Compiler Enhanced Scheduling for OpenMP for Heterogeneous Multiprocessors

Jyothi Krishna V S
IIT Madras
jkrishna@cse.iitm.ac.in

Shankar Balachandran
IIT Madras
shankar@cse.iitm.ac.in

ABSTRACT

Scheduling in Asymmetric Multicore Processors (AMP), a special case of Heterogeneous Multiprocessors, is a widely studied topic. The scheduling techniques which are mostly runtime do not usually consider parallel programming pattern used in parallel programming frameworks like OpenMP. On the other hand, current compilers for these parallel programming platforms are hardware oblivious which prevent any compile time optimization for platforms like big.LITTLE and has to completely rely on runtime optimization. In this paper we propose a hardware aware Compiler Enhanced Scheduling (CES) where common compiler transformations are coupled with compiler added scheduling commands to take advantage of the hardware asymmetry and improve the runtime efficiency. We implement a compiler for OpenMP and demonstrate its efficiency in Samsung Exynos with big.LITTLE architecture. On an average, we see 18% reduction in runtime and 14% reduction in energy consumption in standard NPB and FSU benchmarks with CES across multiple frequencies and core configurations in big.LITTLE.

Keywords

Heterogeneous Computing, OpenMP, big.LITTLE

1. INTRODUCTION

Heterogeneous multiprocessors use more than one type of processing elements (PEs) to build complex systems, typically hoping to conserve energy. Asymmetric Multi Processors (AMPs) are a special case where the PEs have the same Instruction Set Architecture (ISA) but may have different clock speeds, cache sizes or micro architecture. ARM's *big.LITTLE* and NVIDIA's Kal-El are examples of AMP.

In AMP, PEs can be logically classified into memory centric and computation centric. The memory centric cores have lower cache latency, but lower computation power when compared to compute centric cores. Consequently for different threads, based on the number of memory operations and computations performed, the performance can vary across these types of processors. Thus various scheduling patterns of threads can yield different performance and power readings for the same set of inputs. Scheduling in AMP is a widely studied [13, 16, 20, 7, 12, 18] area. Different scheduling mechanisms were proposed to exploit the hardware heterogeneity for optimal performance and/or power.

Table 1: big and LITTLE cores : Comparison

	LITTLE core	**big core**
Core Types	Cortex-A7	Cortex-A15
Pipeline	simple 8-stage in-order	out-of-order, multi-issue
Frequency	600 - 1300 MHz	800 - 1900 MHz
Speed	1.9 DMIPS [4]	3.5-4.01 DMIPS
Instruction Set	Thumb-2	

big.LITTLE is an AMP from ARM targeting the mobile industry. The powerful cores are known as big core and the weaker power-efficient cores are known as LITTLE cores. A comparison of big and LITTLE cores in the system we use is given in Table 1. Numerous scheduling algorithms have been suggested for big.LITTLE to extract the best out of both set of cores. The most popular and widely accepted one is the Heterogeneous Multiprocessing (HMP) [9] scheduling which is integrated into fair scheduling policy in Linux [1] kernel.

In HMP scheduling implemented in big.LITTLE, the scheduler uses big cores for compute intensive tasks and LITTLE cores for less compute intensive (or memory intensive) tasks. Based on the recent *cpu utilization* by each thread, the threads are up-migrated (from LITTLE to big) or down-migrated (from big to LITTLE).

In this work, we introduce a compiler enabled scheduling (CES) framework for multithreaded programs, through program transformations to improve on the execution time and energy consumption. Compiler analysis and transformations typically accomplish optimizations by efficiently estimating the run-time behaviour [11, 17, 14]. We chose OpenMP, as it is a widely used parallel programming platform. Many of the parallel programming design patterns are easily realizable in OpenMP.

The rest of paper is organized as follows. Section 2 will give a brief introduction to OpenMP. Section 3 explains our proposed transformation. Section 4 provides the implementation details and Section 5 provides the results obtained. Section 6 lists out the related works and Section 7 provides the conclusion.

2. OpenMP API

In this section we provide an introduction to OpenMP. OpenMP API is designed for shared memory parallelism in

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

C, C++ and FORTRAN. We focus on C and C++ programs. Special directives (*#pragma*) are used by the programmer to specify OpenMP program behaviour.

A parallel region, written inside *#pragma omp parallel* (omp_parallel), creates a team of threads, which may run in parallel to execute the code defined. The number of threads (`N_THREADS`) in a team can be set using environment variables or using OpenMP library calls. For our experiments and transformations, we assume the number of threads is equal to the number of available cores, unless specified inside the input program.

The master thread is a special thread which creates the team on encountering the omp_parallel region. It has id 0 and the rest of the threads, referred to as non-master threads, have ids from 1 to `N_THREADS - 1`. Barriers (*#pragma omp barrier*) are used to synchronize among the threads in a team. Threads waits at the barrier until rest of the threads in the team reach the barrier. There is implicit barrier at the end of parallel regions as well.

Work-sharing constructs define units of work, each of which is executed exactly once by one of the threads in the team. The work sharing constructs have an implicit barrier at the end, which can be removed using *nowait* clause. There are three worksharing constructs available in C/C++. i) omp_for (*#pragma omp for*), implements a parallel for loop in `N_ITRS` iterations, where each iterations (`N_ITRS`) are executed once by one of the threads. The iterations can be executed in parallel with other iterations. The scheduling pattern of iterations to threads may be specified using static, dynamic or guided clauses. In static scheduling each thread is given equal-sized chunks of iterations in a round-robin fashion, until there are no iterations left. The size of a chunk unless specified, will be taken as (`N_ITRS/N_THREADS`). In dynamic scheduling, each thread is supplied with a chunk of iterations on demand by thread, until there are no iterations left. Guided scheduling is very much similar to dynamic scheduling except that the chunk size (whose intial value can be specified using th eoptional parameter) +starts off large and decreases in later steps to handle the load imbalance better. Guided and dynamic scheduling are used for handling load imbalance between iterations, but has higher overhead compared to static scheduling. In ii)omp_sections (*#pragma omp sections*) we have multiple sections each encapsulated by *#pragma omp section* (single_omp_section), executed by one of the threads in the team. The scheduling is arbitrary. iii) omp_single contain a block that is executed by only one of the threads.

3. CES

In this section we explain our compiler enhanced scheduling (CES) framework.

The OpenMP programming model design works well for Symmetric Multicore Processor (SMP) environment. However, the vast difference in computing power between big and LITTLE cores can result in large difference in execution time among the threads leading to hardware under-utilization. The HMP scheduler is aimed at reducing this gap, but it works solely based on history, leaving the scheduling oblivious of future workload. The hardware-asymmetry aware CES compiler is aimed at reducing this gap. CES optimizes the parallel execution time by reducing the disparity in individual thread running time. Optimizations can also be drafted to reduce overall system power consumption.

An omp_parallel region in the input OpenMP code is divided into separate *parallel-segment*s. A parallel-segment can be (i)one of the OpenMP worksharing constructs or (ii)other blocks inside omp_parallelregions bounded by barriers. Each parallel-segment is analyzed and optimized separately. The execution time of a parallel-segment will be determined by the longest running thread in the team.

For effective compile-time scheduling, we need to estimate the runtime behaviour of a parallel-segment in each core. We have a workload model to estimate the performance of each thread in a core. Using the estimate we transform the code to normalize the execution time for each thread.

3.1 Workload Modelling

The Performance Estimation Mathematical (PEM) model would give an estimate of the performance of a thread in big and LITTLE cores. PEM use an adapted version of [14] for multithreaded programs. Whilst execution, the CPU cycles could be spent in executing ALU or memory operations, or on branch mis-predictions. The estimated performance P, of a core can be denoted as:

$$P(ct) = P_{OP}(ct) + P_{MEM}(ct)$$

where ct can be either big or LITTLE. The $P_{OP}(ct)$ component will estimate the performance of each core for ALU operations from the number of arithmetic, floating point and bit-wise operations performed in the code segment. $P_{MEM}(ct)$ will give an estimate of the time spent in memory operations and branches based on their count. The ratio of $P_{OP}(big)$ to $P_{OP}(LITTLE)$ will be very small since big cores will execute the ALU operations much faster than LITTLE cores for same code. Whereas the ratio of $P_{MEM}(big)$ to $P_{MEM}(LITTLE)$ will be much higher. The $P(ct)$ is takena s the workload for thread i, $wl(i)$.

When there are multiple paths (due to branches), we take the largest cost from all paths as the workload (wl) of that thread. We collect the amount of memory, ALU and branch operations that might be performed through simple compiler passes. Based on these metrics the performance of a thread in a particular core can be estimated [14]. The big and LITTLE cores are profiled extensively to identify the effect of each hardware operation (memory, ALU, branching) during execution. The estimate of recursions and non-deterministic loops (while and do-while) are treated as *unknown* with high costs.

Once the PEM is modelled, a thread is then scheduled into a core that suites its workload with an intention to minimize the workload imbalance among threads inside a parallel-segment.

$$\min_{i,j \in team} wl_{im} = |wl(i) - wl(j)|$$

Here wl_{im} denotes the workload imbalance in the system.

In CES, an initial thread-to-core scheduling is fixed with thread i scheduled to core i. This can be changed if better (lower wl_{im}) scheduling pattern can be identified at either compile time or runtime.

Now we explain compiler transformations for each type of parallel-segment.

3.2 Scheduling omp_for

One of the most important parallel-segments which affects the performance is omp_for. The current scheduling policies for the omp_for are static, dynamic and guided. The static scheduling in OpenMP, is designed to work well for balanced

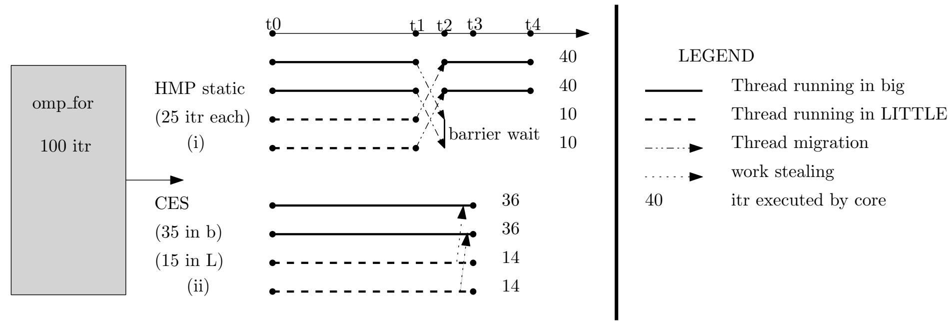

Figure 1: Figure comparing execution of omp_for. The system demonstrated contains 2 big and 2 LITTLE cores. Numbers are symbolic.

A — Finding victim thread to steal

```
1  int getthread(){
2    i = wl_with_max_itrs_left();
3    if(end[i]-itr[i] > chunk){
4  #pragma omp atomic write
5      status[i] = SHARED;
6      return i ;
7    }
8    return N_THREADS;
9  }
```

Figure 2: Selection of victim thread for stealing

load, in a SMP. In big.LITTLE, there is an intrinsic imbalance in hardware, leaving static scheduling a non-viable option. Both dynamic and guided scheduling have huge overhead associated (roughly 5 times that of static scheduling [5]) making us look for more efficient options.

Figure 1 shows the inefficiency of static scheduling in HMP. Consider an omp_for with `N_ITRS`=100 executed with 4 parallel threads, on a system with 2 big cores and 2 LITTLE cores. The threads are allocated 25 iterations each. In HMP, once the threads that are initially scheduled in big cores finish their work (time t1), they are migrated to LITTLE while threads that are scheduled in LITTLE are migrated to big (t2). Assume the LITTLE cores have executed 10 iterations by time t1. The remaining 15 iterations in those threads will be executed by big cores (t2 to t4). Meanwhile the LITTLE cores do no meaningful work (the threads are waiting on barrier), leading to their underutilization, while big cores are overworked. With CES, we have a more balanced division of iterations based on the power of each core and the workload in each iteration. This increases effective parallelism, thereby reducing the execution time. To eliminate any imbalance in the initial division (due to inefficient modelling of runtime behaviour), CES also has a work-stealing framework built into it. A thread that has finished all its work can *steal* work from other threads having pending work. The LITTLE cores consume lesser energy compared to big cores to execute the same code. In CES, LITTLE cores take up relatively more work than in HMP, thereby consuming lesser energy.

CES transforms omp_for to use a worklist based approach. Each thread has a private worklist allocated with unequal chunk of iterations, such that the wl_{im} is minimal. Once a thread finishes all iterations in its workload, it first identifies the thread with maximum iterations left (getthread function shown in Figure 2) as the victim thread for stealing. When a victim thread is selected for stealing, its worklist ceases being private and all operations done on the worklist should be protected, by taking a `lock` on the worklist. `lock_update(var,val,lock)` (B.6 and B.16 in Figure 3) acquires `lock` and proceed to write `val` to `var` and returns the old value in `var`. The data structure `status` is maintained, to denote current state of the worklist. The value `SHARED` (set when worklist is selected for stealing, line A.5 in Figure 2) denotes shared state of the worklist and in other two stages (`PRIVATE` and `INITIAL_PRIVATE`) the worklist is considered to be private.

B — Worklist stealing-code

```
1   int doitr(int t){
2    if ( itr[ t ] < end[ t ]){
3     if(status[t] != SHARED)
4      return itr[ t ] ++;
5     else
6      return lock_update(itr[t],itr[t]+1,lock[t]);
7    }else {
8   #pragma omp atomic write
9      status[t] = PRIVATE;
10    int f = 1;
11    do{
12     int i = getthread ();
13     if(i == N_THREADS) return -1;
14     int newend = end[i] - chunk;
15     int end = end[i];
16     oe= lock_update(end[i],newend,lock[i]);
17     if (oe == end ){
18      end[ t ] = end ;
19      return itr[ t ]++;
20     }else f = 0;
21    } while (!f);
22   }
23   return -1;
24  }
```

Figure 3: Do iteration code with stealing.

C	Initial Loop	D	After
		1	`#pragma omp parallel`
		2	`{`
1	`#pragma omp parallel`	3	`...`
2	`{`	4	`initialize();`
3	`...`	5	`#pragma omp barrier`
4	`#pragma omp for`	6	`do{`
5	`for(i=0;i<N;i++){`	7	`i = doitr(tid);`
6	`S(i);`	8	`S(i);`
7	`}`	9	`}while(i != -1)`
8	`...`	10	`#pragma omp barrier`
9	`}`	11	`update_scaledend_i();`
		12	`...`
		13	`}`

Figure 4: Transformations for omp_for.

The proposed transformation is shown in Figure 4. The `initialize` function initializes `end` and `itr` based on `scaledend_i` and also other variables used for transformation. The `doitr` function (shown in Figure 3), returns the next iterations to be performed by the thread. If the worklist

is empty it proceeds to stealing (lines B.8-24). It returns -1, when there are no iterations available for the thread, on which the thread will stop working and start waiting at the barrier (line D.10) for other threads.

The variables used in the transformation are (i) `end`: this array keeps the last iteration in the worklist of the thread. ii) `itr`: This array keeps track of the current iteration that is being executed by each thread. iii) `scaledend_i`: This array is used to set the initial division of iterations. It will contain a scaled end point and will need to be multiplied by `N_ITRS` for actual end points. If the calculated ratio depends on a variable whose value is known only at runtime, the `scaledend_i` will be replaced by an equation which will be evaluated at runtime. iv) `chunk`: This value decides the chunk_size of iterations for stealing based on stealing cost and iteration cost. During execution, if only one team of threads is live (no nested parallelism) at a given point of time, all data structures can be made global (except `scaledend_i` created for i^{th} omp_for).

If the omp_for has a chance of being re-executed, we update the initial division for iterations based on the current execution (the number of iterations executed by each thread), such that we will have a more balanced division of wl on re-entry (D.11).

If an iteration has very little work (around 2-3 instructions), the stealing cost would dominate the amount of work stolen thereby reducing the advantage we get from stealing. We could increase the value of `chunk` stolen by the stealer thread at a time, but increasing the stealing chunk-size beyond a point would remove the balancing effect of the stealing process. We discard the stealing logic for such loops and proceed the execution with the initial division of worklist. We categorize these loops as *fixed size loop*. This static division could lead to slight increase in execution time and/or energy consumption, if the division is not balanced.

3.3 Scheduling omp_sections

In omp_sections, the default scheduling is arbitrary, which could result in inefficient scheduling and unnecessary migrations on big.LITTLE. A big single_omp_section might be scheduled in LITTLE core while big cores are scheduled with smaller workload. As a result the HMP scheduler will later introduce thread migrations.

In CES the sheduling of omp_sections is divided into two stages. In the *affinity-allocation* stage, suitability of each core for a single_omp_section is determined using the type of operations performed in it (termed as affinity). Based on the affinity, the single_omp_section is allocated to the core in big or LITTLE set with least workload.

Once the affinity allocation is complete, CES proceeds to *normalization stage*, where we try to balance the wl in each core. From the core with highest estimated wl, a single_omp_section which shows least affinity towards it is selected as the target-section. The scheduler then allots the target-section to the core with least wl. If the new allocation configuration has a lower wl_{im}, we select the new allocation, else we go ahead with the current allocation.

To implement this preferred scheduling the omp_sections is transformed into an omp_for with static scheduling, with the `N_ITRS` set to maximum single_omp_section allocated to a thread multiplied by `N_THREADS`. Each iteration would be given a single_omp_section or nothing. If the parallel-segment is omp_single, then it is taken as a special case of omp_sections with just one section for analysis.

Thread switching transformation			
E	**Before**	F	**After**
		1	`#pragma omp parallel`
		2	`{`
		3	`Si; i=0;`
		4	`if(inLITTLE()){`
		5	`while(i<N_THREADS){`
1	`#pragma omp parallel`	6	`if(mg[i]>=pt`
2	`{`	7	`&&inbig(i))`
3	`Si;`	8	`migrate(tid,i);`
4	`//mp`	9	`break;`
5	`Sj;`	10	`}`
6	`// mgp`	11	`i++;`
7	`Sk;`	12	`}`
8	`}`	13	`Sj;`
		14	`if(inbig(tid)){`
		15	`#pragma omp atomic`
		16	`mg[tid] = i;`
		17	`}`
		18	`Sn;`
		19	`}`

Figure 5: Transformation for thread switching. inLITTLE() and inbig() is to check core running current thread.

3.4 Thread Migration

For the rest of the parallel-segment the amount of code executed by individual threads in team will not change due to scheduling. To reduce the wl_{im} in these regions we introduce *thread-exchange* between big and LITTLE cores to reduce the imbalance.

We identify *migration-points* (denoted as `mp` in line E.4, Figure 5) in code, as points with the lowest thread exchange cost (c_{ex}). The c_{ex} is calculated using the amount of live variables at that point, and big.LITTLE thread migration cost. On reaching the migration-point, the thread currently allocated in LITTLE core (*attacker* thread) will identify a thread in big core which has passed the *minimum-guarantee* point (denoted as `mgp` in line E.6) in its execution as the *victim-thread*. The minimum-guarantee point is selected such that the wl_{LITTLE} for remaining code of the victim thread is not larger than wl_{big} for the remaining code of the attacker. Now if the estimated reduction in wl_{im} is larger than the c_{ex} the thread-exchange is made. When the ratio of LITTLE and big cores available is more than one we might require more than one migration-point. Each migration-point will have a paired minimum-guarantee point. In the figure `pt`(line F.6) represents the migration point id, and the victim-thread should have reached the corresponding minimum guarantee-point for the migration to happen (line F.7).

4. IMPLEMENTATION

We use IMOP [15], a source to source OpenMP compiler. The transformed code will also be in OpenMP and can be compiled using compilers like gcc. The compiler framework is divided into two phases, the analysis phase and transformation phase. Pre-processing includes removal of macros, function in-lining whenever possible and also MHP analysis for division of parallel-segment. The analysis phase involves generating program metrics, and balancing the wl on each thread. The transformation phase implements the transformations based on the suggestions made during the analysis phase.

5. RESULTS

Table 2: Excution time and Energy consumption for different configurations of big and little. Time in seconds and Energy in Joules. 4 b 4 L stands for 4 big and 4 LITTLE cores. The LITTLE are ran at 1.3 GHz while big are ran at 1.9GHz.

BM	4 b 4 L				2 b 4 L				2 b 2 L			
	hmp		ces		hmp		ces		hmp		ces	
	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy
`EP A`	15.05	58.06	13.07	55.09	22.06	51.68	19.09	49.09	29.05	61.64	19.66	56.06
`EP B`	58.17	221.31	51.19	207.03	86.02	203.65	75.23	186.19	114.80	244.14	77.08	225.40
`CG`	5.07	43.18	3.16	29.11	6.06	36.61	4.09	24.24	5.57	32.61	4.8	26.77
`IS`	1.62	6.53	1.60	6.44	1.44	4.52	1.42	4.51	1.4	4.02	1.38	4.09
`sec`	58.52	272.68	59.78	151.9	96.7	236.85	59.8	165.65	117.05	291.37	76.25	253.50

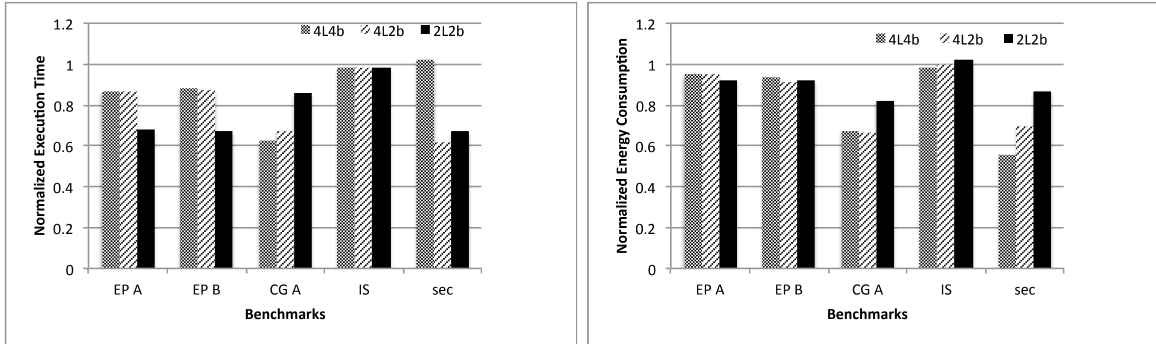

Figure 6: Normalized execution time and energy consumption running LITTLE at 1.3 GHz and big at 1.9GHz.4L4b stands for 4 LITTLE cores and 4 bigcores.

Results of our transformations on standard NPB benchmarks [6] are explained here. We have set up a Linux environment (Ubuntu 14.04) enabled with HMP scheduling in Odroid-XU3 board [2] installed with Samsung Exynos 5422 [3]. Exynos 5422 implements *big.LITTLE* technology with 4 little and 4 big cores.

Table 2 shows execution time and overall system energy consumption of each benchmark. Readings shown here are the average of 5 runs with LITTLE cores running at 1.3Ghz and big cores running at 1.9GHz. We have tried our compiler in three different hardware combinations : (i) 2 LITTLE and 2 big cores (ii) 4 LITTLE and 2 big core (iii) 4 LITTLE and 4 big cores. Figure 6 shows normalized execution time and energy consumption for the different hardware combinations. It shows on an average 18% reduction in execution time and 14% reduction in energy consumption.

`EP` has a single omp_for which has been transformed using a worklist with stealing. `EP` has been studied in two sizes (`A` & `B`) to find the effect of work-stealing on performance. For both sizes we get on an average, 20% improvement in runtime and 8% improvement in energy consumption showing the stealing algorithm scales well with size. Benchmarks `CG` and `IS` are made of small omp_for (one to two instructions). These are transformed to simple static worklists. Changing ratio of big and LITTLE cores available have low effect on performance improvement with the CES able to adapt to the changes in hardware configurations.

We have also taken a modified version of multitask benchmark from FSU-OpenMP benchmark (denoted by `sec` in the tables). The original program had two single_omp_section (one computing prime and another sine) which was copied 4 times (since maximum `N_THREADS` = 8) to make the modified benchmark. Prime single_omp_section showed more affinity towards big. In HMP, arbitrary scheduling of sections caused variations in execution time based on the initial scheduling, which was more evident when `N_THREADS` was less than the number of single_omp_section. By obliging to section *affinity* of single_omp_section we are able to gain huge energy benefits with little effect on the execution.

We have also tested effectiveness of CES, for different frequencies for big.LITTLE. Figure 7 shows normalized execution time and energy consumption for two frequency configurations. Readings of $f1$ are obtained while running big at 1.9GHz and LITTLE at 1.3GHz while that of $f2$ are obtained while running big at 1.9GHz and LITTLE at 1GHz. For $f1$ on an average, we get 16% improvement in execution time and 23% reduction in energy consumption. For $f2$ we get 11% reduction in runtime and 24% reduction in energy consumption. Energy gain for `IS` in $f2$ is higher (28% compared to 3% in $f1$) since the workload division obtained was more balanced when compared to what we got with $f1$.

6. RELATED WORK

Most of the scheduling techniques proposed are run-time in nature [20, 13, 8, 7]. In PIE (Performance Impact Estimation) [20], hardware counters for CPI, ILP and MLP are used to estimate the performance that can be achieved, if the thread was to run in other type of core at run-time. The scheduling is then decided based on the estimate. Asymmetric Multi Processor Scheduling (AMPS) [13] is another run time scheduler which quantifies the computing power of each core based on the weakest core (scaled power). The scheduling policy is structured to balance the workload based on each core's scaled power. Profiling is used to find out the affinity of each thread efficiently to schedule the threads to core which shows [12] maximum affinity and improve the efficiency of the system. The works above do not consider the nature and type of multithreaded programs that are run, and the run-time methods proposed are based on the current state of the system.

Scheduling algorithms described in [10, 8, 18] that find

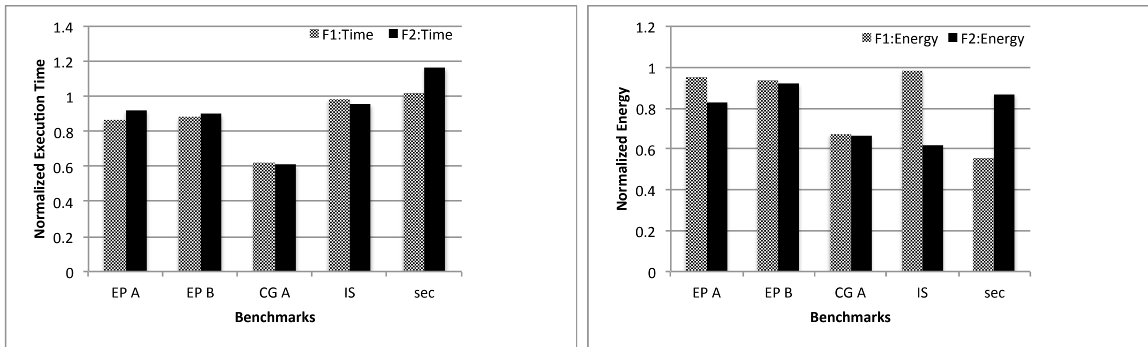

Figure 7: Normalized execution time and energy consumption for different configurations. For f1 we ran LITTLE at 1.3GHz and for f2 LITTLE are ran at 1GHZ. big were ran at 1.9GHz for both. System with 4 LITTLE and 4big cores.

threads and program sections that are critical in nature, and schedule those in big cores. In [8] the critical sections are identified at run-time, by prioritizing those tasks that form part of the longest path in dynamic task dependency graph. There are also scheduling algorithms which are aimed at real time applications in a QoS (Quality of Service) - performance trade off [19]. These works show that knowledge about nature of workload can help in optimizing the performance. Compiler analysis and transformations are heavily relied upon, in these works, for optimization.

7. CONCLUSION

This paper explain a hardware aware compiler for OpenMP in big.LITTLE. The compiler directed scheduling can help balancing the future workload in each thread to reduce the runtime and power consumption in runtime. The present day hardware-unaware compiler assumes SMP and the scheduling leads to unfair division of workload. Our method shows great promise with around 18% improvement in execution time on an average 14% improvement in energy consumption. Our future focus involves providing a runtime framework to handle hardware heterogeneity, finding the best hardware configuration for big.LITTLE in terms of power and runtime.

8. ACKNOWLEDGEMENT

Thanks to Raghesh A and Indu K for for lending their time for constant review, suggestions and comments on this work.

9. REFERENCES

[1] CFS Scheduler. Technical report, linux.

[2] Odroid-XU3. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127.

[3] Exynos 5 Octa. http://www.samsung.com/global/business/semiconductor/minisite/Exynos/w/solution.html#?v=octa_5422, 2013.

[4] R P Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *Commun. ACM*, 27(10):1013–1030, October 1984.

[5] J. M. Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. In *EWOMP*, pages 99–105, 1999.

[6] Bailey et al. The NAS Parallel Benchmarks:Summary and Preliminary Results. In *SC*, pages 158–165, New York, NY, USA, 1991. ACM.

[7] Becchi et al. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *CF*, pages 29–40. ACM, 2006.

[8] Chronaki et al. Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures. In *ICS*. ACM, 2015.

[9] Chung et al. Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM big.LITTLE Technology. "http://www.arm.com/files/pdf/Heterogeneous_Multi_Processing_Solution_of_Exynos_5_Octa_with_ARM_bigLITTLE_Technology.pdf", 2013.

[10] Du Bois et al. Criticality Stacks: Identifying Critical Threads in Parallel Programs Using Synchronization Behavior. In *ISCA*, New York, NY, USA, 2013. ACM.

[11] Kadayif et al. Compiler-directed High-level Energy Estimation and Optimization. *ACM Trans. Embed. Comput. Syst.*, 4(4):819–850, November 2005.

[12] Koufaty et al. Bias Scheduling in Heterogeneous Multi-core Architectures. In *EuroSys*, pages 125–138. ACM, 2010.

[13] Li et al. Efficient Operating System Scheduling for Performance-asymmetric Multi-core Architectures. In *SC*, pages 53:1–53:11. ACM, 2007.

[14] McKinley et al. Improving Data Locality with Loop Transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, July 1996.

[15] Nougrahia et al. IIT Madras OpenMP(IMOP) Framework. http://www.cse.iitm.ac.in/~amannoug/imop.

[16] Pricopi et al. Power-performance Modeling on Asymmetric Multi-cores. In *CASES*, pages 15:1–15:10, 2013.

[17] Shih et al. Compiler Optimization for Reducing Leakage Power in Multithread BSP Programs. *ACM Trans. Des. Autom. Electron. Syst.*, 20(1):9:1–9:34, November 2014.

[18] Suleman et al. Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures. In *ASPLOS*, 2009.

[19] Tan et al. Approximation-aware scheduling on heterogeneous multi-core architectures. In *ASP-DAC 2015*, 2015.

[20] Van Craeynest et al. Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE). In *ISCA*, pages 213–224, 2012.