

EECS 262a

Advanced Topics in Computer Systems

Lecture 13

Resource allocation: Lithe/DRF
March 7th, 2016

John Kubiawicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

<http://www.eecs.berkeley.edu/~kubitron/cs262>

Today's Papers

- **Composing Parallel Software Efficiently with Lithe**
Heidi Pan, Benjamin Hindman, Krste Asanovic. Appears in Conference on Programming Languages Design and Implementation (PLDI), 2010
- **Dominant Resource Fairness: Fair Allocation of Multiple Resources Types**
A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, Usenix NSDI 2011, Boston, MA, March 2011
- Thoughts?

3/7/2016

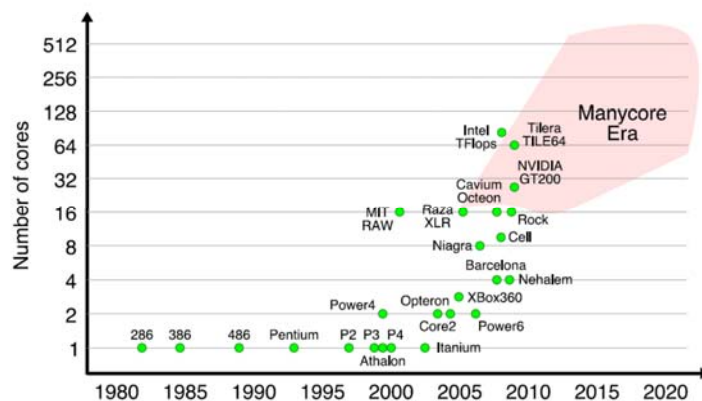
cs262a-S16 Lecture-14

2

The Future is Parallel Software

Challenge: How to build many different large parallel apps that run well?

- ❖ *Can't rely solely on compiler/hardware:* limited parallelism & energy efficiency
- ❖ *Can't rely solely on hand-tuning:* limited programmer productivity

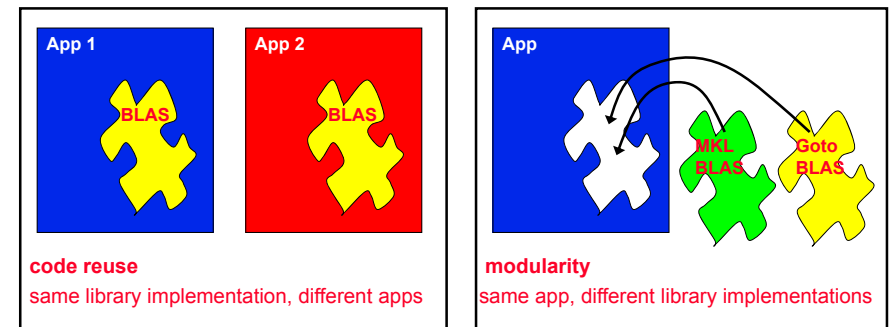


3/7/2016

cs262a-S16 Lecture-14

3

Composability is Essential



Composability is key to building large, complex apps.

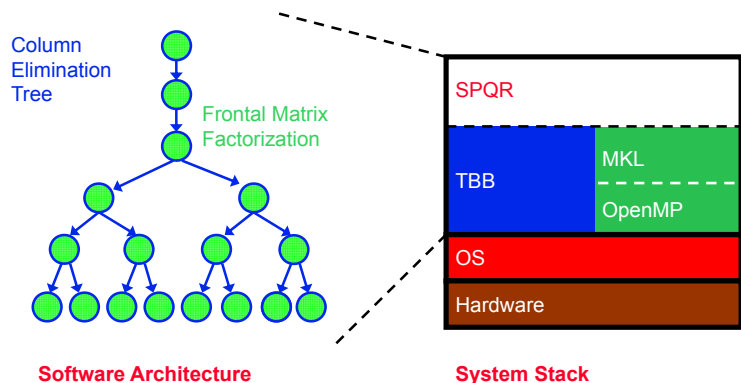
3/7/2016

cs262a-S16 Lecture-14

4

Motivational Example

Sparse QR Factorization
(Tim Davis, Univ of Florida)

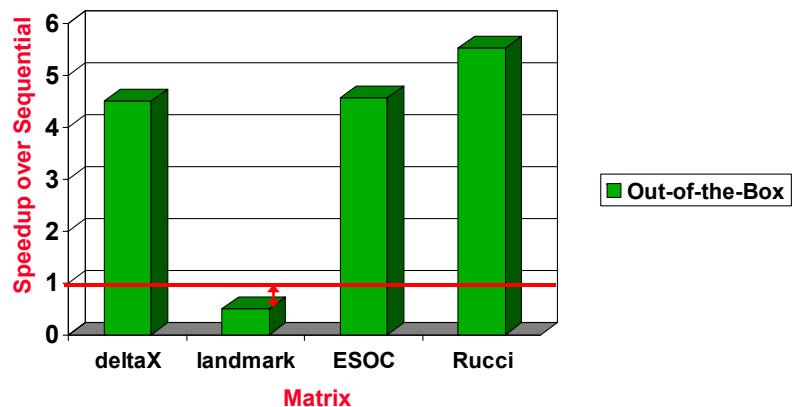


TBB, MKL, OpenMP

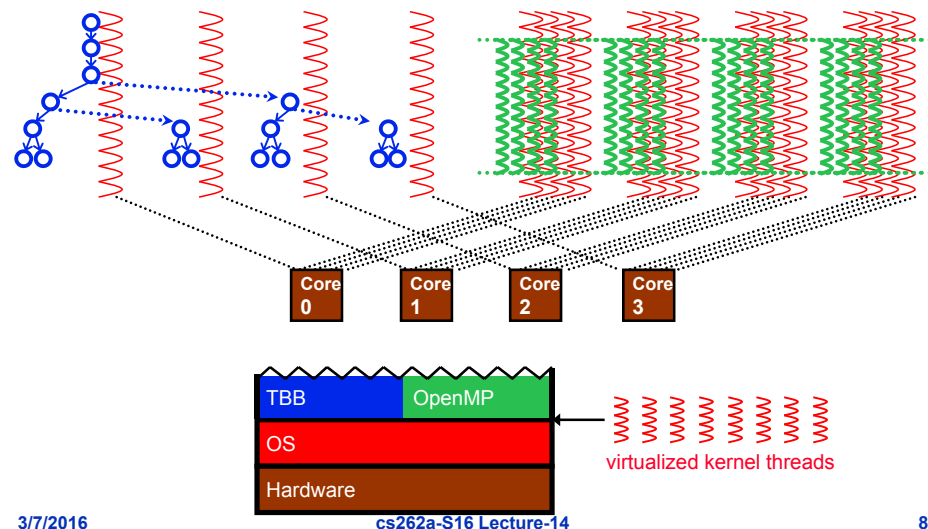
- **Intel's Threading Building Blocks (TBB)**
 - Library that allows programmers to express parallelism using a higher-level, task-based, abstraction
 - Uses work-stealing internally (i.e. Cilk)
 - Open-source
- **Intel's Math Kernel Library (MKL)**
 - Uses OpenMP for parallelism
- **OpenMP**
 - Allows programmers to express parallelism in the SPMD-style using a combination of compiler directives and a runtime library
 - Creates SPMD teams internally (i.e. UPC)
 - Open-source implementation of OpenMP from GNU (libgomp)

Suboptimal Performance

Performance of SPQR on 16-core AMD Opteron System



Out-of-the-Box Configurations



Providing Performance Isolation

Using Intel MKL with Threaded Applications

<http://www.intel.com/support/performance/tools/libraries/mkl/sb/CS-017177.htm>

Software Products

Intel® Math Kernel Library (Intel® MKL)

Page Contents

- Memory Allocation (M): Memory appears to be allocated and not released when calling some Intel MKL routines (e.g. `spqrfs`).
- Using Threading with BLAS and LAPACK
- Setting the Number of Threads for OpenMP (OMP)
- Changing the Number of Processors for Threading During Runtime
- Can I use Intel MKL if I thread my application?

Memory Allocation (MML): Memory appears to be allocated and not released when calling some *InterAPI* routines (e.g. *g_yield*). One of the advantages of using the *InterAPI*, is that it is multithreaded using *OpenMP*. *OpenMP* requires buffers to perform some operations and allocates memory when the single-threaded and/or single-linear applications. This memory allocation occurs since the first time the *OpenMP* software is encountered in the program. This memory allocation persists until the application terminates. In addition, the Windows' operating system will allocate a stack equal to the main stack for every additional thread created, so the amount of memory that is automatically allocated will depend on the main stack, the *OpenMP* allocations and the number of threads used.

Using Threading with BLAS and LAPACK
Intel MKL is threaded in a number of places. LAPACK ("GETRF", "POTRF", "GEMM", "Level 3 BLAS, DFTs, and FFTs, Intel MKL uses OpenMP threading software. The situations in which conflicts can exist that make the use of threads in Intel MKL, provide the list them here with recommendations for dealing with these. First, a brief about

If the user threads the program using OpenMP directives and the Intel C++ compiler the program, Intel MPI, and the user program will both use the same thread (Intel MPI), thus to determine if it is a parallel region in the program, and if it is, spread to operations over multiple threads. But user MPI can be aware that it is a region only if the threaded program and Intel MPI, allowing the same threading to use program is provided by some other means, Intel MPI, user operates to modify

- * User threads the program using OS threads (affinity on Linux*, Win32* threads on Windows*). If more than one thread calls `Intel MKL`, and the function being called is threaded, it is important that threading in `Intel MKL` be turned off. Set `OMP_NUM_THREADS=1` in the environment.

- * Use threads the program using OpenMP directives and/or pragmas and compile the program using a compiler other than a compiler from Intel. This is more problematic because setting `OMP_NUM_THREADS` in the environment affects both the compiler's threading library and the threading

- Multiple programs are running on a multiple-CPU system. In cluster applications, the parallel program can run separate instances of the program on each processor. However, the threading software will use multiple processors on the system even though each processor has a separate process running on it. In this case `OMP_NUM_THREADS` should be set to 1.
- If the variable `OMP_NUM_THREADS` environment variable is not set, then

* Windows®: Open the Environment panel of the System Properties box of the Control Panel on Microsoft® Windows NT®, or it can be set in the shell if the program is running in with the command: set OMP_NUM_THREADS= (number of threads to use).

• Linux*: To set and export the variable: `export OMP_NUM_THREADS=`
 (number of threads to use).

• Setting the variable when running on Microsoft® Windows® 20 or Windows® 7 is a workaround, since multiprocessing is not supported.

Changing the Number of Processors for Threading During Runtime
 It is not possible to change the number of processors during runtime using the environment variable `OMP_NUM_THREADS`. You can call OpenMP API functions from your program to

threads the program using OS threads (as on Windows*). If more than

on being called is threaded, it is
 off. Set OMP_NUM_THREADS

ted off. Set OMP_NUM_THREADS

```

a:=0; b:=0; c:=0; aa:=0; ab:=0; ac:=0; p:=0;
for i:=1 to 100 do
begin
  for j:=1 to 100 do
  begin
    for k:=1 to 100 do
    begin
      a:=a+1; b:=b+1; c:=c+1;
      aa:=aa+1; ab:=ab+1; ac:=ac+1;
      p:=p+1;
    end
  end
end

```

$$\begin{aligned} q^2/12 &= (\text{moder})^2 \\ q^2/12 &= (\text{moder})^2 \\ q^2/12 &= (\text{moder})^2 \end{aligned}$$

```
print('weights: %s')
for i in range(1, n+1):
    print('b0: %f, b1: %f, i: %d' % (b0, b1, i))
}

stop_val = min(weights):

for i in range(1, n+1):
    for j in range(1, n+1):
        w[i][j] = (w[i][j] + (b0 - b1) * (b0 - b1)) / 2
        w[i][j] = (w[i][j] + (b0 - b1) * (b0 - b1)) / 2
        w[i][j] = (w[i][j] + (b0 - b1) * (b0 - b1)) / 2
    }

classifiers = ClassifierAggr. ClassifierTrans. ClassifierTrans
m, n, k, alpha, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z
```

```

printf("vector size:");
for (i=0; i<10; i++)
    printf("%d\t",*(a+i));
printf("\n");

omp_get_num_threads();
for (i=0; i<SIZE; i++)
    for (j=0; j<SIZE; j++)
        a[i*SIZE+j] = (rand()%10);

```

ds (pthreads on Linux*, W
thread calls Intel MKL and

important that threading in I/O is done in the environment.

delete []
delete []
delete []

The Intel Math Kernel Library is designed and compiled for threaded programs that are threaded. Calling Intel MKL routines that application threads can lead to conflict (including incorrect answers). Calling library differs from the Intel MKL threading library.

- User threads the program using OS threads (pthreads on Linux*, Win32* threads on Windows*). If more than one thread calls Intel MKL and the function being called is threaded, it is important that threading in Intel MKL be turned off. Set `OMP_NUM_THREADS=1` in the environment.

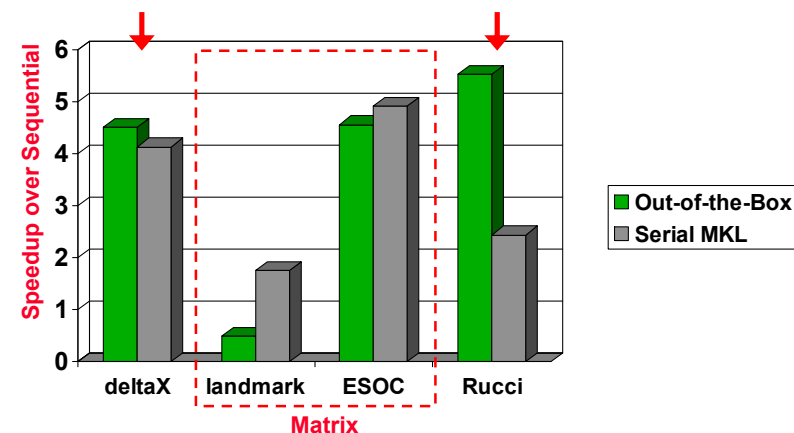
3/7/2016

cs262a-S16 Lecture-14

9

“Tuning” the Code

Performance of SPQR on 16-core AMD Opteron System

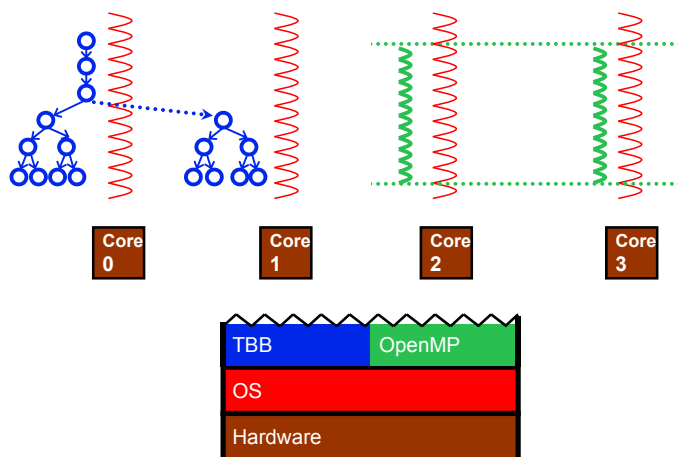


3/7/2016

cs262a-S16 Lecture-14

10

Partition Resources



Tim Davis' "tuned" SPQR by manually partitioning the resources.

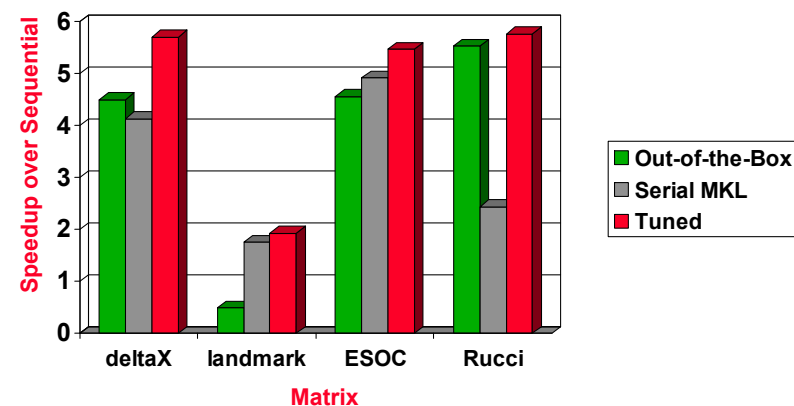
3/7/2016

cs262a-S16 Lecture-14

11

“Tuning” the Code (continued)

Performance of SPQR on 16-core AMD Opteron System



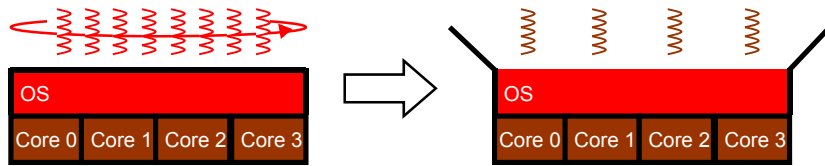
3/7/2016

cs262a-S16 Lecture-14

12

Harts: Hardware Threads

virtualized kernel threads



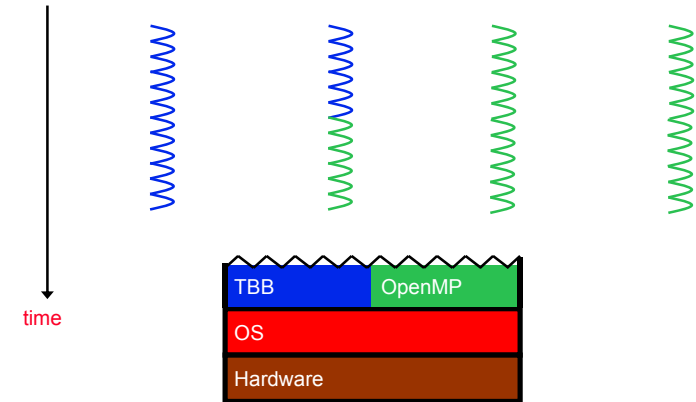
- ❖ **Expose true hardware resources**
 - Applications requests harts from OS
 - Application “schedules” the harts itself (two-level scheduling)
 - Can both space-multiplex and time-multiplex harts ... but never time-multiplex harts of the same application

3/7/2016

cs262a-S16 Lecture-14

13

Sharing Harts (Dynamically)

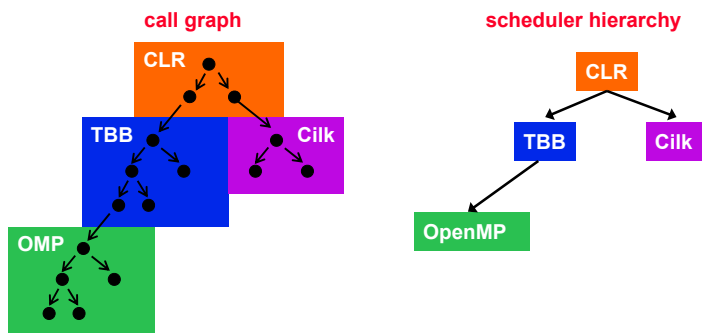


3/7/2016

cs262a-S16 Lecture-14

14

How to Share Harts?



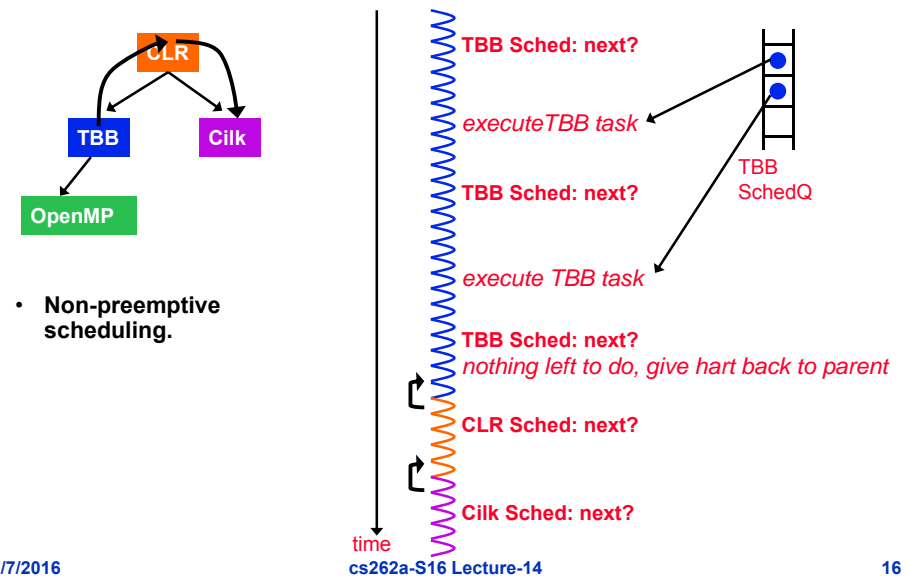
- ❖ **Hierarchically:** Caller gives resources to callee to execute
- ❖ **Cooperatively:** Callee gives resources back to caller when done

3/7/2016

cs262a-S16 Lecture-14

15

A Day in the Life of a Hart

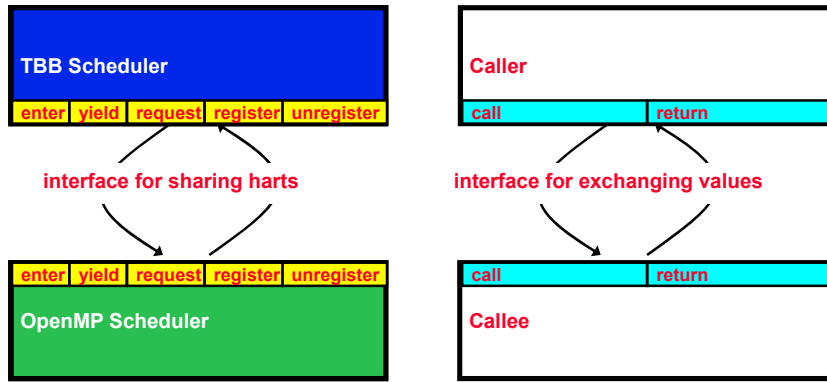


3/7/2016

cs262a-S16 Lecture-14

16

Lithe (ABI)

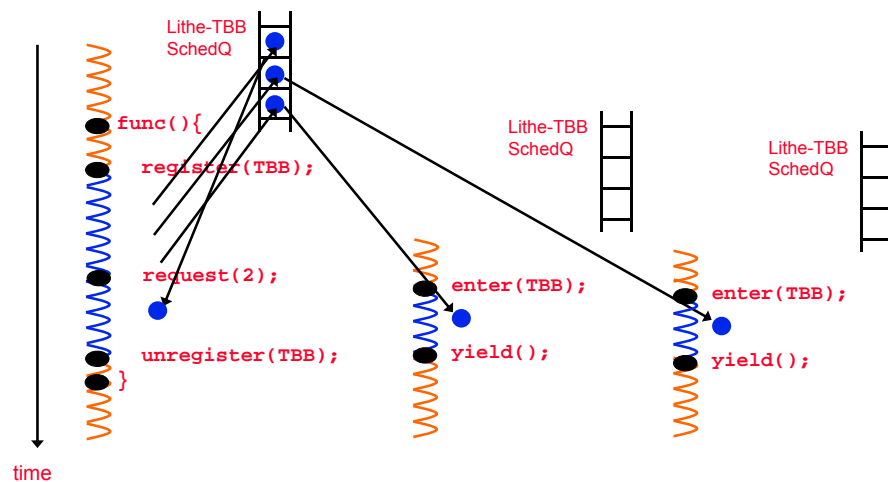


❖ Analogous to function call ABI for enabling interoperable codes.

A Few Details ...

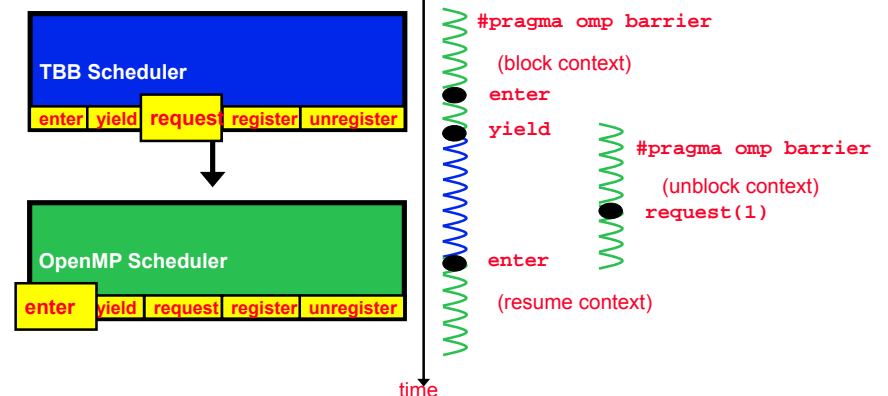
- A hart is only managed by one scheduler at a time
- The Lithe runtime manages the hierarchy of schedulers and the interaction between schedulers
- Lithe ABI only a mechanism to share harts, not policy

Putting It All Together



Synchronization

- Can't block a hart on a synchronization object
- Synchronization objects are implemented by saving the current "context" and having the hart re-enter the current scheduler



Lithe Contexts

- Includes notion of a stack
- Includes context-local storage
- There is a special transition context for each hart that allows it to transition between schedulers easily (i.e. on an enter, yield)

Lithe-compliant Schedulers

- TBB
 - Worker model
 - ~180 lines added, ~5 removed, ~70 modified (~1,500 / ~8,000 total)
- OpenMP
 - Team model
 - ~220 lines added, ~35 removed, ~150 modified (~1,000 / ~6,000 total)

Overheads?

- TBB
 - Example micro-benchmarks that Intel includes with releases

| | tree sum | preorder | fibonacci |
|---------------------|----------|----------|-----------|
| Lithe-Compliant TBB | 54.80 | 228.20 | 8.421 |
| TBB | 54.80 | 242.51 | 8.722 |

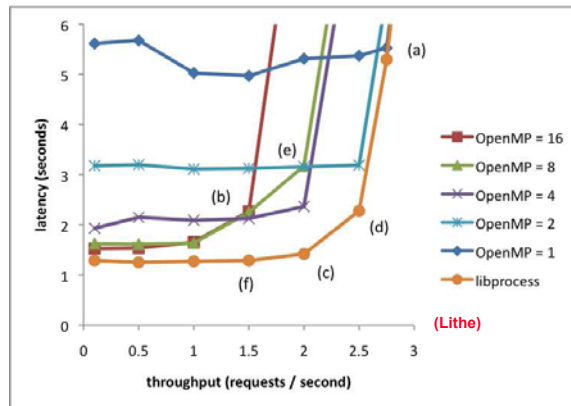
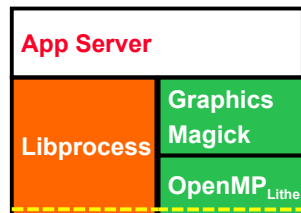
- OpenMP
 - NAS benchmarks (conjugate gradient, LU solver, and multigrid)

| | conjugate gradient (cg) | LU solver (lu) | multigrid (mg) |
|----------------------------|-------------------------|----------------|----------------|
| Lithe-Compliant GNU OpenMP | 57.06 | 122.15 | 9.23 |
| GNU OpenMP | 57.00 | 123.68 | 9.54 |

Flickr Application Server

- GraphicsMagick parallelized using OpenMP
- Server component parallelized using threads (or libprocess processes)
- Spectrum of possible implementations:
 - Process one image upload at a time, pass all resources to OpenMP (via GraphicsMagick)
 - + Easy implementation
 - Can't overlap communication with computation, some network links are slow, images are different sizes, diminishing returns on resize operations
 - Process as many images as possible at a time, run GraphicsMagick sequentially
 - + Also easy implementation
 - Really bad latency when low-load on server, 32 core machine overwhelmed
 - All points in between ...
 - + Account for changing load, different image sizes, different link bandwidth/latency
 - Hard to program

Flickr-Like App Server



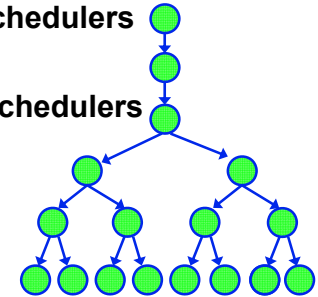
Tradeoff between throughput saturation point and latency.

Case Study: Sparse QR Factorization

Different matrix sizes

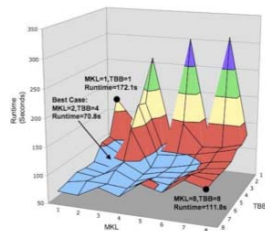
| | landmark | deltaX | ESOC | Rucci1 |
|----------|----------------|-------------------|------------------|------------------------------|
| Size | 71,952 x 2,704 | 68,600 x 21,961 | 327,062 x 37,830 | 1,977,885 x 109,900 |
| Nonzeros | 1,146,868 | 247,424 | 6,019,939 | 7,791,168 |
| Domain | surveying | computer graphics | orbit estimates | ill-conditioned least-square |

- deltaX creates ~30,000 OpenMP schedulers
- ...
- Rucci creates ~180,000 OpenMP schedulers
- Platform: Dual-socket 2.66 GHz Intel Xeon (Clovertown) with 4 cores per socket (8 total cores)



Case Study: Sparse QR Factorization

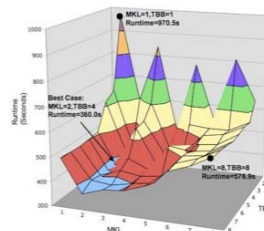
ESOC



| | |
|-----------------|-------|
| Tuned: | 70.8 |
| Out-of-the-box: | 111.8 |
| Sequential: | 172.1 |

Lithe: 66.7

Rucci

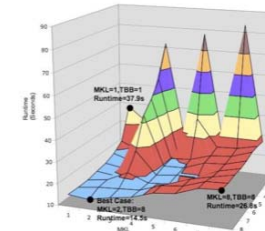


| | |
|-----------------|-------|
| Tuned: | 360.0 |
| Out-of-the-box: | 576.9 |
| Sequential: | 970.5 |

Lithe: 354.7

Case Study: Sparse QR Factorization

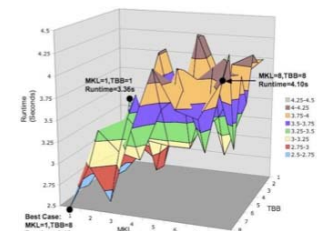
deltaX



| | |
|-----------------|------|
| Tuned: | 14.5 |
| Out-of-the-box: | 26.8 |
| Sequential: | 37.9 |

Lithe: 13.6

landmark



| | |
|-----------------|-----|
| Tuned: | 2.5 |
| Out-of-the-box: | 4.1 |
| Sequential: | 3.4 |

Lithe: 2.3

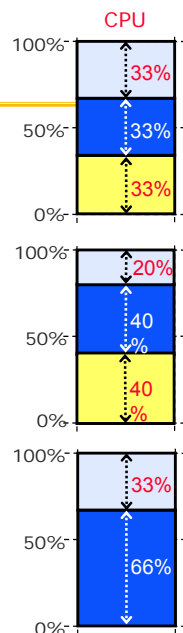
Is this a good paper?

- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?

Break

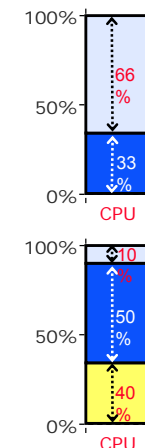
What is Fair Sharing?

- n users want to share a resource (e.g., CPU)
 - Solution:
Allocate each $1/n$ of the shared resource
- Generalized by *max-min fairness*
 - Handles if a user wants less than its fair share
 - E.g. user 1 wants no more than 20%
- Generalized by *weighted max-min fairness*
 - Give weights to users according to importance
 - User 1 gets weight 1, user 2 weight 2



Why is Fair Sharing Useful?

- *Weighted Fair Sharing / Proportional Shares*
 - User 1 gets weight 2, user 2 weight 1
- *Priorities*
 - Give user 1 weight 1000, user 2 weight 1
- *Reservations*
 - Ensure user 1 gets 10% of a resource
 - Give user 1 weight 10, sum weights ≤ 100
- *Isolation Policy*
 - Users cannot affect others beyond their fair share



Properties of Max-Min Fairness

- **Share guarantee**
 - Each user can get at least $1/n$ of the resource
 - But will get less if her demand is less
- **Strategy-proof**
 - Users are not better off by asking for more than they need
 - Users have no reason to lie
- Max-min fairness is the only “reasonable” mechanism with these two properties

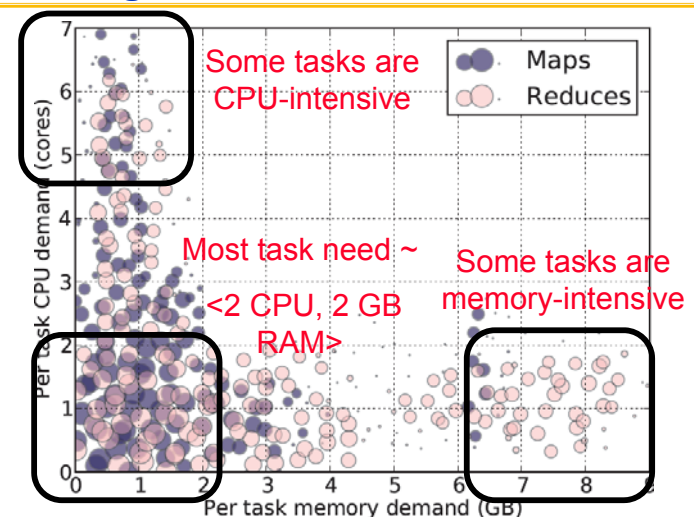
Why Care about Fairness?

- Desirable properties of max-min fairness
 - **Isolation policy:**
A user gets her fair share irrespective of the demands of other users
 - **Flexibility** separates mechanism from policy:
Proportional sharing, priority, reservation,...
- **Many schedulers** use max-min fairness
 - Datacenters: Hadoop’s fair sched, capacity, Quincy
 - OS: rr, prop sharing, lottery, linux cfs, ...
 - Networking: wfq, wf2q, sfq, drr, csfq, ...

When is Max-Min Fairness not Enough?

- Need to schedule *multiple, heterogeneous* resources
 - Example: Task scheduling in datacenters
 - » Tasks consume more than just CPU – CPU, memory, disk, and I/O
- What are today’s datacenter task demands?

Heterogeneous Resource Demands

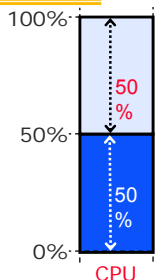


2000-node Hadoop Cluster at Facebook (Oct 2010)

Problem

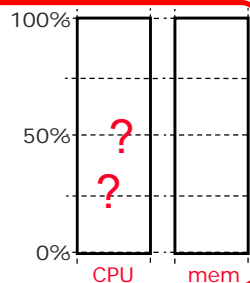
Single resource example

- 1 resource: CPU
- User 1 wants <1 CPU> per task
- User 2 wants <3 CPU> per task



Multi-resource example

- 2 resources: CPUs & memory
- User 1 wants <1 CPU, 4 GB> per task
- User 2 wants <3 CPU, 1 GB> per task
- What is a fair allocation?



3/7/2016

cs262a-S16 Lecture-14

37

Problem definition

How to **fairly** share **multiple resources** when users have **heterogeneous demands** on them?

3/7/2016

cs262a-S16 Lecture-14

38

Model

- Users have **tasks** according to a **demand vector**
 - e.g. <2, 3, 1> user's tasks need 2 R_1 , 3 R_2 , 1 R_3
 - Not needed in practice, can simply measure actual consumption
- Resources given in multiples of demand vectors
- Assume divisible resources

3/7/2016

cs262a-S16 Lecture-14

39

What is Fair?

- **Goal:** define a fair allocation of multiple cluster resources between multiple users
- **Example:** suppose we have:
 - 30 CPUs and 30 GB RAM
 - Two users with equal shares
 - User 1 needs <1 CPU, 1 GB RAM> per task
 - User 2 needs <1 CPU, 3 GB RAM> per task
- **What is a fair allocation?**

3/7/2016

cs262a-S16 Lecture-14

40

First Try: Asset Fairness

- **Asset Fairness**

- Equalize each user's *sum of resource shares*

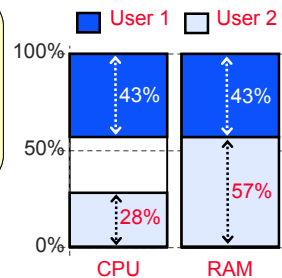
Problem

User 1 has < 50% of both CPUs and RAM

Better off in a separate cluster with 50% of the resources

- Asset fairness yields

- U_1 : 15 tasks: 30 CPUs, 30 GB ($\Sigma=60$)
- U_2 : 20 tasks: 20 CPUs, 40 GB ($\Sigma=60$)



Lessons from Asset Fairness

“You shouldn’t do worse than if you ran a smaller, private cluster equal in size to your fair share”

Thus, given N users, each user should get $\geq 1/N$ of her dominating resource (i.e., the resource that she consumes most of)

Desirable Fair Sharing Properties

- Many desirable properties

- Share Guarantee
- Strategy proofness
- Envy-freeness
- Pareto efficiency
- Single-resource fairness
- Bottleneck fairness
- Population monotonicity
- Resource monotonicity

DRF focuses on these properties

Cheating the Scheduler

- Some users will *game* the system to get more resources
- Real-life examples
 - A cloud provider had quotas on map and reduce slots
Some users found out that the map-quota was low
 - » Users implemented maps in the reduce slots!
 - A search company provided dedicated machines to users that could ensure certain level of utilization (e.g. 80%)
 - » Users used busy-loops to inflate utilization

Two Important Properties

- Strategy-proofness

- A user should not be able to increase her allocation by lying about her demand vector
- Intuition:
 - » Users are incentivized to make truthful resource requirements

- Envy-freeness

- No user would ever strictly prefer another user's lot in an allocation
- Intuition:
 - » Don't want to trade places with any other user

Challenge

- A fair sharing policy that provides
 - Strategy-proofness
 - Share guarantee
- Max-min fairness for a single resource had these properties
 - Generalize max-min fairness to multiple resources

Dominant Resource Fairness

- A user's *dominant resource* is the resource she has the biggest share of
 - Example:

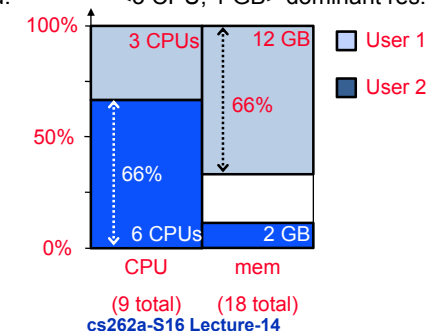
| | |
|----------------------|----------------|
| Total resources: | <10 CPU, 4 GB> |
| User 1's allocation: | <2 CPU, 1 GB> |

 Dominant resource is memory as $1/4 > 2/10$ ($1/5$)
- A user's *dominant share* is the fraction of the dominant resource she is allocated
 - User 1's dominant share is 25% ($1/4$)

Dominant Resource Fairness (2)

- Apply max-min fairness to dominant shares
- Equalize the dominant share of the users
 - Example:

| | |
|------------------|---------------------------------|
| Total resources: | <9 CPU, 18 GB> |
| User 1 demand: | <1 CPU, 4 GB> dominant res: mem |
| User 2 demand: | <3 CPU, 1 GB> dominant res: CPU |



DRF is Fair

- DRF is **strategy-proof**
- DRF satisfies the **share guarantee**
- DRF allocations are **envy-free**

See DRF paper for proofs

Online DRF Scheduler

Whenever there are available resources and tasks to run:

*Schedule a task to the user with smallest **dominant share***

- $O(\log n)$ time per decision using binary heaps
- Need to determine demand vectors

Alternative: Use an Economic Model

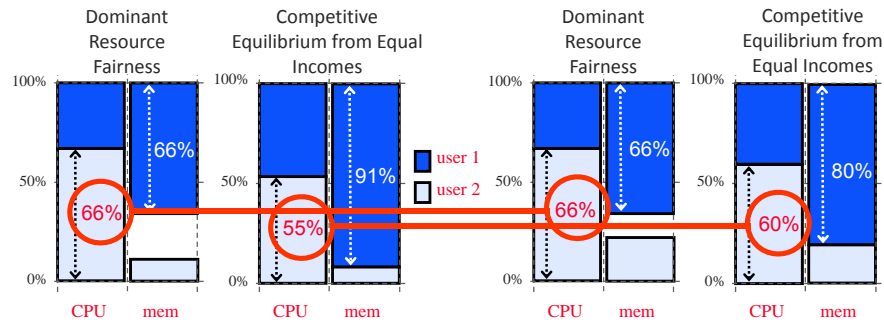
- Approach
 - Set **prices** for each good
 - Let users buy what they want
- How do we determine the right prices for different goods?
- Let the market determine the prices
- **Competitive Equilibrium from Equal Incomes (CEEI)**
 - Give each user $1/n$ of every resource
 - Let users trade in a perfectly competitive market
- Not strategy-proof!

Determining Demand Vectors

- They can be **measured**
 - Look at actual resource consumption of a user
- They can be **provided** the by user
 - What is done today
- In both cases, strategy-proofness incentivizes user to consume resources wisely

DRF vs CEEI

- User 1: <1 CPU, 4 GB> User 2: <3 CPU, 1 GB>
 - DRF more fair, CEEI better utilization



- User 1: <1 CPU, 4 GB> User 2: <3 CPU, 2 GB>
 - User 2 increased her share of both CPU and memory

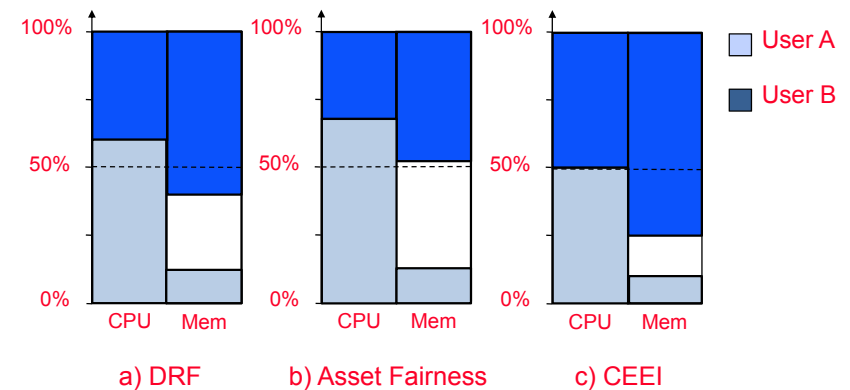
3/7/2016

cs262a-S16 Lecture-14

53

Example of DRF vs Asset vs CEEI

- Resources <1000 CPUs, 1000 GB>
- 2 users A: <2 CPU, 3 GB> and B: <5 CPU, 1 GB>



3/7/2016

cs262a-S16 Lecture-14

54

Desirable Fairness Properties (1)

- Recall *max/min fairness* from networking
 - Maximize the bandwidth of the minimum flow [Bert92]
- Progressive filling (PF) algorithm*
 1. Allocate ϵ to every flow until some link saturated
 2. Freeze allocation of all flows on saturated link and goto 1

3/7/2016

cs262a-S16 Lecture-14

55

Desirable Fairness Properties (2)

- P1. Pareto Efficiency*
 - » It should not be possible to allocate more resources to any user without hurting others
- P2. Single-resource fairness*
 - » If there is only one resource, it should be allocated according to max/min fairness
- P3. Bottleneck fairness*
 - » If all users want most of one resource(s), that resource should be shared according to max/min fairness

3/7/2016

cs262a-S16 Lecture-14

56

Desirable Fairness Properties (3)

- Assume *positive demands* ($D_{ij} > 0$ for all i and j)
- **DRF will allocate same dominant share to all users**
 - As soon as PF saturates a resource, allocation frozen

Desirable Fairness Properties (4)

- **P4. Population Monotonicity**
 - If a user leaves and relinquishes her resources, no other user's allocation should get hurt
 - Can happen each time a job finishes
- CEEI violates population monotonicity
- **DRF satisfies population monotonicity**
 - Assuming positive demands
 - Intuitively DRF gives the same dominant share to all users, so all users would be hurt contradicting Pareto efficiency

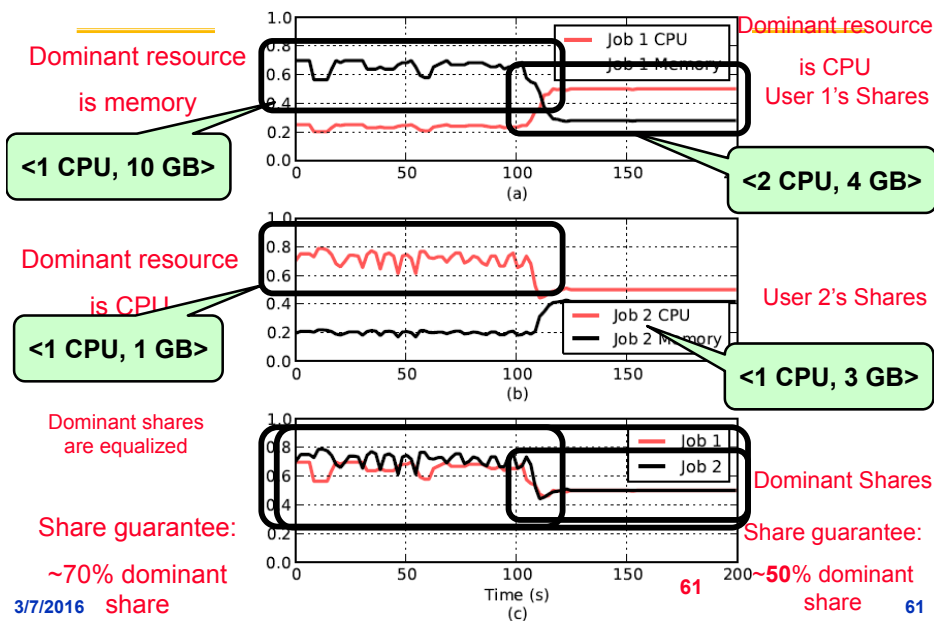
Properties of Policies

| Property | Asset | CEEI | DRF |
|--------------------------|-------|------|-----|
| Share guarantee | | ✓ | ✓ |
| Strategy-proofness | ✓ | | ✓ |
| Pareto efficiency | ✓ | ✓ | ✓ |
| Envy-freeness | ✓ | ✓ | ✓ |
| Single resource fairness | ✓ | ✓ | ✓ |
| Bottleneck res. fairness | | ✓ | ✓ |
| Population monotonicity | ✓ | | ✓ |
| Resource monotonicity | | | |

Evaluation Methodology

- Micro-experiments on EC2
 - Evaluate DRF's dynamic behavior when demands change
 - Compare DRF with current Hadoop scheduler
- Macro-benchmark through simulations
 - Simulate Facebook trace with DRF and current Hadoop scheduler

DRF Inside Mesos on EC2



Fairness in Today's Datacenters

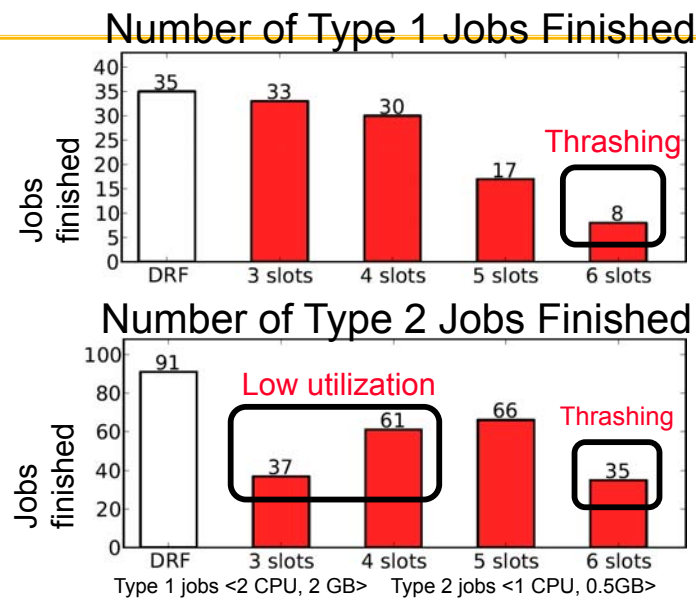
- Hadoop Fair Scheduler/capacity/Quincy
 - Each machine consists of k slots (e.g. $k=14$)
 - Run at most one task per slot
 - Give jobs "equal" number of slots, i.e., apply max-min fairness to slot-count
- This is what DRF paper compares against

3/7/2016

cs262a-S16 Lecture-14

62

Experiment: DRF vs Slots

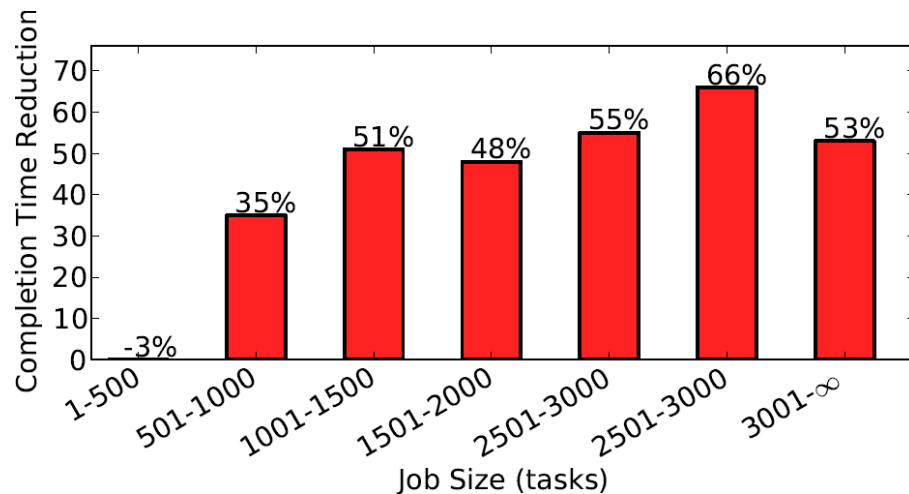


Experiment: DRF vs Slots



Reduction in Job Completion Time DRF vs Slots

- Simulation of 1-week Facebook traces



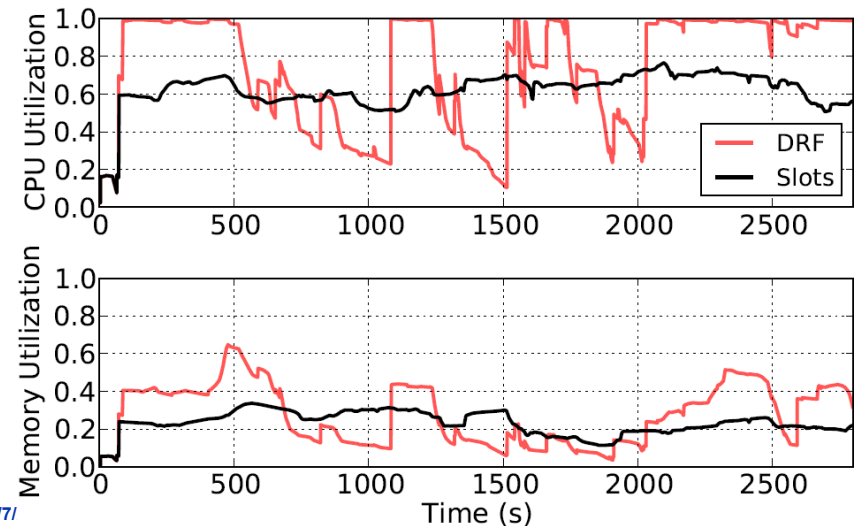
3/7/2016

cs262a-S16 Lecture-14

65

Utilization of DRF vs Slots

- Simulation of Facebook workload



3/7/

66

Summary

- DRF provides *multiple-resource fairness* in the presence of *heterogeneous demand*
 - First generalization of max-min fairness to multiple-resources
- DRF's properties
 - Share guarantee*, at least $1/n$ of one resource
 - Strategy-proofness*, lying can only hurt you
 - Performs better than current approaches

3/7/2016

cs262a-S16 Lecture-14

67

Is this a good paper?

- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?

3/7/2016

cs262a-S16 Lecture-14

68