite: Striped and well defined regions for stack and dynamically allocated segments
  – *We explored*: Page Table, Intelligent data placement and Memory mgt.
• Thread Spawning
  – Graphite: mapping of app threads to target cores (fixed at spawn)
  – *We explored*: Migrating threads between target cores (via thread-core mapping)
• System Calls
  – Graphite: Designed for correctness
  – Performance modeling tricky as they go through centralized MCP
System (simulator specific code)

- common/system/{simulator, core_manager}
- Centralized, so a good place to put globally visible hooks
- We used it to store our
  - Page Table
  - Thread-to-Core Mapping
Memory Accesses

- common/tile/core/core.{cc, h}
- This where we trigger migrations!
- Watch out for x86 ISA related issues
  - Data accesses can span multiple cache lines
  - LOCK prefix instructions etc.
- Graphite rewrites memory accesses and
  - Allows control and visibility into accessed data
  - Requires functional correctness (if changed)
Caches & Coherence

- common/tile/memory_subsystem/*
- Directory based coherence
- Protocol variants: MSI, MOSI
- Allows to explore $-$to-$ transfer tradeoffs
- Directory cache variants

+ Access to network API makes the protocol messages easy to track => helped with understanding and stats gathering etc.
+ Easy to modify the protocol e.g.,
  + we got MI protocol working with 1-line change in Graphite!
  + We added directory to memory controller indirection
Network Models

- common/network/*
- Prime example of the *modularity* of Graphite
- Multiple networks can co-exist
  - Easy to model different networks for different purposes e.g., we needed a high-bandwidth, low-latency network for migrations
- Hop-by-Hop model is a detailed version
  - Mesh, XY routing, Contention enabled
  - Allows variable flit size and variable hop latency
- Hop counter is simple and fast
  - Good for initial explorations
Core Models

- `common/tile/core/core_model.{cc,h}`
- Probably the weakest part of Graphite
- Very simple model of a core
- We extended a bit to support:
  - Multithreading (multiple threads per core)
  - Variable issue-slots etc.
- Watch out for x86 ISA related issues
  - Macro Ops vs. Micro Ops
  - String instructions, redirectPopf etc.
• Relied heavily on the tracing capability of Graphite to debug hacks and models

• Stats are distributed among the models and easy to modify/add
Some Observations

• PIN front-end is pretty solid and didn’t require much hacking, but changes can expose to (possibly) nasty errors
  • Be prepared to update several paths in graphite e.g., if you change memory instructions, you need to worry about redirection of some memory operations
• Get familiar with the `carbon_sim.cfg` file and the simulator parameters
• Small tests (unit, apps) are useful to get started and when you add models
• Makefile system is not completely robust. You need to clean, make whenever an .h file changes
• Performance modeling of syscalls is tricky