Compiler Enhanced Scheduling for Single ISA Heterogeneous Architecture

Presented by: Jyothi Krishna V S

IIT Madras

September 25, 2015

Overview

- Asymmetric Multicore Processors : Evolution
- Scheduling in Asymmetric Multicore Processor
- *big.LITTLE*
- Scheduling in *big.LITTLE*
- OpenMP
- Preliminary Work
- Future Work & Summary

Heterogeneous Multiprocessor : Evolution

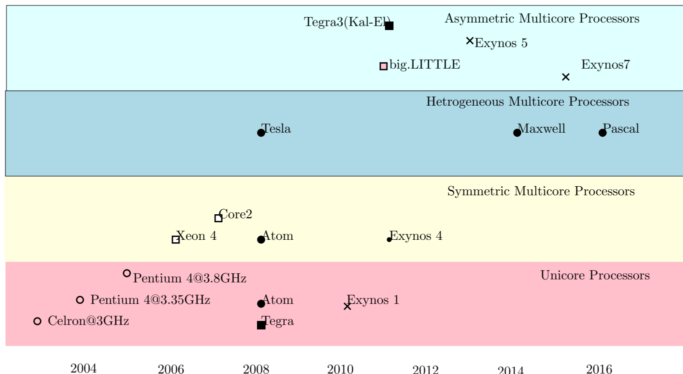
Definition (Moore's Law)

The number of transistors incorporated in a chip will approximately double every 24 months

Heterogeneous Multiprocessor : Evolution

Definition (Moore's Law)

The number of transistors incorporated in a chip will approximately double every 24 months



Heterogeneous Multiprocessors & Asymmetric Multicore Processors

- **Heterogeneous Multiprocessor** : More than one processing element(PE) combined.
  - Multiple ISA :
    - Predefined division of tasks to cores.
    - Have multiple versions machine code for different kind of cores.
  - Single ISA : Asymmetric Multicore Processors(AMP)

Heterogeneous Multiprocessors & Asymmetric Multicore Processors

- **Heterogeneous Multiprocessor** : More than one processing element(PE) combined.
  - Multiple ISA :
    - Predefined division of tasks to cores.
    - Have multiple versions machine code for different kind of cores.
  - Single ISA : Asymmetric Multicore Processors(AMP)
- **Asymmetric Multicore Processor**
  - Combination of power efficient *weak* cores and complex, powerful cores.
  - The weaker cores will much lesser dynamic power & Leakage Current.
  - Schedule less compute intensive threads to weaker core.

Scheduling in AMP

- AMPS (Asymmetric Multi-Processor Scheduler) [1]
  - Asymmetry-aware load balancing :
    - Scaled power P = S(Scale) * F (Frequency).
    - Load, L = load/P | $L_{max} - L_{min} \leq 1$
  - Faster-core-first scheduling & NUMA-aware migration.

2. Bias Scheduling in Heterogeneous Multi-core Architectures by D Koufaty et al. in EuroSys '10

1. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures by T Li et al. in SC '07.

Scheduling in AMP

- AMPS (Asymmetric Multi-Processor Scheduler)[1]
  - Asymmetry-aware load balancing :
    - Scaled power P = S(Scale) * F (Frequency).
    - Load, L = load/P | $L_{max} - L_{min} \leq 1$
  - Faster-core-first scheduling & NUMA-aware migration.
- Bias Scheduling[2]
  - Application Bias : Core type best suited for thread.
  - Balanced and Imbalanced system
  - Imbalanced system : Migrate Thread from busiest to idlest core.
  - Balanced system : Thread swap if the bias is not honored.

---

2. Bias Scheduling in Heterogeneous Multi-core Architectures by D Koufaty et al. in EuroSys '10

1. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures by T Li et al. in SC '07.

Scheduling in AMP

- PIE (Performance Impact Estimation [3])
  - Collects CPI stack, MLP and ILP.
  - Estimate the performance in other core.
  - Simplified system assumption : identical cache hierarchy and branch prediction on both core types.
  - No Power Modelling

3. Scheduling Heterogeneous Multi-Cores through Performance Impact Estimation (PIE) by K V Craeynest et al., ISCA '12
4. Power-Performance Modeling on Asymmetric Multi-Cores by M Pricopi et al., CASES '13

Scheduling in AMP

- PIE (Performance Impact Estimation [3])
  - Collects CPI stack, MLP and ILP.
  - Estimate the performance in other core.
  - Simplified system assumption : identical cache hierarchy and branch prediction on both core types.
  - No Power Modelling
- CPI Based Power-Performance Modelling [4]
  - Data analysis : to estimate data dependency, structural hazards.
  - CPI = $CPI_{steady} + CPI_{miss}$
  - CPI_{steady} : cycles spent in pipeline stages, data dependency, structural hazards
  - CPI_{miss} : Cache misses and their latencies.
  - Formulate & Train based on training Data.
  - Power modelling for power constraint environment.

3. Scheduling Heterogeneous Multi-Cores through Performance Impact Estimation (PIE) by K V Craeynest et al., ISCA '12
4. Power-Performance Modeling on Asymmetric Multi-Cores by M Pricopi et al., CASES '13

Program feature based Scheduling

Critical Section :

- Accelerating Critical Section Execution [5] :
  - Serialization due to Critical Sections : threads waiting to enter critical sections.
  - Leverage big cores to Critical Sections.
  - Compiler transformation CSCALL and CSRET : to execute CS on a big processor.
  - For critical section intensive workloads.

5. Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures by A M Suleman et. al., ASPLOS '09
6. Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures by K Chronaki et al., ICS '15

Program feature based Scheduling

Critical Section :

- Accelerating Critical Section Execution [5] :
  - Serialization due to Critical Sections : threads waiting to enter critical sections.
  - Leverage big cores to Critical Sections.
  - Compiler transformation CSCALL and CSRET : to execute CS on a big processor.
  - For critical section intensive workloads.
- Criticality Aware Scheduling for OmpSs : [6]
  - OmpSs : Task based parallel programming model. TDGs.
  - The criticality dynamically calculated : No of threads that are waiting for task.
  - Each task is categorized into critical and non-critical.
  - Critical-ready Queue : executed by big cores.

---

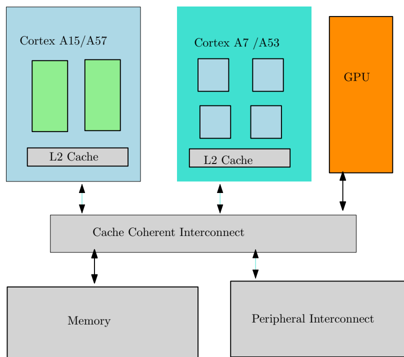
5. Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures by A M Suleman et. al., ASPLOS '09
6. Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures by K Chronaki et al., ICS '15

Scheduling in AMP

- Approximation Aware Scheduling [7] :
  - For soft real-time application.
  - Meeting Thermal Design Power (TDP) constraint.
  - Cluster Architecture
  - Maximizing QoS(Quality of Service).
  - Offline Scheduler : EDF(Early deadline First) -> Relax (TDP) -> Improve (QoS) -> ADAPT(DVFS).
  - Online Scheduler : Shutdown unused cluster, Predictive policy to improve future jobs.

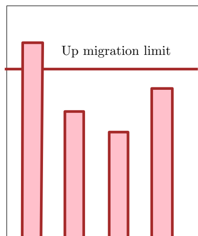
7. Approximation-Aware Scheduling on Heterogeneous Multi-core Architectures by C Tan et al. ASP-DAC '15

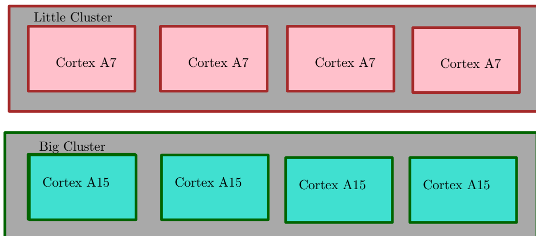
big.LITTLE

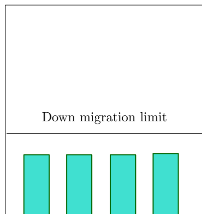


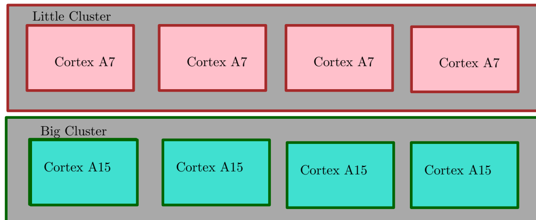
- Asymmetric Multicore Architecture from ARM.
- Big cores : Cortex A15 / A57.
- Little cores : Cortex A7 / A53.
- Software : Global Task Scheduling.
- Big-little migration : less time than a DVFS state transition.

big.LITTLE Scheduling : Cluster Switching

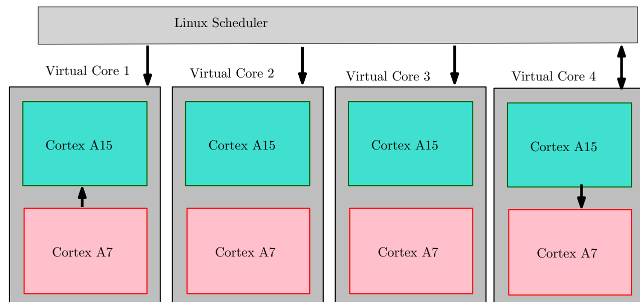


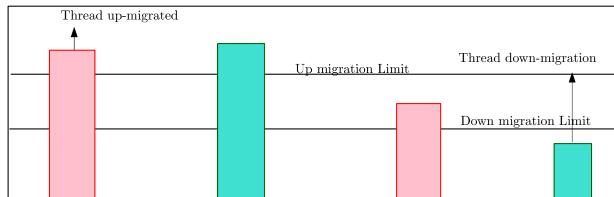




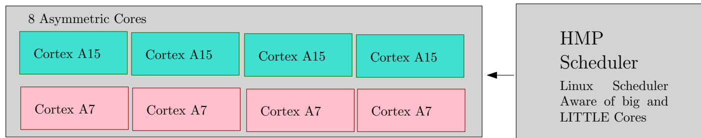


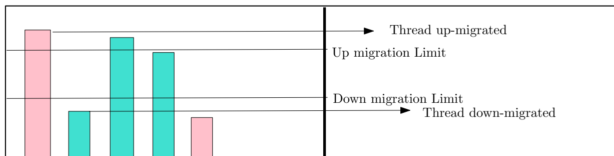
big.LITTLE Scheduling : In-Kernel Switching





big.LITTLE Scheduling : HMP Scheduling





OpenMP API

- Popular Shared Memory Parallelism.
- Team of threads executing parallel regions.
- Barrier synchronized.
- Work sharing construct : for, single, sections.
- Data sharing attributes.

Preliminary Work

- Setup Odroid-XU3 installed with Exynos-5 with Lubuntu.
- HMP scheduling enable/disable switch.
- Exposed up and down migration as system calls.
- Intel OpenMP runtime.
- Uses IMOP [9] frame work to
  - Combine two parallel regions.
  - Compare the number of memory operations performed and computations performed.
  - Extract parallel for and sections from parallel regions.
  - Splitting parallel regions.

9. IIT Madras OpenMP Framework by Aman Nougrahia et al.

Scheduling Sequential Regions

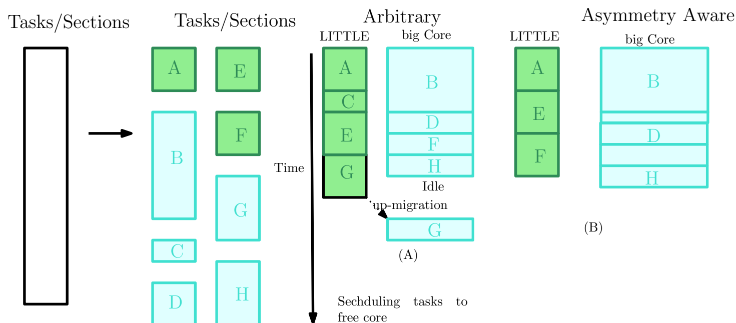
- OpenMP Program => Sequential => Parallel => Sequential ...
- Sequential region executed by master Thread.
- Master thread scheduled in little core :(
- Non-master thread down-migrated during sequential region.
- Schedule sequential region in thread scheduled in big Core.
- Analysis : Identifying the thread to run sequential region.
- Transformation : Absorb the sequential region inside parallel.
- *single* construct in OpenMP.

Code Transformation

Sequential Workload Transformation	

#pragma omp parallel
{
  S1;
}
S2;
#pragma omp parallel
{
  S3;
}
	#pragma omp parallel
	{
	  S1;
	  #pragma omp single if (omp_thread_num()==threadid)
	  {
	    S2;
	  }
	  S3;
	}

Scheduling Tasks on Threads



- Task / Sections : independent *chunks* executed in parallel.
- AMP : Arbitrary Scheduling would be inefficient.
- Analysis : Execution time for each task on different cores.
- Transformation : Each task as an iteration in for loop.

Scheduling Tasks : Code Transformation

```
#pragma omp parallel
{
  #pragma omp sections
  {
    Section_A; // Each a section
    Section_B;
    Section_C;
    Section_D;
    Section_E;
    Section_F;
    Section_G;
    Section_H;

  }
}
```

```
#pragma omp parallel
{
#pragma omp for
  for(__i =0;i<10;i++){
    if(__i = 0) Section_A;
    if(__i = 1) Section_E;
    if(__i = 2) Section_F;
    if(__i = 5) Section_B;
    if(__i = 6) Section_C;
    if(__i = 7) Section_D;
    if(__i = 8) Section_G;
    if(__i = 9) Section_H;
    }
}
```

FIGURE – Transformation for scheduling sections in openMP. Hardware config a little and one big core. On compiler analysis we predict B,C,D sections in big core and A & E in little core. The thread 0 is predicted to be in little core and thread 1 in big core.

Scheduling Equal Workload Threads

- Equal Workload : All threads executing same code.
- Efficient in SMP.
- Barrier synchronization.
- AMP : Threads scheduled in big cores finish first.
- Forced migration/switching to reduce the gap.
- Normalized Switching & On Barrier switching.
- Analysis : Find optimal switching point based on Data dependencies and Types of operations.
- Transformation : Adding SYSCALLS for migration.

Scheduling Equal Workload Threads

- Equal Workload : All threads executing same code.
- Efficient in SMP.
- Barrier synchronization.
- AMP : Threads scheduled in big cores finish first.
- Forced migration/switching to reduce the gap.
- Normalized Switching & On Barrier switching.
- Analysis : Find optimal switching point based on Data dependencies and Types of operations.
- Transformation : Adding SYSCALLS for migration.

Benchmark	4 Little cores	With BS	With NS	With BS & NS	4 Big Cores
ep S	2.19	1.62	1.54	1.52	1.06
ep A	34.61	26.03	24.49	24.87	16.8
ep B	138.2	102.4	96.4	97.1	70.7
ep C	552.3	409.6	397.26	397.79	319.9
ep W	4.35	3.04	3.07	3	2.5

FIGURE – Experiment setup : 2 big cores 2 little cores. No. of threads 4. BS : switching on barrier hit. NS : Switching at a point where the ratio of big and little is maintained.

Code Transformation

```
#pragma omp parallel
{
   S1;
   Sr;
   //-->Optimal point
   //   for up migration
   .....
   St;
   //--> Optimal point
   //    for up migration
   ...
   Sn;
}
```

```
#pragma omp parallel
{
   int __threadmigrated = 0;
   S1;
   Sr;
   if(__inlittleCore()){
    __do_upmigration;
    __threadmigrated = 1;
   }
   St;
   if(__inbigCore()){
     if(__threadmigrated == 0)
    __do_downmigration;
   }
   Sn;
}
```

FIGURE – Transformation for equal workload threads. __threadmigrated is to prevent frequent migration of same thread.

Scheduling Iterative Workload

- Static scheduling : very good in SMP.
- AMP : threads in big core finishes iterations quickly.
- Dynamic scheduling : Costly.
- Analysis : Finding execution cost for each iteration in little and big.
- Also find ratio of workload in each iteration.
- Transformation : A dynamic work-list for each thread based on ratio XX.
- Also code for stealing & updating XX.

Scheduling Iterative Workloads : Code Transformation

Iterative Workload Transformation

Initial Loop

```
#pragma omp parallel
{
 #pragma omp for
 for(i=0;i<N;i++){
    S1;
 }
}
```

```
#pragma omp parallel
{
  int __ratio = XX;
   #pragma omp single
  {
      __init(&__iter,__ratio,N);
  }
  do{
    __foo_for(omp_get_thread_num(),&__iter)
    if(__finished(&__iter){
      break;
    }else{
      __steal(&__iter, &__ratio,omp_get_thread_num());
    }
  }while(1);
}
...

void _foo_for(int * _iter[][],int tid)
{
  for(i = iter[tid][0];i<iter[tid][1];i++){
    S1;
  }
}
```

FIGURE – Transformations for an Iterative workloads. The functions _init, _finished and _steal and array _iter will be common for a single program. The ratio XX will be specific for each loop, which denote how much faster the big core is when compared to little core while executing one iteration in the loop.

```
//Helper Functions
int __iter[NO_OF_THREADS][2];
...
void _init(int _iter[][],_int ratio, int N)
{
  double part = N * (XX* (NO_OF_BIG_CORES) + NO_OF_LITTLE_CORES);
  int t = 0;
  for(i=0;i<NO_OF_THREADS;i++){
    iter[i][0] = t;
    if(big_core(i))
       t = t + (XX * part);
    else
       t = t + part;
   iter[i][1] = t-1;
  }
}
int _finished(int* _iter[][])
{
  int t = 0;
  for(i=0;i<NO_OF_THREADS;i++){
    if(iter[i][0] == iter[i][1])
      t++;
  }
  if(t == NO_OF_THREADS)
    return 1;
  else return 0;
}
void _steal(int* _iter[][],int *_ratio)
{
  //  update_XX();
  // steal from little cores first.

}
```

FIGURE – Common Functions for Iterative workload Trasformation.

Future Work & Summary

- AMP : Scheduling techniques can improve performance and power consumption.
- Majority rely on dynamic scheduling techniques.
- HMP scheduling in big.LITTLE.
- Compiler transformation compatible with HMP.
- Compiler Analysis and Transformation to optimize scheduling.
- Parallelizing Graph Algorithms.
- User-directed migrations : up and down migrations as SYSCALLS.
- Handling power critical workloads.

Future Work & Summary

- AMP : Scheduling techniques can improve performance and power consumption.
- Majority rely on dynamic scheduling techniques.
- HMP scheduling in big.LITTLE.
- Compiler transformation compatible with HMP.
- Compiler Analysis and Transformation to optimize scheduling.
- Parallelizing Graph Algorithms.
- User-directed migrations : up and down migrations as SYSCALLS.
- Handling power critical workloads.

THANK YOU