Hindawi
Mobile Information Systems
Volume 2017, Article ID 3021565, 8 pages
https://doi.org/10.1155/2017/3021565

Research Article

Efficient Thread Mapping for Heterogeneous Multicore IoT Systems

Thomas Mezmur Birhanu,[1] Zhetao Li,[2] Hiroo Sekiya,[3] Nobuyoshi Komuro,[3] and Young-June Choi[1]

[1]*Department of Computer Engineering, Ajou University, Suwon 16499, Republic of Korea*
[2]*College of Information Engineering, Xiangtan University, Hunan 411105, China*
[3]*Graduate School of Advanced Integration Science, Chiba University, Chiba 263-8522, Japan*

Correspondence should be addressed to Young-June Choi; choiyj@ajou.ac.kr

Received 17 November 2016; Accepted 16 January 2017; Published 27 February 2017

Academic Editor: Jeongyeup Paek

This paper proposes a thread scheduling mechanism primed for heterogeneously configured multicore systems. Our approach considers CPU utilization for mapping running threads with the appropriate core that can potentially deliver the actual needed capacity. The paper also introduces a mapping algorithm that is able to map threads to cores in an $O(N \log M)$ time complexity, where N is the number of cores and M is the number of types of cores. In addition to that we also introduced a method of profiling heterogeneous architectures based on the discrepancy between the performances of individual cores. Our heterogeneity aware scheduler was able to speed up processing by 52.62% and save power by 2.22% as compared to the CFS scheduler that is a default in Linux systems.

1. Introduction

The multicore architecture has penetrated into the industry because of its apparent advantage in aiding for higher performance. And more recently, a heterogeneous variant of this architecture has also found its way into the mainstream ARM's big.LITTLE [1] for various Internet-of-Things (IoT) applications as well as mobile devices. The popularity of big.LITTLE architectures is found in their better performance-power ratio and more efficient space utilization than their homogeneous counterparts [2]. These architectures consist of cores having different computing capacities but under the same instruction set architecture (ISA).

They act as a way to conform to the variation witnessed in a typical workload sets, thereby helping maintain performance while saving power through aversion of core underutilization. Other than underutilization, harboring a fast thread on sluggish core would be penalized by a significant power consumption since it would take a long time for it to complete, as seen in [3]. The variation between core performance and energy footprint is the result of discrepancies between their features like cache size, frequency, issue width, in-order/out-order, and such [4–9]. Though this multicore architectures seem to be quite advantageous, designing an app littered to a multicore, let alone heterogeneous, architecture is made difficult because the developer is usually exposed to the low level details. Paper [10], proposes a developer friendly interface to harness GPUs and other multicore architectures. As such, we believe much of the logic for harnessing these architectures should be pushed to the OS scheduling. In [11], they even went as far as proposing their own experimental heterogeneous programming system.

These heterogeneous architectures are a new trend; as such they can render previously existent schedulers obsolete. The Completely Fair Scheduler (CFS) in Linux is one of them. CFS was introduced starting form Linux 2.6.23 as a replacement of the previous vanilla scheduler's SCHED_OTHER interactivity code [12]. CFS is widely chosen against other schedulers such as FIFO, in that it averts the problem of process starvation and assures fairness. CFS does not account

core heterogeneity while making its scheduling decision. This can bear a far reaching consequence in terms of the energy and performance that can potentially be saved and improved, respectively. As CFS is the default scheduler in Linux, our focus will be towards its improvements in the arena of heterogeneous architectures.

This paper proposes Fastest-Thread-Fastest-Core (FTFC) dynamic thread scheduling mechanism that bases its mapping decision on conformity of running threads CPU utilization with the performance of available cores. It does this by dynamically and periodically measuring the CPU utilization of running threads so those threads that exhibit high CPU utilization are assigned to cores that can deliver high performance needed, while those threads with low CPU utilization are assigned to low performance cores. We measure the threads CPU utilization based on their CPU time; that is, the amount of time that thread spends executing on the core, as opposed to, for example, waiting for input/output (I/O) operations or entering low-power (idle) mode. The FTFC scheduler does its mapping using a new scheme which we coined as Binary Searched Mapping (BSM), which runs in $O(N \log M)$ time complexity, where N is the number of cores and M is the number of type of cores.

This paper also introduces a new means of modeling heterogeneity based on the discrepancy between the performances of cores. We coined it as heterogeneity measure (HM) that describes how heterogeneous the system is. Using this measurement technique we are able to consider a whole spectrum of heterogeneous configurations. Through our experimentation, the FTFC scheduler saves power by up to 2.22% and speeds up processing by up to 52.62%, compared to CFS.

The rest of the paper is organized as follows. Section 2 elaborates on related works. Section 3 presents our heterogeneity measure. Section 4 describes the proposed scheduler. Section 5 shows the experimental environment and results. Finally, Section 6 concludes.

2. Related Work

Regarding multicore scheduling, several studies are found in literature. In [13], Koufaty et al. proposed a method of looking at the CPI stacks of the running applications to infer the core to which they would belong to. The HASS technique, proposed by Shelepov et al. [14], sought for efficient mapping by using program signatures which are embedded in the program binaries prior to execution. In [15], Chen and John used Weighted Euclidean Distance (WED) measure between the desired resource demand of a program and the core configurations to map a thread to an optimal core. Both techniques rely on prior offline profiling of the programs.

Most previous work relied on analysis of programs into their constitute subtasks, formulating constraint graph, and partitioning the graph to minimize the overall execution time [16] or the data transfer on buses [16, 17]. Their approach requires the microarchitecture dependent performance/power data as a priori information for scheduling, whereas our approach does not need this. Chen and John [18] used fuzzy-logic based program-core suitability computation. This method is inapplicable when the number of core characteristics increases. This is because the complexity of their algorithm increases exponentially as the characteristics increases. Our approach is totally oblivious to the number or type of characteristics, as it only concerns the core performance.

Gulati et al. [19] explored the benefits exhibited from dynamic mapping of tasks to cores of flexible design. Kumar et al. [2] proposed a dynamic program scheduling scheme based on sampled EDP in subsequent runs. This is just a straightforward trial-and-error scheduling approach to find the appropriate match of core and thread. Becchi and Crowley [20] followed up on their work by using IPC ratios between two programs for program migrations. All these methods, as do ours, exploit intraprogram diversity and could adapt to program phase changes. In [21], a dynamic scheduling was proposed by tweaking core priority so as the OS load balancer does its job in a biased manner. This bias is formulated by taking identify key metrics.

The work in [22] proposes a static heuristic method for heterogeneous processors based on the use of an acyclic precedence graph. The use of this heuristic, however, requires a priori knowledge of the characteristics of the workloads. One of relatively recent papers that seek to address this mapping problem is investigated by Liu et al. [23]. They mathematically formulated the problem as a 0-1 integer linear program (Integer LP) and also proposed an iterative $O(N^2/M)$ heuristic algorithm for solving it. We were able to do it in an $O(N \log M)$ fashion.

We assume that each core is executing a single thread at most and assumed that the variations of the workloads rather than the type of workloads are more important when choosing the type of benchmarks. In [24], they refer to the variation of workload as more important metric when they designed their flexible multicore (FMC) architecture. Reference [25] advocates individual cores properties to be tailored to the different subsets of application characteristics as a main idea behind their designs.

3. Heterogeneity Measure

With the introduction of heterogeneity to architectures, the need to not only tell the heterogeneity but also the degree of the heterogeneity itself became apparent. We formulated it as the standard deviation between the normalized performances of individual cores. The performance of cores is once again modeled by total CPU utilization gotten from running four of PARSEC benchmarks serially on that core, core performance being directly proportional to the average CPU utilization. We define heterogeneity measure as

$$\mathrm{HM} = \sqrt{\sum_{i=1}^{N} \left(\frac{T_i - T_{\mathrm{avg}}}{T_{\max}} \right)^2}, \tag{1}$$

where T_i is the ith core CPU performance. From the equation above we can deduce that if we had a heterogeneous dual core configuration and the variance of 0.5 ($\mathrm{HM} = 1/\sqrt{2}$), it

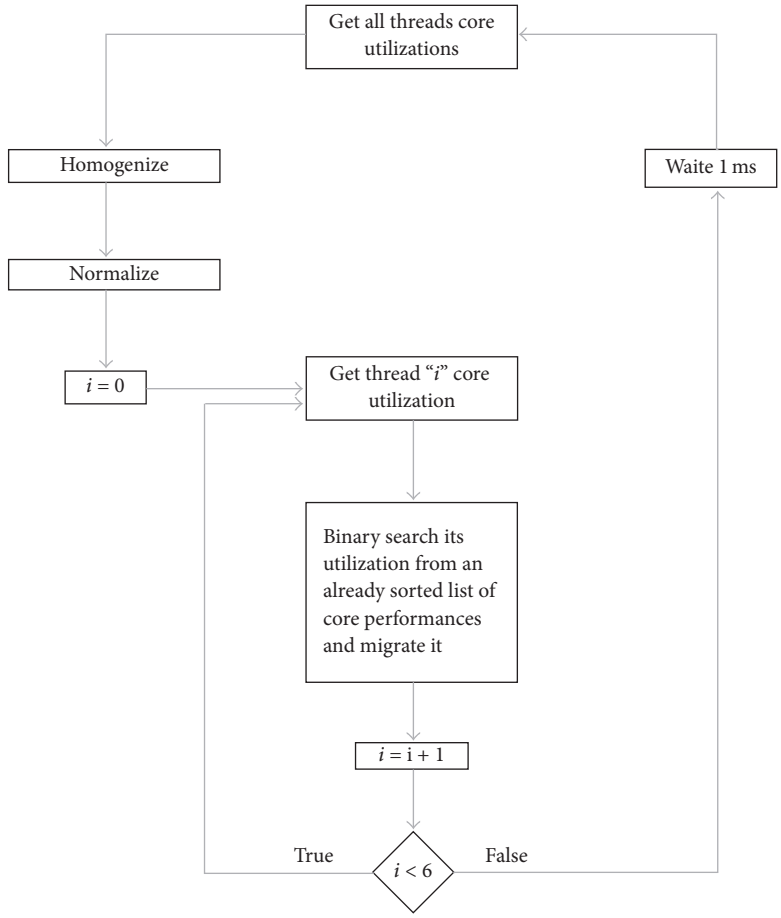

Figure 1: Flow diagram of FTFC scheduler.

means one core is twice as fast as the other, where T_{max} is one of either T_1 or T_2. HM can never be greater than or equal to 1. If it is 1, it means only one core is functioning. We use HM as an important metric to express heterogeneity of the configurations we used throughout the experimentation.

4. Fastest-Thread-Fastest-Core Scheduler

Unlike heuristic mapping methods proposed in [23], our method does not need to model the power, as it bases its methods based on the understanding of power consumption being dependent upon the independent variable performance. As such, we focus on performance optimization and let the dependent variable power follow up with its own optimization.

Before the FTFC scheduler takes over, six of the PARSEC benchmarks are launched and randomly assigned to one of the six available cores. Stage 1 of Figure 2 shows an example how it is done using four cores and four threads. T signifies the normalized CPU performance. U signifies the throughput of the threads on their respective core. The lifetimes of these threads are assigned arbitrary, the maximum being 30 seconds and the minimum being 5 seconds. Each thread was assigned the same priority as the other. The scheduling process ends as all threads end their execution. CPU utilization was monitored by measuring the CPU time. FTFC is the same as CFS except that FTFC considers heterogeneity and does dynamic thread migrations. When the configuration is homogeneous, that is, HM = 0, FTFC is essentially CFS, as there will not be any thread migrations. In such ways, FTFC can be seen as an extension of CFS.

Once all the threads are executed, the FTFC scheduler periodically monitors the CPU utilizations of the running threads. We set the time interval to be 1 ms. For such granularity, previous work [26] has shown cold cache effect and context migration costs can be brought to minimum. Some of the considerations that should be taken when doing migrations between cores with private L1s and L2s and shared L3 are, remembering the predictor state, allowing L2-L2 data transfers from inactive to active core and also maintaining the tags of the various L2 cache coherent. Transferring between registers and flushing a dirty private L1 data have a negligible performance overhead [26]. The FTFC scheduler can endure at most N migrations, where N is the number of cores. The diagrammatic illustration of FTFC scheduler is shown on Figure 1.

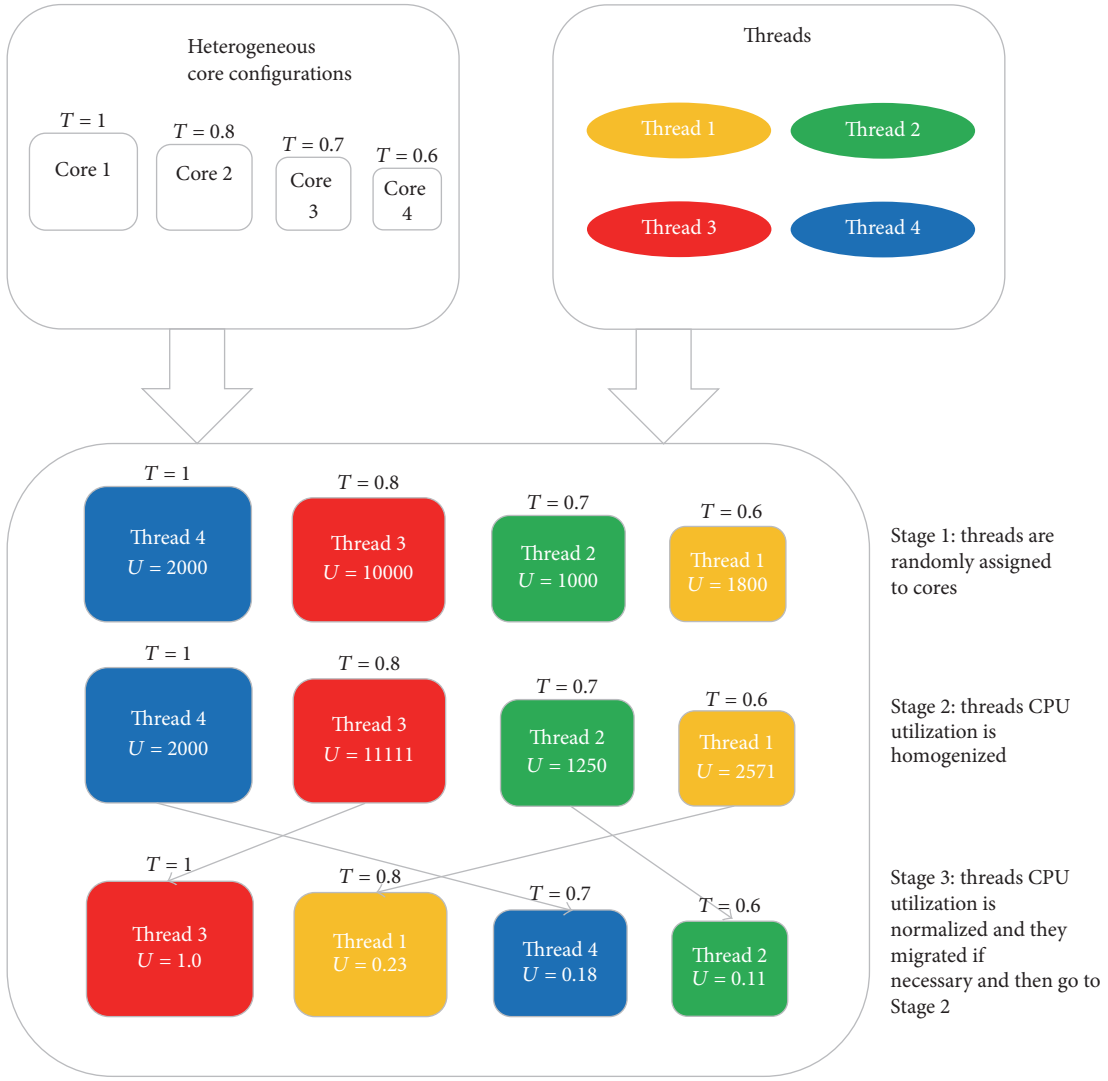

Figure 2: Example of FTFC scheduler.

4.1. Homogenization. Once the CPU utilization of a thread is gotten, it is homogenized. This process is necessary as it helps avoiding unfair comparison of CPU utilizations between threads running on small cores and those on bigger cores. "Small" and "Big" describe the relative processing power of the cores. By homogenizing we get the core-neutral CPU utilization value of the threads. It is done by multiplying the CPU utilization of a thread with a ratio of the normalized performance of the fastest core with the core it is running on ($T_{\max}/T_i = 1/T_i$). The homogenizing process is expressed as

$$U_i = \frac{U_i}{T_i}, \tag{2}$$

where T_i is the normalized ith core performance and U_i is the CPU utilization of ith thread. Stage 2 of Figure 2 shows the homogenizing process. We can see that thread 1 has same core utilization as thread 4 on stage 1. After the threads are homogenized thread 4 has lower utilization per core than thread 1. This means thread 4 was using less of the performance that core 1 could potentially deliver. The homogenizing process is needed to avoid static comparison of utilization values between cores, which is shown in the scenario of thread 1 and thread 4. The homogenization gives fairness in comparison of threads utilization according to the relative performance of cores.

4.2. Normalization. The next step is to normalize the homogenized CPU utilization of the running sets of threads. By normalizing we know exactly which thread is the most CPU utilizing among the whole sets of threads, because we already homogenized it.

$$U_i = \frac{U_i}{U_{\max}}, \tag{3}$$

where U_i is the ith threads CPU utilization and $U_{\max}$ is the maximum CPU utilization among the threads.

(1) //Individual core types performance is already gotten and listed in ascending order
(2) Let W_x be the xth core performance
(3) Let P_j be the jth thread in thread queue ($j = 1 \cdots L$)
(4) Let C_i be the processor of core type i. ($i = 1 \cdots M$)
(5) for j ($1 \cdots L$)
(6) T_j = jth thread CPU utilization
(7) left = 0
(8) right = number of type of cores
(9) while right ! = left
(10) mid = (right + left)/2
(11) if $W_{\text{mid}} > T_p$
(12) right = mid
(13) else
(14) left = mid
(15) end if
(16) end while
(17) place thread T_j on core C_{mid}
(18) end for

Algorithm 1: BSM algorithm.

Using this, we map the relatively high utilizing thread to the relatively high performance core, and the relatively low utilizing thread to the relatively low performance core. Using such relativity technique we ensure a complete abstraction of the hardware details of the core and program characteristics of a thread into a single variable of CPU performance and utilization value, respectively. In contrast to Performance Impact Estimation (PIE) proposed in [27], FTFC simplifies this estimation greatly. The relativistic nature of this mapping also ensures the avoidance of underutilization in a better way as compared to, for example, Faster-Core-First Scheduling as seen in [28], which aggressively assigns threads to the fastest core. If the core to be migrated upon is empty, the thread is simply migrated but if not we do the migration based on the to be migrated thread's utilization that is closest to CPU performance as compared to that of the already resident thread's.

Stage 3 of Figure 2 shows the normalizing process. Here, the thread utilization is normalized primarily to binary search it through an already sorted list of normalized core performances.

4.3. Binary Searched Mapping. After the homogenizing and normalizing phases, the mapping phase takes over. Here, we propose to use binary searching mapping (BSM) to reduce time complexity and computation overhead. The mapping of threads is performed in a binary search manner and the already sorted list of CPU performance is searched for the CPU utilization value of the subject thread. This would ensure that the final entry at which searching terminates is the closest CPU performance to the threads utilization in question. This is done from lines 6 to 15 of Algorithm 1. This searching runs at $O(\log M)$, where M is the number of unique cores, and, in the worst case, that is, if all the cores except the last one of the core types are occupied, it takes N times of iteration to reach it. BSM ensures a complexity of $O(N \log M)$ where M is the number of distinct core types in the system and N is the number of threads to be mapped.

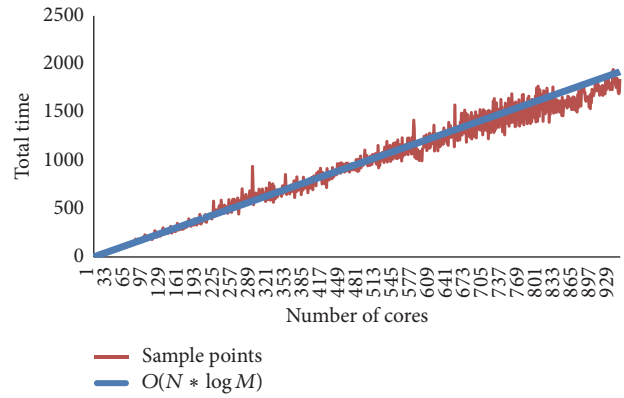

Figure 3: BSM time complexity.

Table 1: Thread mapping time complexity.

Algorithm	Time complexity
Binary searched mapping (BSM)	$O(N \log M)$
Weighted Euclidean distance (WED)	$O(NM)$
Maximize then swap (MTS)	$O(N^2/M)$
Naive	$O(N!)$

Though the scope of this paper is limited to heterogeneous multicore architectures in mobile devices, for the purpose of validating our mathematically gotten time complexity, we created dummy thread performance values and mapped them to a set of dummy cores but with the same HM always. We generated a set of cores ranging in numbers from 2 to 1000 and observed the trend of the postulated time complexity to the actual observed time. The results of this experiment can be seen in Figure 3. These findings can also suggest the significance of the BSM algorithm in an expanded scope of heterogeneous multicore processors as a whole.

As can be seen in Table 1, the BSM algorithm with the FTFC scheduler outperforms the others. The WED algorithm requires the offline profiling of applications based on their core complementary characteristics, which is another encumbrance to the compiling process and a loom on its scalability. The naive approach to this mapping problem runs at $O(N!)$ and efficient mapping is guaranteed (mapping wise), but it is too computationally expensive to be used, especially in resource constrained mobile devices. MTS runs at $O(N^2/M)$. The main drawback of the MTS heuristic approach is that it does not completely exhaust all possible swaps before it terminates. Its swaps are defined and prioritized by the ratio of the change in power to the change in throughput that the swap could produce. This leaves out combinations of swaps that could be possibly valuable, thus not always resulting in optimized mapping.

Table 2: Heterogeneous core configurations.

Heterogeneity	Core	Issue width	L2 cache	Branch predict size	ROB size
0	0	4	512 KB	16384	256
	1	4	512 KB	16384	256
0.02	0	64	1 MB	1	4096
	1	4	512 KB	265	64
0.04	0	64	4 MB	2048	2048
	1	64	128 KB	4098	1024
0.06	0	4	1 MB	2048	64
	1	64	4 MB	512	64
0.08	0	16	4 MB	256	32
	1	64	2 MB	256	32
0.1	0	64	4 MB	16384	1024
	1	1	256 KB	1024	512

5. Results

To conduct our experiments, we used ESESC [29] simulator which supports the heterogeneous configurations and the multitudinousness we needed. The ESESC (enhanced SESC) simulator uses Time Based Sampling (TBS), which they deemed to be ideal when working with multithreaded executions. As seen in Figure 1, the workloads we used for the experiment consist of six PARSEC benchmarks [30], namely, *swaptions*, *canneal*, *fluidanimate*, *lu_cb*, *blackscholes,* and *freqmine*. All these benchmarks were used in their medium sized inputs. We had to change these benchmarks into libraries prior to our simulation since ESESC supports executing one program at a time that is able to initiate multiple threads which can run independent of each other on separate cores. We had two binaries, one for FTFC and the other for CFS. These schedulers take turns to run and spawn all the six benchmarks from the PARSEC libraries they included, during which their power consumption and throughput are monitored and stored. One thread is allowed to execute for a maximum of 30 seconds and a minimum of 5 seconds. We took 30 samples in this manner and compared the difference.

We define core heterogeneity as any inconsistencies between complementary features of cores, such as microarchitectures, frequency/voltage, or any attributes that can cause performance differences. One of the important program characteristics are instruction level parallelism (ILP), data locality, and branch predictability. The respective attributes for cores are Issue Width, Data Cache Size, and Branch Predictor Size. As such, we achieved core heterogeneity based on variations of these three attributes in addition to the ROB Size. We assume that these four attributes produce significant core performance discrepancy enough to create the arbitrary heterogeneity needed. All other attributes, including the frequency and voltage, were maintained constant. Voltage and frequency were kept constant for each core type. The ESESC simulator allows the modification of various attributes of the cores except the frequency or voltage. As such, there was no problem maintaining these, as it runs on their default static configurations.

We assume each core is executing a single thread at most and assumed the variations of the workloads rather than the type of workloads are more important when choosing the type of benchmarks. Our experiments contained two types of cores sharing a common L3 cache of size 3 Mb and having a dedicated L2 cache of the same or different size. There were three caches of each core type.

To create the configuration of required heterogeneity, we first made a core generator script whose sole purpose was to create a set of cores with random variations of their prominent characteristics. We generated 2000 unique core types by randomly varying the Issue Width, L2 Cache, Branch Predict size, and ROB size. Issue width is 4th power from the range of 1 to 64. L2 Cache is a power of 2 ranging from 128 to 32768. Branch Predict and ROB size are a power of 2 from the range of 16 to 32768. This is unlike the method used in [31], where the instruction set of homogeneous processors is augmented via custom instructions.

The second phase was to assess the performance of generated cores. We did this by running four of the PARSEC benchmarks serially on each core type for a total of 60 seconds. We profiled each core type by calculating the total CPU utilization achieved by those running benchmarks. The benchmark selection is based on their proximity to the characteristics of workloads exhibited in mobile devices. The benchmarks we used for this purpose were swaptions, canneal, fluidanimate, and freqmine.

The third step was to select cores in such a way as to guarantee their combination results in the heterogeneity desired. We took those 2000 utilization values from the second step and used them to generate a set of cores that satisfy any heterogeneity we desired but under the roof of heterogeneity possible given the set of variations in the recorded utilizations. We generated six configurations with an incremental heterogeneity of 0.02, starting from HM = 0. We run six of the PARSEC benchmarks on these cores and used Linux's CFS and the proposed FTFC to schedule them and recorded the total power and throughput. All the six core configurations used can be seen in Table 2.

We compared the result of FTFC with that of CFS. We can see from the result in Figure 4 that when the system is in its

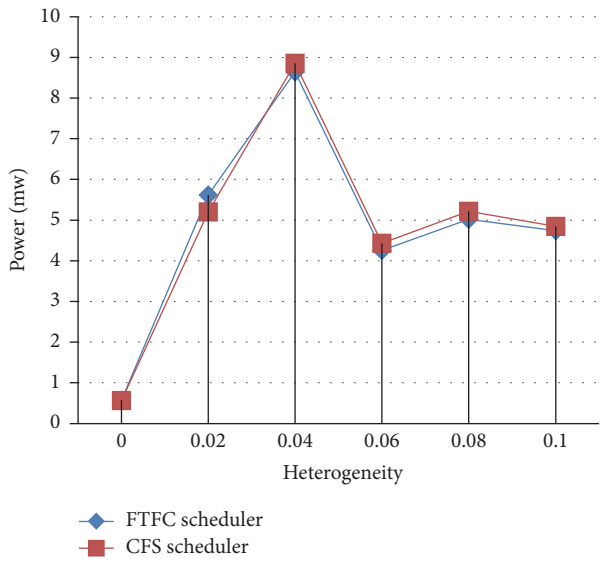

Figure 4: Power comparison between FTFC and CFS.

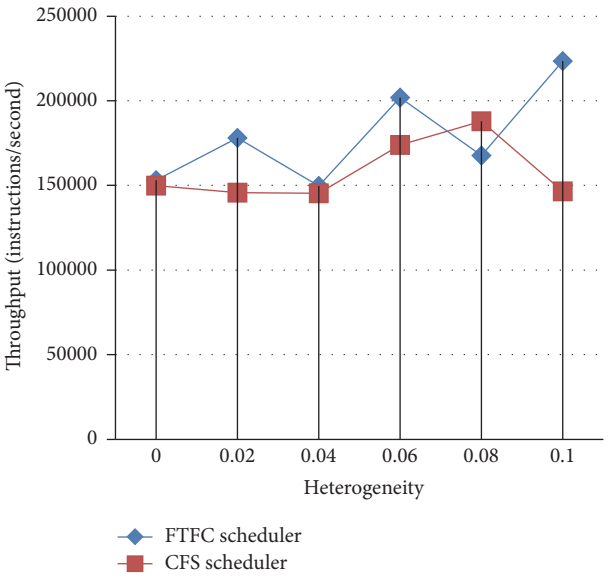

Figure 5: Performance comparison between FTFC and CFS.

highest configuration, that is, HM = 0.1, FTFC saves power by 2.22% and improves performance by 52.62% as compared to CFS scheduler. Figure 5 shows the performance of FTFC as compared to CFS. The variations of workload utilization also play an important role for FTFC scheduler operation. We can see from Figures 4 and 5 that the power and throughput for FTFC are essentially the same as CFS when the system is in its homogeneous configuration; that is, HM = 0. This is because FTFC is built on top of CFS and zero heterogeneity reverts it to operate as CFS. The highest performance gain is gotten when the system was in its highest configuration. The FTFC generally maintains its advantage over the CFS in both its power and throughput across the heterogeneous configurations we investigated.

6. Conclusion

This paper presented a solution to thread mapping on heterogeneous configurations using thread-core performance complementarity based on their CPU utilization and core capacity. Core performance was modeled by getting the total CPU utilization of fixed set of benchmarks running on a core. We were able to achieve a power save of 2.22% and a performance ramp up of 52.62% compared to the CFS scheduler. We were able to do this with an improved time complexity of $O(N \log M)$. As future work, we propose to include frequency/voltage to achieve higher heterogeneity measures.

Competing Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This research was supported by the MISP (Ministry of Science, ICT & Future Planning), Korea, under the National Program for Excellence in SW supervised by the IITP (Institute for Information & communications Technology Promotion) (R22151610020001002) and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2015R1D1A1A01058025) and supported by the Ajou University Research Fund.

References

[1] P. Greenhalgh, *Big. Little Processing with ARM Cortex-A15 & Cortex A7*, White Paper, ARM, Boston, Mass, USA, 2011.

[2] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction," in *Proceedings of the 36th International Symposium on Microarchitecture (MICRO '03)*, pp. 81–92, IEEE, San Diego, Calif, USA, December 2003.

[3] S. Kim, H. Kim, J. Kim, J. Lee, and E. Seo, "Empirical analysis of power management schemes for multi-core smartphones," in *Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication (ICUIMC '13)*, Kota Kinabalu, Malaysia, January 2013.

[4] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, and P. Husbands, "The landscape of parallel computing research," A View from Berkeley. UC Berkeley Technical Report/Electrical Engineering and Computer Sciences 2006-183, 2006.

[5] S. Borkar, "Thousand core chips—a technology perspective," in *Proceedings of the 44th ACM/IEEE Design Automation Conference (DAC '07)*, pp. 746–749, June 2007.

[6] S. Smith, "Many-core operating systems," in *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA-34*, Keynote Speech, 2007.

[7] Q. Han, M. Fan, O. Bai, S. Ren, and G. Quan, "Temperature-constrained feasibility analysis for multicore scheduling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 2082–2092, 2016.

[8] G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo, "Global real-time memory-centric scheduling for multicore systems," *Institute of Electrical and Electronics Engineers. Transactions on Computers*, vol. 65, no. 9, pp. 2739–2751, 2016.

[9] L. He, H. Zhu, and S. A. Jarvis, "Developing graph-based co-scheduling algorithms on multicore computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 6, pp. 1617–1632, 2016.

[10] F. Bodin and S. Bihan, "Heterogeneous multicore parallel programming for graphics processing units," *Scientific Programming*, vol. 17, no. 4, pp. 325–335, 2009.

[11] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, Micro-42*, pp. 45–55, New York, NY, USA, December 2009.

[12] https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt.

[13] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proceedings of the 5th ACM EuroSys Conference on Computer Systems (EuroSys '10)*, pp. 125–138, ACM, Paris, France, April 2010.

[14] D. Shelepov, J. C. S. Alcaide, S. Jeffery et al., "HASS: a scheduler for heterogeneous multicore systems," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 66–75, 2009.

[15] J. Chen and L. K. John, "Efficient program scheduling for heterogeneous multi-core processors," in *Proceedings of the 46th ACM/IEEE Design Automation Conference (—DAC '09)*, pp. 927–930, July 2009.

[16] J. Liu, P. H. Chou, N. Bagherzadeh, and F. Kurdahi, "Power-aware scheduling under timing constraints for mission-critical embedded systems," in *Proceedings of the 38th Design Automation Conference*, pp. 840–845, Las Vegas, Nev, USA, June 2001.

[17] M. Ruggiero, A. Guerri, and D. Bertozzi, "Communication aware systems-on-chip allocation and scheduling framework for stream-oriented multiprocessor," DATE pp 3–8, April 2006.

[18] J. Chen and L. K. John, "Energy-aware application scheduling on a heterogeneous multi-core system," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '08)*, pp. 5–13, IEEE, Seattle, Wash, USA, September 2008.

[19] D. P. Gulati, C. Kim, S. Sethumadhavan, S. W. Keckler, and D. Burger, "Multitasking workload scheduling on flexible-core chip multiprocessors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*, pp. 187–196, Toronto, Canada, October 2008.

[20] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *Proceedings of the 3rd Conference on Computing Frontiers (CF '06)*, pp. 29–39, ACM, Ischia, Italy, May 2006.

[21] D. Koufaty, D. Reddy, and S. Hahn, "Core architecture optimization for heterogeneous chip multiprocessors," in *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*, pp. 125–138, Paris, France, April 2010.

[22] H. Oh and S. Ha, "A static scheduling heuristic for heterogeneous processors," in *Euro-Par'96 Parallel Processing*, vol. 1124 of *Lecture Notes in Computer Science*, pp. 573–577, Springer, Berlin, Germany, 1996.

[23] G. Liu, J. Park, and D. Marculescu, "Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems," in *Proceedings of the IEEE 31st International Conference on Computer Design (ICCD '13)*, pp. 54–61, October 2013.

[24] R. A. Jain and D. V. Padole, "Scalable and flexible heterogeneous multi-core system," *International Journal of Advanced Computer Science and Applications*, vol. 3, no. 12, pp. 174–179, 2012.

[25] R. Kumar, D. M. Tullsen, and N. P. Jouppi, "Core architecture optimization for heterogeneous chip multiprocessors," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT '06)*, pp. 23–32, ACM, Seattle, Wash, USA, September 2006.

[26] T. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Seznec, "Performance implications of single thread migration on a chip multicore," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 80–91, 2005.

[27] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*, pp. 213–224, IEEE, Portland, Ore, USA, June 2012.

[28] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn, "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *Proceedings of the ACM/IEEE Conference on Supercomputing (SC '07)*, November 2007.

[29] E. K. Ardestani and J. Renau, "ESESC: A fast multicore simulator using Time-Based Sampling," in *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA '13)*, Shenzhen, China, February 2013.

[30] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*, pp. 72–81, October 2008.

[31] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "Synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors," in *Proceedings of the 18th International Conference on VLSI Design*, Kolkata, India, January 2005.