Asymmetry-Aware Scheduling in Heterogeneous Multi-core Architectures

Tao Zhang[1,2], Xiaohui Pan[3], Wei Shu[1,2], and Min-You Wu[1]

[1] Shanghai Jiao Tong University, Shanghai, China
{tao.zhang,shu,mwu}@sjtu.edu.cn
[2] University of New Mexico, Albuquerque, USA
{zhang,shu}@ece.unm.edu
[3] Shanghai University of Political Science and Law, China
panxiaohui@shupl.edu.cn

Abstract. As threads of execution in a multi-programmed computing environment have different characteristics and hardware resource requirements, heterogeneous multi-core processors can achieve higher performance as well as power efficiency than homogeneous multi-core processors. To fully tap into that potential, OS schedulers need to be heterogeneity-aware, so they can match threads to cores according to characteristics of both. We propose two heterogeneity-aware thread schedulers, PBS and LCSS. PBS makes scheduling based on applications' sensitivity on large cores, and assigns large cores to applications that can achieve better performance gains. LCSS balances the large core resource among all applications. We have implemented these two schedulers in Linux and evaluated their performance with the PARSEC benchmark on different heterogeneous architectures. Overall, PBS outperforms Linux scheduler by 13.3% on average and up to 18%. LCSS achieves a speedup of 5.3% on average and up to 6% over Linux scheduler. Besides, PBS brings good performance with both asymmetric and symmetric workloads, while LCSS is more suitable for scheduling symmetric workloads. In summary, PBS and LCSS provide repeatability of performance measurement and better performance than the Linux OS scheduler.

Keywords: Scheduling, Heterogeneous, Asymmetric, Multi-core.

1 Introduction

Multi-core processors have become mainstream since they have better performance per watt and larger computational capacity than complex single-core processors. To efficiently utilize on-chip resource, recent research [13] [15] [17] advocates heterogeneous (or asymmetric) multi-core architectures consisting of a combination of cores with different computational capabilities. These processors are attractive because they have the potential to improve system performance, to reduce power consumption[1], and to mitigate Amdahl's law [3]. Since a heterogeneous multi-core architecture consists of a mix of different cores, it can better cater for heterogeneous workloads [2]. People could execute cpu-intensive

C.-H. Hsu et al. (Eds.): NPC 2013, LNCS 8147, pp. 257–268, 2013.

threads on fast cores and memory-intensive threads on slow cores to achieve better energy efficiency.

The heterogeneity of such architectures can be classified into three levels: high, medium and low. A high-heterogeneity processor consists of cores of different instruction set architectures. A typical example is AMD FUSION APU [20] which incorporates CPU cores and GPU(Graphics Processing Unit) cores. A medium-heterogeneity processor integrates cores of overlapping instruction set architectures, such as IBM CELL [21]. A Cell processor contains a power processor element (PPE) with several synergistic processor elements (SPEs). Finally, a low-heterogeneity processor contains cores of same instruction set architectures but different performances. Such cores are called fast and slow cores [16], or big and small cores [17] in previous study.

Despite their benefits in energy and performance, heterogeneous architectures pose significant challenges on the design of operating systems or programming environments, which has traditionally assumed homogeneous hardware [17]. A key challenges is scheduling [7]. Current parallel programming environments such as MIT Cilk [4], Cilk++ [5], OpenMP [6], and the default Linux OS scheduler [7] still assume all cores provide equal performance (Asymmetry-unaware). As a result, the default linux scheduler will schedule these threads in a random fashion, resulting in non-optimal performance [7] [16] [22]. In general, there are two issues: the scheduling is not optimal; system repeatability is low. Different run of the same application(s) has different performance. This phenomenon is called completion time jitter [22].

This paper presents Preference Based Scheduling (PBS) and Large Core Splitting Scheduling (LCSS) for arranging multiple multi-threaded applications on heterogeneous multi-core systems. LCSS is simpler and requires no scheduling hints, but less effective. On the contrary, PBS needs application preference to make scheduling decisions, and improves application performance more significantly than LCSS. Overall, the two proposed scheduling schemes improve performance and repeatability of application performance measurement over the Linux scheduler.

2 Preference Based Scheduling (PBS)

2.1 Application Preference

We define application preference as the degree of performance improvement as it receives more large cores. In general, asymmetry-aware schedulers should allocate more large cores to high-preference applications, and more small cores to low-preference applications. More specifically, we define the speedup of running an application (all its threads) on large cores compared to half large, half slow cores as α, and define the speedup of running an application (all its threads) on half large, half slow cores compared to slow cores as β, then:

- An application has a high-preference if α and β are big and almost identical.
- An application has a low-preference if α is much larger than β.
- An application has a medium-preference if it falls between high-preference and low-preference.

2.2 Correlation between Application Preference and Fork-join

From the applications in PARSEC benchmark, we observed correlation between application preference and its fork-join structures. Applications in high-preference category and low-preference have only one fork-join structure. However, they distinguish from each other by the execution time of the sequential part versus the parallel part. For high-preference applications, the sum of sequential part takes longer time than the parallel part. Given more large fast cores, the performance of high-preference applications increases proportionally. On the contrary, for low-preference applications, the parallel part dominates the total execution time of each application. Therefore, application performance is limited by the slowest thread in the parallel phase. Applications in this category can not benefit significantly when receiving more large cores. On the other hand, the medium-preference applications have multiple fork-join structures. Therefore, their performance is limited not only by the slowest thread in each parallel phase, but also by the sequential phase. In general, their performance sensitivity for number of large cores is between high-preference and low-preference category.

Let α be the summed time of all sequential parts of an application, and β be the summed time of all parallel parts, then we have an alternative definition of application preference:

- An application has a high-preference if it has only one fork-join structure, and $\alpha \geq \beta$.
- An application has a low-preference if it has only one fork-join structure, and $\alpha < \beta$.
- An application has a medium-preference if it has multiple fork-join structures.

2.3 Preference Based Scheduling

Preference Based Scheduling(PBS) allocates processor core resource to application threads according to their preference. Application threads with higher preference have higher priority for large cores. A complete scheduling scheme should contain a policy for initial assignment, a policy for wake up assignment and a policy for load balancing [16]. In this work, we consider the case that the total number of application threads (in contrast to system threads) does not exceed the number of cores. High performance computing is such a case that there is at most 1 application thread running on each core.

```
Input: The set of all large cores in a system: C = {l_1, l_2, ..., l_{k_1}}
Input: The set of all small cores in a system: S = {s_1, s_2, ..., s_{k_2}}
Input: The set of applications to schedule: A = {a_1, a_2, ..., a_N}
Func. UThreads(a_i): return the number of unscheduled threads of a_i;
if a_1.preference = a_2.preference = ... = a_N.preference then
    Assign large cores to threads in first come first serve manner
else
    Q = sum_{i=1}^{N} UThreads(a_i);
    while C ≠ φ and Q > 0 do
        find a_j ∈ A with the highest preference;
        X = UThreads(a_j);
        if |C| ≥ X then
            assign X large cores U to a_j;
            C = C − U;
            A = A − {a_j};
        else
            assign C to a_j;
            C = φ;
        end
        update Q;
    end
    if Q > 0 then
        for each a_i ∈ A do
            X = UThreads(a_i);
            assign X small cores U to a_i;
            S = S − U;
            A = A − {a_i};
        end
    end
end
```

Algorithm 1. Preference Based Assignment (PBA) policy

Preference Based Assignment(PBA) Policy: The policy assigns cores to threads based on the application's preference. Threads within an application inherit its preference. Large cores are assigned to threads with the highest preference, one to each thread. In case there are large cores available, but the remaining applications all have the same preference, then this policy works in First Come First Serve(FCFS) fashion. That is, the first application in task queue receives large cores, then the second application, and so on. The detailed process is described in Algorithm 1.

Wake up on Previous Core (WPC) Policy: When a thread is woken up, it is assigned to the core on which it was previously running. In case that core is occupied by a new thread, then the new thread is migrated to another core following the load balancing rule explained in Algorithm 2.

```
   Input: The set of idle large cores in a system: C
   Input: The set of idle small cores in a system: S
   Input: The set of applications to schedule: A = {a_1, a_2, ..., a_N}.
   Input: A vector recording current core occupation status: which core is
          used by which thread of which application
 1 Func. UThreads(a_i): return the number of unscheduled threads of a_i;
 2 Func. SThreads(a_i): return the number of threads of a_i that running on
   small cores;
 3 Func. LThreads(a_i): return the number of threads of a_i that running on
   large cores;
 4 if a_1.preference = a_2.preference = ... = a_N.preference then
 5 |  Assign large cores to threads in first come first serve manner;
 6 else
 7 |  Sort A in descending order of a_i.preference;
 8 |  for i=1,..,N do
 9 |  |  Q = UThreads(a_i) + SThreads(a_i);
10 |  |  if Q > 0 then
11 |  |  |  if |C| > Q then
12 |  |  |  |  allocate Q large cores U to a_i;
13 |  |  |  |  C = C - U;
14 |  |  |  else
15 |  |  |  |  allocate C to a_i;
16 |  |  |  |  Q = Q - |C|;
17 |  |  |  |  C = φ;
18 |  |  |  |  j = N;
19 |  |  |  |  while Q > 0 and j > i do
20 |  |  |  |  |  P = LThreads(a_j);
21 |  |  |  |  |  if P > 0 then
22 |  |  |  |  |  |  X = P > Q?Q : P;
23 |  |  |  |  |  |  give a_j's X large cores to a_i ;
24 |  |  |  |  |  |  Q = Q - X;
25 |  |  |  |  |  end
26 |  |  |  |  |  j = j - 1;
27 |  |  |  |  end
28 |  |  |  end
29 |  |  end
30 |  end
31 |  for each a_i ∈ A do
32 |  |  X = UThreads(a_i);
33 |  |  assign X small cores U to a_i;
34 |  |  S = S - U;
35 |  |  A = A - {a_i};
36 |  end
37 end
```

Algorithm 2. Preference Based Balancing (PBB) Policy

Preference Based Balancing (PBB) Policy: When performing load balancing, large cores will be assigned to threads with the highest preference. In case remaining applications have a same preference, PBB assigns large cores in First Come First Serve manner, just like the preference based assignment(PBA) policy. This policy ensures: large cores are always busy as long as there are applications; high preference application receives priority in getting large cores.

Load balancing with the PBB policy is made in two steps. The first step constructs a new thread to core mapping according to Algorithm 2. The second step is to migrate old threads or assign new threads onto cores to reach that mapping.

3 Large Core Splitting Scheduling (LCSS)

Large Core Splitting Scheduling (LCSS) tries to distribute large cores among multi-thread applications equally. It is designed to be simple and provide a modest performance.

3.1 Thread Assignment Policy

Large Core Splitting Assignment (LCSA): Given K large cores in a system, then N applications will get K/N large cores each unless an application has less than K/N threads. In that case, spare large cores will be distributed equally among the rest applications. The scheduling algorithm will allocate one core for each thread.

3.2 Wake up Assignment Policy

Wake up on Previous Core (WPC) Policy: When a thread is woken up, it is assigned to the core on which it was previously running. In case that core is occupied by a new thread, then the new thread is migrated to another core.

3.3 Load Balancing Policy

Large Core Splitting Balancing (LCSB) Policy: This policy maintains the evenly distribution of large cores among applications through adjusting the number of large cores of each application. The load balancing is done in two steps. The first step is to construct a new thread to core mapping. The second step is to migrate old threads or assign new thread onto cores to reach that mapping.

4 Experimental Methodology

4.1 Simulation Methodology

We use the Gem5 simulator to construct various heterogeneous systems of desired heterogenity. The Gem5 simulator is an event-driven architecture simulator with

proved accuracy [9]. We simulate two types of cores, large and small. A large core (L) is an Out-of-Order, 7-stage pipeline core while a small core (S) is a simple In-order core. Each core has a 32KB i-cache and a 64KB d-cache, both 2-way associative. All cores share an 8-way associative 2MB L2 cache. All caches use 64 byte lines. There is 4GB external memory. The performance of a large core is roughly twice of a small core. The area of a large core is around 2.5× that of a small core. And we use three heterogeneous multi-core architectures of roughly the same area: 8 large cores and 8 small cores (8L8S), 6 large cores and 13 small cores (6L13S), and 4 large cores and 18 small cores (4L18S).

Gem5 simulator supports full-system simulation which runs a commodity OS and user applications on the simulator. We ran Linux 2.6.27 on the simulator. Besides, we modified and integrated the Clavis tool [14] to control the Linux scheduler to follow our scheduling schemes.

4.2 Workload

We use combinations of applications from the PARSEC benchmark [8] to test our system. The workloads are described in Table 1. There are applications of high, low and medium preference with or without the supplement data. Canneal, Freqming, and Dedup are memory intensive, computation intensive, and communication intensive, respectively. We assign explicit value $3, 6, 9$ to low, medium, and high preference, respectively. In addition, we assign value $+1, -1, -1$ to computation intensive, communication intensive, and memory intensive, respectively. The initial value of application preference and the supplement data are then summed to get a final preference value. In summary, the BF and BB are asymmetric workloads since the two applications in BF or BB have an difference of 3 in preference value. However, the BS and DC are symmetric workloads because applications in BS or DC have identical preference value. Finally, the CF is a weakly asymmetric workload since Canneal and Freqmine have an difference of 2 in overall preference value. All applications are 8 threaded, and their order in the "description" column indicates their order in task queue. For example blackscholes sits before freqmine in task queue.

Table 1. Experiment Workloads

Workloads	Description	Preference	Supp. Data	Symmetry
BF	Blackscholes and Freqmine	Low + high	N.A.	Asymmetric
BB	Bodytrack and Blackscholes	Medium + Low	N.A.	Asymmetric
CF	Canneal and Freqmine	High + High	Mem. + Commp.	Weakly Asymmetric
BS	Blackscholes and Swaptions	Low + Low	N.A.	Symmetric
DC	Dedup and Canneal	High + High	Commu. + Mem.	Symmetric

4.3 Performance Comparison Metric

We use the makespan of all applications as the performance metric. We also perform repeatability test to ensure the correctness of our results. Since application

performance with LIN varies from one run to another, we use the average performance of at least 5 runs with LIN in comparison. The performance of PBS, LCSS and Linux OS scheduler (LIN) will be compared.

5 Results

5.1 Evaluation on Repeatability

To evaluate the repeatability of application performance with PBS and LCSS, we ran the BF and BS workloads for multiple times with PBS and LCSS scheduler respectively, and recorded their execution time. Figure 1 shows the results. For both schedulers, all the execution time is generally within $\{97\%, 103\%\}$ of the average execution time. There are many reasons that could cause this variation of execution time. For example, the difference in scheduling activity, the stochastic interaction between threads, and the interference of system threads (in OS) to application threads. Considering the relative small percent (6%) of time variation, we think that PBS and LCSS provide acceptable repeatability for application performance.

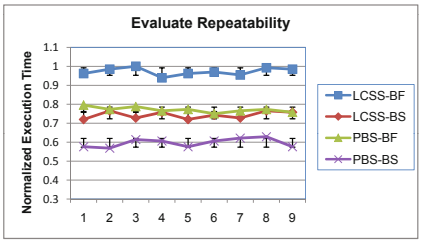

Fig. 1. Evaluation of Repeatability

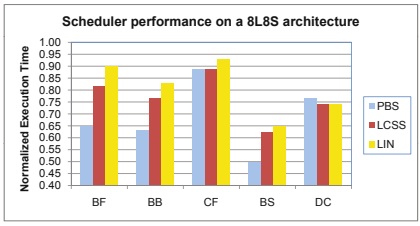

Fig. 2. Execution Time on 8L8S

5.2 Performance Comparison of Schedulers

The performance of PBS, LCSS and LIN scheduling schemes on three heterogeneous architectures are presented in Figure 2, Figure 3, and Figure 4, respectively. Overall, PBS outperforms LIN by 13.3% on average and up to 18%. LCSS has a speedup of 5.3% on average and up to 6% over LIN.

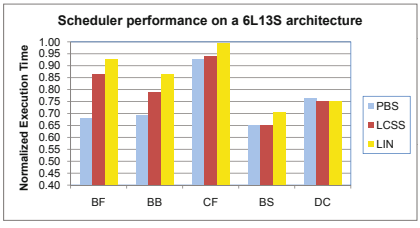

Fig. 3. Execution Time on 6L13S

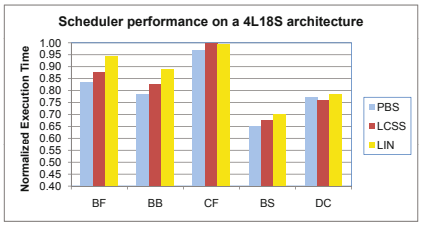

Fig. 4. Execution Time on 4L18S

Of the 15 cases (5 workloads on 3 architectures), PBS scheduler outperforms others in 10 cases. PBS & LCSS work equally best in 2 cases, and LCSS & LIN lead together in another 2 cases. Finally, LCSS win the remaining 1 case. For asymmetric workloads BF, BB and weakly asymmetric workload CF, PBS always achieves the best performance. PBS still works best on the asymmetric workload BS. However, LCSS Outperforms PBS on the asymmetric workload DC.

In summary, PBS is more efficient for asymmetric and weakly asymmetric workloads, while LCSS and LIN are more suitable for symmetric workloads. In some cases, PBS performs better or equally with LCSS on symmetric workloads. As the number of large cores decreases, the platform becomes more and more homogeneous, thus the performance difference among schedulers becomes less prominent.

5.3 Effectiveness of the Load Balancing Policies

To evaluate the effectiveness of the load balancing policies, we compared the performance of PBS and LCSS with and without load balancing, as shown in Figure 5 and Figure 6 respectively. The execution time is the average on three architectures(8L8S, 6L13S, 4L18S). Without load balancing, threads generally run to completion on initially allocated cores, possibly leaving large cores idle and lowering system performance.

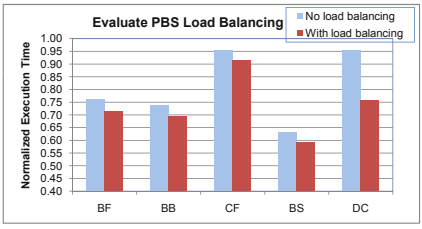

Fig. 5. PBS Load Balancing

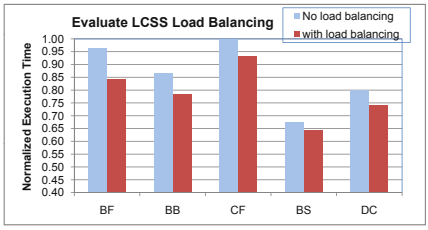

Fig. 6. LCSS Load Balancing

In Figure 6, not surprisingly, performance of all workloads gets better with load balancing. The essentially balanced resource assignment of the LCSS is non-optimal for asymmetric workloads, and load balancing is a remedy. This is illustrated by the substantial improvement with load balancing on the BF and BB workload. On average, employing load balancing improves the performance by 9.2%, which includes the thread migration cost. For the BF workload, the performance of LCSS was improved by 14.4% with core reallocation.

In Figure 5, on average, core reallocation improves the performance by 9.9%, including the thread migration cost. For the DC workload, the performance was improved by 26.3%. Although the average improved performance is close to LCSS's, there are some differences. For LCSS, all five workloads were improved by around 9%. However, for PBS, the DC workload was improved by 26.3% while the other four workloads were improved by less than 7%. This means that

LCSS has a larger space to improve after the initial thread assignment. Since the average performance of the PBS algorithm is better than LCSS (see Figure 2, Figure 3, and Figure 4), we can conclude that the initial thread assignment of PBS is better than LCSS on the BF, BB, and CF workloads.

6 Related Work

Effective task scheduling algorithms are essential for multi-thread applications to make good use of heterogeneous multi-core architectures. Many previous work use DVFS to emulate heterogeneous multi-core systems [7] [10] [15] [16]. They tune the frequencies of cores to get different numbers of large/fast and small/slow cores, and they keep a constant total number of cores in a system. On the contrary, our simulated system consists of large Out-of-Order cores and small In-Order cores, thus having larger micro-architecture differences (eg. heterogeneity). Besides, our system keeps a constant area while the number of cores varies. Our heterogeneous system is similar to that in [17] [18].

Many studies on scheduling in heterogeneous multi-core systems focus on achieving high system throughput by balancing the hardware resources (e.g., cores, caches) among different applications [1] [16] [17] [18] [19]. They make scheduling decision based on runtime profiling/monitoring. Becchi's IPC driven algorithm [18] periodically samples threads' instructions per cycle (IPC) on cores of both types and gives threads that have a higher fast-to-slow IPC ratio priorities in running on the fast cores. Kumar et al. [1] proposes a similar technique, except that he uses more than one sample per core type per thread to improve accuracy. Lakshminarayana et al. [16] proposes age-based scheduling to schedule the threads with larger remaining time to fast cores. The remaining time of threads is predicted at runtime. Koufaty et al. [17] proposes a bias scheduling which matches threads to the right type of cores through dynamically monitoring the bias of the threads in order to maximize the system throughput.

There are some existing scheduling schemes that make no runtime profiling/sampling. Li et al. [7] designed a heterogeneity-aware scheduler for Linux, AMPS, that makes sure the load on each core is proportional to its power and that fast cores are never under-utilized. AMPS needs no scheduling hints, just as our LCSS. However, AMPS does not guarantee optimal performance. For example, it may run a memory-intensive thread on a fast core and lose efficiency. Shelepov et al. [10] proposed HASS that also puts more load on faster cores, but makes this decision based on the offline architectural signature of threads. The architectural signature includes thread information such as cpu-intensive versus memory-intensive, cache miss-rate and so on. This information is generated offline and provided to the scheduler as a hint before scheduling. Similarly, our PBS scheme requires estimated or offline profiled information as scheduling hints, named application preference. Application preference reflects its ability to boost performance through more and more large, fast cores. Our experiment results show that PBS has better performance over LCSS and Linux scheduler.

7 Conclusion

We proposed the PBS and LCSS scheduling schemes to map symmetric and asymmetric workloads efficiently onto heterogeneous architectures. LCSS employs the Large Core Splitting (LCS) idea and aims to balance the large core resource among applications. In contrast, PBS adopts the Preference Based Scheduling policy and aims to assign large cores to applications that can achieve greater performance improvement. LCSS is simpler, and needs no scheduling hints. PBS requires scheduling hints, and provides more significant performance improvements over the Linux OS scheduler. Both scheme ensure that the large cores are always busy unless there are insufficient tasks. Besides, LCSS and PBS guarantee repeatability that the Linux OS scheduler can not provide.

The experiment results show that PBS and LCSS provide better performance than Linux OS scheduler with asymmetric and symmetric workloads on different heterogeneous architectures. Overall, PBS outperforms Linux scheduler by 13.3% on average and up to 18%. LCSS has a speedup of 5.3% on average and up to 6% over Linux scheduler. The results also manifest that PBS can work with both asymmetric, weakly asymmetric and symmetric workloads, although the speedup with asymmetric or weakly asymmetric workloads is bigger. On the contrary, LCSS is more suitable for symmetric workloads. Although we presented only the results for two applications running together, our algorithms are applicable to more than two applications.

Acknowledgment. The authors would like to thank Linghe Kong, Sandy Harris and anonymous reviewers for their fruitful feedback and comments that have helped us improve the quality of this work. This work is supported by Program for Changjiang Scholars and Innovative Research Team in University (IRT1158, PCSIRT), China.

References

1. Kumar, R., Tullsen, D.M., Ranganathan, P., Jouppi, N.P., Farkas, K.I.: Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In: Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA 2004). IEEE Computer Society (2004)
2. Balakrishnan, S., Rajwar, R., Upton, M., Lai, K.: The impact of performance asymmetry in emerging multicore architectures. In: Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA 2005), pp. 506–517. IEEE Computer Society (2005)
3. Hill, M., Marty, M.: Amdahl's law in the multicore era. J. Computer 41(7), 33–38 (2008)
4. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. Journal of Parallel and Distributed Computing 37(1), 55–69 (1996)
5. Leiserson, C.: The Cilk++ concurrency platform. In: Proceedings of the 46th Annual Design Automation Conference, pp. 522–527. ACM (2009)

6. Ayguade, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., Zhang, G.: The design of openmp tasks. IEEE Transactions on Parallel and Distributed Systems 20(3), 404–418 (2009)
7. Li, T., Baumberger, D., Koufaty, D.A., Hahn, S.: Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC 2007). ACM (2007)
8. Bienia, C.: Benchmarking Modern Multiprocessors. Ph.D. Thesis, Princeton University (2011)
9. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. SIGARCH, Computer Architecture News 39, 1–7 (2011)
10. Shelepov, D., Saez, J.C., Jeffery, S.: HASS: a Scheduler for Heterogeneous Multicore Systems. ACM Operating System Review 43(2) (2009)
11. Zhuravlev, S., Blagodurov, S., Fedorova, A.: Addressing shared resource contention in multicore processors via scheduling. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS (2010)
12. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Programming Language Design and Implementation, PLDI (2005)
13. Saez, J.C., Shelepov, D., Fedorova, A., Prieto, M.: Leveraging workload diversity through OS scheduling to maximize performance on single-ISA heterogeneous multicore systems. Journal of Parallel and Distributed Computing (JPDC) (2011)
14. Blagodurov, S., Fedorova, A.: A. User-level scheduling on NUMA multicore systems under Linux. In: Linux Symposium (2011)
15. Chen, Q., Cheny, Y., Huangy, Z., Guo, M.: WATS:Workload-Aware Task Scheduling in Asymmetric Multi-core Architectures. In: IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS). IEEE (2012)
16. Lakshminarayana, N., Lee, J., Kim, H.: Age based scheduling for asymmetric multiprocessors. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pp. 25–36 (2009)
17. Koufaty, D., Reddy, D., Hahn, S.: Bias scheduling in heterogeneous multi-core architectures. In: Proceedings of the 5th European Conference on Computer Systems (EuroSys 2010), pp. 125–138. ACM (2010)
18. Becchi, M., Crowley, P.: Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In: Proceedings of the 3rd Conference on Computing Frontiers. ACM (2006)
19. De Vuyst, M., Kumar, R., Tullsen, D.: Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006), pp. 10–20. IEEE (2006)
20. Brookwood, N.: Amd fusion family of apus C enabling a superior, immersive pc experience. AMD white paper, `http://www.amd.com/us/Documents/48423_fusion_whitepaper_WEB.pdf`
21. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the Cell Multiprocessor. IBM J. Research and Development 49(4/5), 589–604 (2005)
22. Fedorova, A., Vengerov, D., Doucette, D.: Operating System Scheduling on Heterogeneous Core Systems. In: First Workshop on Operating System Support for Heterogeneous Multicore Architectures (2007)