ARMSS: Adaptive Reconfigurable Multi-core Scheduling System

Samar Nour[1*] Shahira Mahmoud[1]

[1]Faculty of Engineering, Helwan University, Egypt,
*E-mail: samar.nour@bue.edu.eg

Abstract

A multi-core processor is a single computing component with two or more independent actual processing elements (called "cores"), which are able to read and execute program instructions. Nowadays Multi-core processors are widely used in many application domains, including general-purpose, embedded systems, network, digital signal processing (DSP), and graphics processing unit (GPU). Task scheduler used for achieving full utilization of system resource, so obtaining an optimum scheduling is one of the most challenging problems. Task scheduling aimed to improve system performance by minimizing makespan for tasks and maximizing resource utilization to those tasks, also increasing the number of cores in a single chip gain performance enhancement in thread level parallelism (TLP) applications while the instruction level parallelism (ILP) suffer from poor performance. This research paper proposed an adaptive multicore architecture that support for a dynamic workload consisting of a mix of ILP and TLP. Offline and online schedulers that dynamically change the number of the cores allocated to an application, is proposed. The performance evaluation of the adaptive multi-core shows minimizing in makespan and maximizing resource utilization compared to other lecture multi-core task schedulers.

Keywords: *Multi-cores, scheduling, utilization, ILP, TLP, malleable and moldable tasks*

1. Introduction

The concept of multicore technology is mainly centered on the possibility of parallel computing, which can significantly boost computer speed and efficiency by including two or more central processing units (CPUs) in a single chip. This reduces the systems heat and power consumption which meaning much better performance with less or the same amount of energy. Our proposal includes a highly accurate dynamically adaptive multicore technique which has been proposed as an effective solution to optimize performance for peak power constrained processors[1]. To the best knowledge, the architecture of a multicore processor enables communication between all available cores to ensure that the processing tasks are divided and assigned accurately. There are many types of multi-core architectures, a processor with asymmetrical cores is one in which the design of the cores is heterogeneous [2]. Typically this means that, in relation to one another, each of the cores can be designed to operate with different instruction sets, clock speeds, and have differing memory and programming models [Figure 1] [3]. The key benefit of such a model is that each of the cores is typically specialized to accomplish a specific type of task, therefore, yielding improved performance, however, there are a number of disadvantages associated with this model as well. First, relatively to processors with asymmetrical cores, development of applications is very complex and more difficult second, due to the specialization of the processors, one that is underutilized cannot be as easily leveraged to assist with general processing. A processor with symmetrical cores is one in which the design of the cores is homogeneous [Figure 1] [3]. Unlike processors with asymmetrical cores, the cores contained in this type of processor are identical to one another and are intended to be used for all purposes and types of tasking. The advantage of these types of processors is that, because there is only one type of core design, developing applications for them is easier relative to processors with asymmetrical cores. Additionally, because the cores are generalized, the unused processing power of one core can be more easily applied to accomplish the tasking of another. Naturally, the one obvious disadvantage of this model is that, it cannot be optimized to perform a particular type of task because the cores are designed for general use [4]. A processor with adaptive cores is physically fabricated as a set of simple identical cores [Figure 1]. At runtime, two or more such simple cores can be coalesced together to create a more complex virtual core. The adaptive cores have the advantages both asymmetric multi-core and symmetric multi-core. In fact, the adaptive architecture dynamically creates different asymmetric multicore configurations during the makespan, in contrast to rigid symmetric and asymmetric solutions [4].

The optimum scheduler should not work in a manner such that some cores become heavily loaded while other cores run underutilization.[5][6][7]. To the best of our knowledge, no previous work characterized the performance of an adaptive multi-core architecture when both

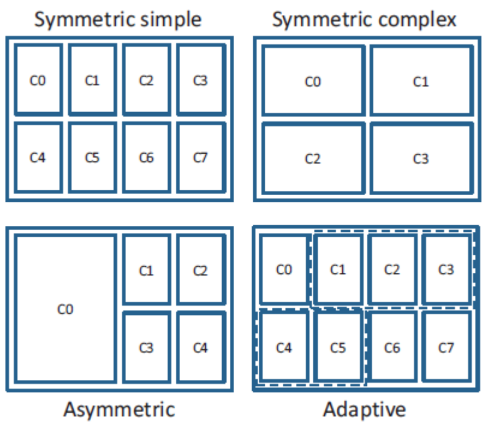

Fig. 1. Multi-core Configurations.

ILP and TLP applications coexist in the system. Our work introduces one of the first performance enhancement study using adaptive multi-core architecture for this realistic scenario. In, Instruction Level Parallelism there are many instructions in code that don't depend on each other. That means its possible to execute those instructions in parallel. This type of parallelism can be found in many applications like graphics, scientific computing, and cryptography. ILP can be exploited in the simple cores, but in asymmetric multi-cores (comprising of both simple and complex cores), gives better performance[8] [9]. On the other hand, multi-cores exploit Thread Level Parallelism (TLP) by executing different threads in parallel on different cores[4] [10] [11]. Symmetric multi-core solutions are the perfect match for parallel applications that can exploit significant thread-level parallelism [12]. Indeed the current trend is to increase the number of cores on the chip to offer more TLP. Adaptive multi-cores appear well poised to support diverse and dynamic workload consisting of a mix of ILP and TLP [4][10][11].

In our paper to avoid load imbalance between the cores, a new model dynamically allocates request based on its service time and the load status of each core.

Multi-core can operate with the two different type of tasks, Moldable task and Malleable task. Moldable tasks are parallel tasks that can be executed using an arbitrary number of cores, but they cannot change the core allocation during execution. The performance of a moldable task is directly related to the number of allocated cores. Suboptimal solutions for scheduling moldable tasks have been studied in [13] [14] [15]. However, Malleable tasks are parallel tasks that may be processed simultaneously by a number of cores, where the speedup of the task is dependent on the number of allocated cores. Malleable tasks are allowed to be preempted in addition to changing the number of cores during execution. Scheduling malleable tasks is a promising technique for gaining computational speedup when solving large scheduling problems on parallel and distributed computers. Malleable tasks real applications include simulating molecular dynamics, Cholesky factorization, operational oceanography and quay allocation[16]. In our work, we model the applications as independent preemptive malleable tasks.

lectures address the problem of getting optimal task allocation to the cores as a critical goal so, one of the biggest issues in such systems is the development of effective techniques for the distribution of the Cores. The problem is how to distribute (or schedule) the cores to achieve some performance goal(s), such as minimizing execution time, minimizing communication delays, and/or maximizing resource utilization [17]. From a system's point of view, this distribution choice becomes a resource management problem and should be considered as an important factor during the design phases of multicores system. Task scheduling can be divided into static, dynamic and adaptive scheduling. In static scheduling, the assignment of tasks to processors is done before program execution begins. On the other hand, dynamic scheduling is based on the redistribution of processes among the processors during execution [17]. An adaptive scheduler is the one which takes many parameters into consideration in making its decisions [18][19]. Our proposed is concerned with the efficient utilization of all the cores and resources for this purpose we need an efficient adaptive task scheduling technique.

Let us explain succinctly, our contribution is aiming to characterize the true performance possibility of an adaptive multi-core architecture. We utilize an optimal schedule that can smart reconfigure and allocate the cores to the tasks so as to minimize the makespan and maximize resource utilization and get load balance to tasks (ILP and TLP). Two different settings problem are considered (offline and online) adaptive task schedulers.

The rest of the paper is organized as follows. Section 2, presents literature review. In Section 3, the system model is presented, where some relevant background and necessary concepts for the proposed algorithm are demonstrated. Section 4, introduces an optimal schedule (Offline, and Online) using adaptive multi-core architecture. Section 5. shows the results of the quantitative evaluation review and final conclusion is introduced in Section 6.

2. Literature Review

A lot of adaptive multi-core architectures were suggested in the article recently. Core Fusion [20] fuses homogeneous cores that used complex hardware mechanism. They evaluate the performance by running ILP tasks and TLP tasks separately on the adaptive multi-core.Voltron exploits various forms of parallelism by conjugation cores that together work as a VLIW processor [21]. It depends on a very complex compiler that must reveal all forms of parallelism found in ILP task. The evaluation is performed with only ILP tasks and configuring the hardware to exploit different types of parallelism in the application. Federation [22] shows an alternative solution where two scalar cores are coalition into a 2-way out-of-order (ooo) core. Again they evaluate the performance by running only ILP tasks. An analytic study is presented where Amdahl's law [23] is acclimatized to different types of multi-core architectures. The study uses simplistic architecture and application forms. In case of an application comprising ILP task, the results show that asymmetric multi-cores can offer possibility speedups higher than symmetric solutions, while adaptive multi-cores are the best option, being able to offer speedups even higher than the asymmetric architectures.
Preemptable Malleable Task Scheduling Problem was introduced in [24]. The issue of optimal scheduling n independent malleable tasks in a parallel core system was studied. It is assumed that the execution of any task can be preempted and the number of cores allocated to the same task can be changed during its execution. The research presented a rectangle packing algorithm, which converts an optimal solution for the relaxed problem, in which the number of cores allocated to a task is not required to be an integer, into an optimal solution for the original issue in on time.
Task Scheduling on Adaptive Multi-Core[4] was presented, It employed the algorithm that allocates and schedules the tasks on varying number of cores by using adaptive multicore, called Bahurupi. Bahurupi can achieve the performance of complex ooo superscalar processor without paying the price of complex hardware. Bahurupi, in contrast, is a hardware-software cooperative solution that demands minimal changes to both hardware and software[25]. In Towards a Dynamic and Reconfigurable Multicore Heterogeneous System[26] was used a complex hardware to implement a dynamic scheduler. It can be used to allocate threads with low ILP on smaller cores and threads with high ILP on bigger ones. In An Adaptive and Hierarchical Task Scheduling Scheme for Multi-core Clusters[27] and Adaptive Workload-Aware Task Scheduling for Single-ISA Asymmetric Multicore Architectures[28] were used Adaptive task scheduling in multi-cores to improve time and speed up but they did not take into their account the system utilization. However, in Adaptive thermal-aware task scheduling for multi-core systems[29] managed on-line adaptive task scheduling to improve system utilization.
Scheduling Algorithms for Asymmetric Multi-core Processors[30] was tried taxonomies of scheduling algorithms for asymmetric multicore architecture. It has been discussed some representative algorithms. Some scheduling algorithms target efficiency, while other algorithms target thread level parallelism or both. Efficiency specialization algorithms try to get better utilization and performance by assigning CPU intensive threads to powerful cores. Also TLP specialized specialization algorithms assign sequential applications and sequential phase of the parallel application to powerful cores.

3. System Model

The problem of Multi-core scheduling on adaptive multicore architectures[Figure 3] has been considered as an NP-hard problem. Today, computer architects use cycle-level simulators to discover and analyze new multi-core designs. In this section, we first present multi2sim simulator. Second introduces our system models, and then we introduce some pertinent background information and concepts necessary for our research.

3.1. The Multi2Sim Simulation Framework

The research on computer architecture simulator is very important because simulator serves as an important tool for developing computer system architectures and software. Simulation works by modeling the behavior of a real system. Because these systems are usually complex, simulation models key aspects of the system and make a few assumptions about the details. This section describes the Multi2Sim simulator and its model of the x86 superscalar architecture. Multi2Sim is a simulation framework for CPU-GPU(Central processing unit - Graphics processing unit) heterogeneous computing. It includes models for superscalar, multithreaded, and multi-core CPUs, as well as GPU architectures. Multi2Sim is an open-source simulator that used C programming language. It can be downloaded from web site [38], a lot of modifications were performed in Multi2sim so that it could be used in our evaluation[31][32].

The Multi2Sim simulation paradigm is described in [Figure 2]. Three distinct modules comprise the simulator: the disassembler, the emulator (or functional simulator) and the timing simulator (or detailed simulator). In essence, the disassembler is responsible for reading

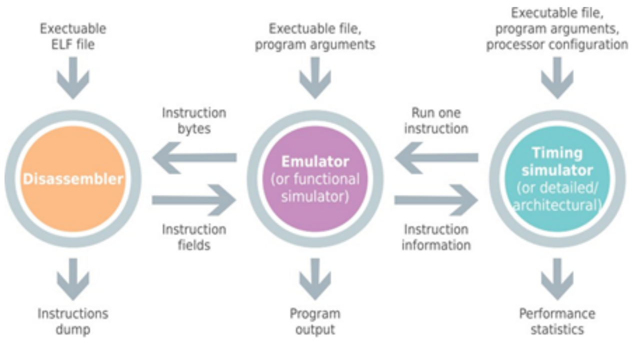
Exectuable ELF file
Exectuable file, program arguments
Executable file, program arguments, processor configuration
Instruction bytes
Run one instruction
Disassembler
Emulator (or functional simulator)
Timing simulator (or detailed/ architectural)
Instruction fields
Instruction information
Instructions dump
Program output
Performance statistics

Fig. 2. Multi2Sim simulation framework.

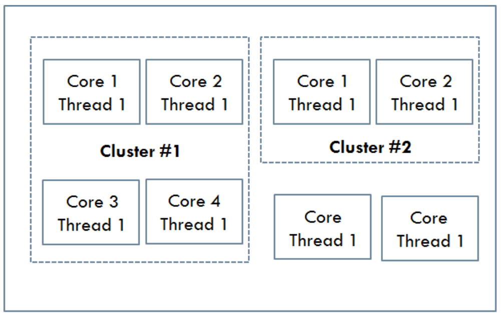

Fig. 3. Adaptive multi-core architecture with 8 cores.

bit streams and interpreting them as machine instructions. the emulator models instruction behavior from an input/output point of view and finally, the simulator models the flow of instructions inside the machine as they execute. The entire simulation framework was modularly designed, such that each module in [Figure 2] requires all modules to the left in order to work (the emulator requires the disassembler, and the simulator requires the emulator). When programs execute on the timing simulator, it requests the functional simulator to execute an instruction. The functional simulator reads the program binary (if necessary) and passes the instruction bytes to the disassembler, which returns the instruction fields. The functional simulator executes the instructions and passes execution information to the timing simulator.

3.2. System Model in The Multi2Sim

Our model has 8 Identical cores consisting of 2-way out-of-order(ooo) cores [Figure 3]. Two clusters have been created from those cores (cluster#1) and (cluster#2), where cluster#1 is consisting 4 Identical cores ($C1-C4$) with one thread each and cluster#2 is consisting of 2 Identical cores ($C1-C2$) with one thread each. Cores in any cluster can be coalesced together to form 4-way, 6-way, or 8-way ooo and created a complex core. For example, a 2-cores coalition work operates a 4-way ooo cores, while a 3-cores coalition operates like a 6-way ooo cores and 4-core coalition operates like an 8-way ooo cores. That coalition could be found in the same cluster. Finally, there are two cores not belong to any cluster.

The adaptive architecture [Figure 3] allows both ILP and TLP tasks to use the timevarying number of cores. Thus, it modeled the applications as malleable workload [33], where the number of cores allocated per task is not fixed and can change during execution through preemption. For the limit study, our goal is to create the optimal schedule for the malleable tasks on the adaptive architecture. We used n tasks mixed from the two main types of Task (TLP and ILP), those tasks are running on m cores (where $n <= m$). The number of cores assigned to each task can be increased according to its utilization time. The utilization of each task was a very important parameter to be calculated during the algorithm running.

We should calculate the Utilization per core. Bini et al. [34] showed that the N tasks will be scheduled on a single-core platform under the following condition:

$$\prod_{i=1}^{N}(U_i+1) <= 2 \tag{1}$$

this calculation determines the possibility of running two tasks on the same core or not.

Also, we need to calculate each cluster utilization U(c) that must satisfy Liu and Layland boundary [35].

$$U(c) <= 1 \tag{2}$$

Our work introduces a study of adaptive reconfigurable multi-core scheduling architecture when both ILP and TLP tasks coexist in the system.

4. Adaptive Reconfigurable Multi-core Scheduling System [ARMSS]

We have been designed an efficient offline and online scheduler using adaptive multi-core architectures. We present an optimal scheduler for a realistic adaptive multi-core architecture. We impose some constraints that help in performing real time task scheduling on adaptive multi-core. our proposed algorithms study both types of scheduler (offline and online). Each algorithm divides tasks that run on the same cluster into high utilization task set (T_uh) and low utilization task set (T_ul). When a task belongs to T_ul finished excitation on one core then a task belongs to T_uh could be used that last core and increased its cores by one. That scenario should be executed between cores within the same cluster only. The maximum number of cores to run one task equal the number of that cluster's cores.

4.1. OFF-Line Adaptive Reconfigurable Multi-core Scheduling System [offline- ARMSS]

When scheduling tasks on a system model, we must take into consideration all the constraints and limitations imposed by the system. More concretely, for our proposed algorithm (ARMSS), we need to consider the following constraints in forming core coalitions for ILP tasks. Those constraints are actually quite generic and are present for almost all adaptive multi-core architectures in the literature even though the exact values for the constraints can be different.
c1) In case of ILP tasks, the difference between the two selected tasks utilization from T_uh(High Utilization Task set) and T_ul(Low Utilization Task set) must be more than 0.1
c2) The same core can run a different type of tasks (TLP and ILP).
c3) ILP task can only use cores that belong to the same cluster
c4) ILP tasks can use at most four cores (Cluster)

ARMSS aimed to achieve max utilization with coalescing between task has low utilization (TLP) and the task has high utilization (ILP).
ARMSS started by making an initialization for the system parameters, in which it is determining the number of tasks (n) and number of cores (m), where n <= m
Those n-independent real-time tasks seeking execution is denoted by $\Gamma = \{\tau 1, \tau 2, ..., \tau n\}$, considering that this task set will be executed on m-identical cores, denoted by C = {C1, C2, . . . , Cm}, with total utilization U and individual utilization Ui those tasks can be of any type TLP or ILP.
Algorithm 1 was trying to get optimization allocation of tasks with different utilization in order to achieve minimum task makespening.
The algorithm begins by loading number of tasks, a number of cores, and clusters, tasks utilization array, and tasks type. Then the algorithm calls Adaptive utilization scheduler function. This function distinguished tasks having low utilization T_ul form tasks having high utilization T_uh after allocating them to cores. After that, the algorithm checks the four constraints and used the check results for reallocating the task to cores and clusters. [Figure 4] explains ARMSS. Tasks information is given in [Table 1].

According to constraint number c1, algorithm 1 divides the given task set into two groups, high utilization task set (T_uhi)= T_uh1, ...T_uhk and low utilization task set (T_ulj)= T_ul1, ...T_ulw. Then it collects tasks from the two groups that satisfy eq. [1 and 2] and at the same time achieved the condition (T_uhi - T_ulj) > 0.1, where i ranged from 1 to k and j ranged from 1 to w. The algorithm trying to increase the number of cores assigned to each task belongs to a high utilization task set group in order to decrease its running time, line 8 to 11 see [Figure 4b] and [Table 1] where T6[belong to T_uhi] and T5[belong to T_ulj].

According to constraint number c2, the algorithm trying to find another core to run part of a task that has high utilization T_uhi (T6). That core actually runs another task has low utilization T_ulj. (T_uhi and T_ulj) that are collected together must belong to different type (TLP, ILP), line 11. See [Figure 4c] and [Table 1] where T6[T_uhi] was chosen to run part of it on the core that runs task T5 [T_ulj].

According to constraint number c3, the algorithm should collect tasks [T_uhi and T_ulj] be running in the same cluster, [Figure 4d]. If T_uhi don't blow to any cluster then collecting T_uhi+T_ulj to a new cluster that satisfies equation[2], line 12 to 30, see [Figure 4d] and [Table 1] where T7[T_uhi] collect with T4[T_ulj].

According to constraint number c4, a maximum number of cores can assign to task equal four cores because of our model maximum cores per cluster equal four. After increasing number of cores assigned to any task, it should recalculate new utilization using equation [3] to this task and update the utilization Array[].

After determining task pair that ensure all the above constraints, the algorithm will increase T_uhi's task running core by one core, see [Figure 4d] the algorithm increase number of cores running task T7 to two cores, and task T6 to three cores.

The final result after applying ARMSS algorithm on tasks [T1 .. T8] [Table 1], is shown in [Table 2] where T6, T5, T3, and T8 were running on the same cluster#1. T7 and T4 were running in the same cluster#2.

International Journal of Scientific & Engineering Research Volume 9, Issue 9, September-2018
ISSN 2229-5518

Table 1. Example of real Tasks

Tasks	Utilization	Type
T1	.7	TLP
T2	.7	TLP
T3	.7	TLP
T4	.6	TLP
T5	.6	TLP
T6	1	ILP
T7	.94	ILP
T8	.78	ILP

Table 2. Final result after apply ARMSS

Tasks	Utilization	Type	Number of cores	Cluster
T1	.7	TLP	1	0
T2	.7	TLP	1	0
T3	.77	TLP	1	Cluster#1
T4	.77	TLP	1	Cluster#2
T5	.75	TLP	1	Cluster#1
T6	.75	ILP	3	Cluster#1
T7	.75	ILP	2	Cluster#2
T8	.78	ILP	1	Cluster#1

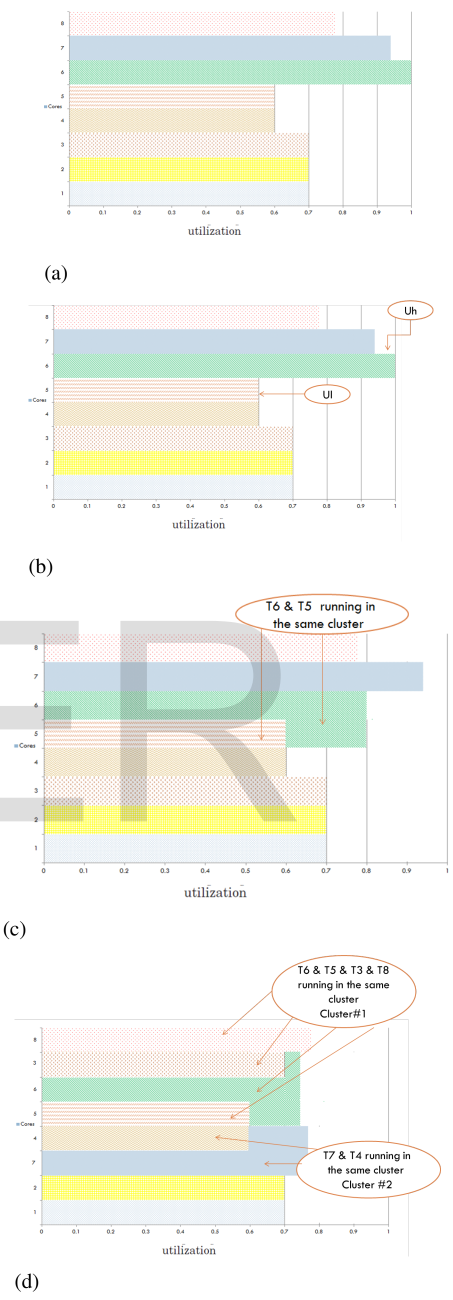

Fig. 4. After Apply ARSSM Offline

ALGORITHM 1: Offline ARMSS Algorithm

```
initialization;
AdaptiveOffline Scheduler
  ($Taskslist, m, n, typeoftasks, Array_u[]$) begin
    n<=m // Types of tasks (TLP, ILP)
    free cluster = m/4
    assign tasks to cores - first in first allocated
end
AdaptiveUtilizationScheduler() begin
    Updated used cluster
    Updated Array_u[]
    Find T_uhi & T_ulj // high & low utilization
    if $T\_uhi - T\_ulj > .01$ then
        while $(T\_uhi+T\_ulj)/2 < 1$ // Liu and Layland boundary do
            if T_uhi and T_ulj not the same type then
                if T_uh has a number of cores > 1 // T_uhi Belongs to a cluster then
                    Find utilization to which has T_uhi
                      after Adding T_uhi + T_ulj
                    if $Ucluster < 1$ then
                        T_uhi & T_ulj, in the same cluster
                          and increase T_uhi's cores by one
                          core
                    end
                    else
                        Break
                    end
                end
                else
                    Find cluster utilization after Adding
                      T_uhi + T_ulj
                    if cluster utilization $< 1$ then
                        T_uhi & T_ulj in the same cluster
                          and increase T_uhi's cores by one
                          core
                    end
                    else
                        Break
                    end
                end
            end
            else
                Find next T_uhi with different type
            end
        end
    end
    else
        Break
    end
end
```

4.2. ON-Line Adaptive Reconfigurable Multi-core Scheduling System [online ARMSS]

In Online ARMS Algorithm we do not have any information about tasks utilization or type. Initialization of Online ARMSS Algorithm includes determining the number of free cores, and free clusters, line 1 to 6. Function OnlineScheduler queue(), first give maximum priority to the task in front of task queue. It assigns that task to the free core if found. This function also updates free cores and used cores lists after each task terminate, line 8 to 16. Second, the function calculates each task utilization, line 19 to 21. If free core found, algorithm 2 increase number of core assigned to the task has maximum utilization by that core. This can be done under constraining that the free core and high utilization task, previous running core(s) belong to the same cluster line 22 to 30. The algorithm tries to speed up the calculation time of tasks that have high utilization. That is done by increasing number of cores running those tasks with the aid of free cores if found. If there isn't free core, then the algorithm searching for high and low utilization tasks run on the same cluster. Then let the core which assigns to the task has low utilization run part of the high utilization task. This will increase the number of core assign to high utilization task by one core. The above processes can't be done unless eq.1 and 2 are not satisfied, line 31 to 43.

ALGORITHM 2: Online ARMSS Algorithm

```
initialization;
InitAdpativeonlineSchedur $(m, n)$ begin
    Free cores=m
    Free cluster = m/4
    Updated free cores (m, used cores )
end
OnlineScheduler_queueu() begin
    if task_queue.empty() ==FALASE then
        if free cores > 0 then
            Next task =teask.queue.foront()
            Pace_task(next_task)
        end
        if any current task is finished then
            Updated used cores
            Updated free cores (m, used cores )
        end
    end
    else
        // calculate utilization to all Tasks
        for all task n do
            Array_u[]=utilization_per_task
        end
        if free cores > 0 then
            Find T_uhi from Array_u[] //high utilization
            if T_uhi and free cores in the same cluster then
                Increase T_uhi's cores by one core
            end
            else
                Find task with max utilization in the
                  (T_uhi), cluster which has free cores
                Increase task (T_uhi) 's cores by one core
            end
        end
        else
            Find T_uhi & T_ulj from Array_u[] // high and
              low utilization
            while (T_uhi+T_ulj)/2 < 1 // Liu and Layland
              boundary do
                if T_uhi and T_ulj in the same cluster then
                    Increase T_uh's cores by one core
                end
                else
                    Find in T_ulj cluster a task with which
                      max utilization(T_uhi)
                    Increase T_uhi's cores by one core
                end
                Find next T_uhi from Array_u[]
            end
        end
    end
    Updated Array_u[], Updated free cores (m, used cores )
end
```

5. Results

In this section, we first present the quantitative characterization of our algorithm ARMSS's performance (offline and online). Comparing between different types of Multi-core Task scheduler (static scheduling, dynamic scheduling, adaptive scheduling (asymmetric, symmetric) and Bahurupi) and our proposed algorithm [ARMSS].

Table 3. Benchmark applications used in our study.

Type	Suite	Inputs	Benchmarks
ILP	SPEC2006	nput.program	bzip
		capture.tst	gobmk
		hyperviscoplastic.inp	calculix
		retro.hmm	hmm
		test.txt	sjeng
		lbm.in	lbm
		an4.ctl	sphinx
		inp.in	mcf
	SPEC2000	mesa.ppm	mesa
		crafty.in	crafty
	MiBench	runme large.sh	basicmath
			bicount
			qsort
			susan
			dijkstra
			patricia
			sha
			adpcm
			fft
			gsm
			stringsearch
TLP	PARSEC	simsmall	blackscholes
			swaptions
			canneal
			vips
			bodytrack

5.1. Workload

Tasks chosen in our study contain 21 ILP applications and 5 TLP applications. ILP applications were taken from (SPEC2006, SPEC2000 [37] and embedded MiBench benchmark suites [36]). TLP applications were taken from PARSEC benchmark suite [39] [40].

The characteristics of the benchmarks appear in Table [3]. We generate different workload (task sets) consisting of varying mix of ILP and TLP tasks. Across all the tasks sets, the ratio of ILP tasks ranges from 35% to 85%, so the ratio of TLP tasks ranges from 15% to 65%.

5.2. Multi-Core Task Scheduler

Multi2Sim introduces the concept of task scheduling similar to the idea of process scheduling in an operating system. The schedule is aimed for mapping software tasks to processing nodes (hardware threads). There are two types of task scheduling in Multi2Sim The Static Scheduler and The Dynamic Scheduler [38].

5.2.1. The Static Scheduler in Multi2Sim

First assigns tasks to threads within a single core and then goes to the next core after one fills up. application using the parallel code, (first threads, then cores). Context switches are not allowed. A running context holds the allocated hardware the first thread until the simulation ends.

5.2.2. The Dynamic Scheduler in Multi2Sim

Mapping of initial contexts at startup does not differ from the static scheduler. However, these allocations are not definitive, and they can vary at runtime. If an allocated context exceeds this quantum, and there is any unallocated context waiting for execution. context stores the processing node identifier of its last allocation, it tries to return to the same processing node where it was run for the last time if it is available. New spawned contexts try to find a processing node that has not been used before by any suspended or evicted context so that any unallocated context waiting for execution can allocate the released processing node again.

5.2.3. The Adaptive Scheduler in Multi2Sim

We introduce another type of task scheduler in Multi2Sim called an Adaptive Scheduler like a Dynamic Scheduler in initialization but different in ContextQuantum value where ContextQuantum= ContextQuantum/FreeNodes and Maximum FreeNodes = AllNode/2. Finally, if task exceeds ContextQuantum and there is any unallocated task waiting for execution increase tasks cores by one core. See [Figure 5] where

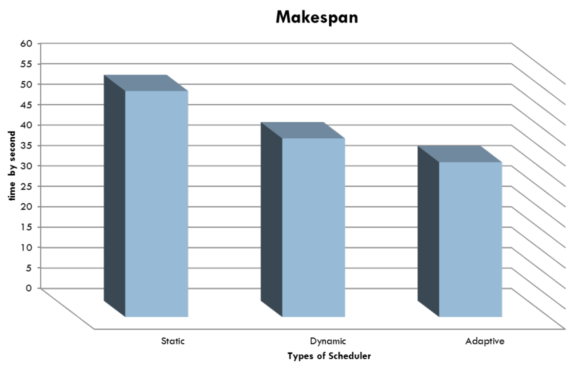

Fig. 5. Multi-Core task scheduler in multi2sin

The Adaptive Scheduler achieves minimum makespan compared to Static Scheduler and Dynamic Scheduler. Our study used Adaptive Scheduler asymmetric and symmetric Multicore configurations, Table 4. Reviews different multi-core configurations used in this study[25].

Table 4. Multi-core configurations used in our study.

Configuration	Description
(S1) 8x2-way	Symmetric eight 2-way cores
(A1) 2x4-way + 4x2-way	Asymmetric two 4-way + four 2-way cores
(Bahurupi) 8x2-way	Bahurupi Adaptive multi-core

5.3. Speedup Functions

Speedup is categorized in computer architecture a process for increasing the performance between two systems processing the same problem. More technically, it is the improvement in speed of execution of a task executed on two similar architectures with different resources. The notion of speedup was established by Amdahl's law, which was particularly focused on parallel processing. However, speedup can be used more generally to show the effect on performance after any resource enhancement. We compile the parallel task with r threads and execute the threads on r cores to obtain the speedup. In other words, the speedup function represents the ideal scenario where the number of threads is equal to the number of cores. In reality, a parallel task compiled with r threads may need to use a different number of cores during execution. Similarly, for an adaptive architecture, a task can be allocated a varying number of cores during execution. However, we noticed little difference in performance when an application compiled with m threads executed on r cores where r<m. We use speedup obtained from core coalition. As going beyond 8-way cores does not provide further speedup due to limited ILP, we restrict the speedup function for serial tasks to the 8-way core.

We compute the average speedup of the task across all the online-ARMSS schedules in which the task participates. The speedup is computed w.r.t. the execution time of the task on a single 2way core. [Figure 6] shows the speedup for the ILP Tasks. Offline-ARMSS is the clear winner here and provides the best speedup for each application among all the multi-core task scheduler. Dynamic-Multi2Sim deploys 2-way cores and hence has better speedup for serial tasks compared to Adaptive-Multi2Sim (S1) using 2-way cores. Adaptive-

Multi2Sim (A1) gives modulate performance. Bahurupi offline simulator gives performance higher than Adaptive-Multi2Sim but lower than ARMSS for serial tasks.
The speedup TLP tasks are close to Adaptive-Multi2Sim(S1), that is because S1 has a large number of simple and identical cores. While Static-Multi2Sim and Dynamic-Multi2Sim perform quite badly for parallel applications see [Figure 7].

5.4. Makespan

Makespan is the time when all tasks have been completed execution. ARMSS has given Minimum Makespan Scheduling comper to other task schedulers (Static scheduling, Dynamic scheduling, Adaptive scheduling (S1, A1) and Bahurupi). For this research, we have chosen a set of five benchmarks: three sequential applications (gobmk, bicount and fft) from SPEC and MiBench benchmark suites that can exploit ILP through complex ooo cores, and two parallel applications (bodytrack and blackscholes) from PARSEC benchmark suite [37] that can exploit TLP through multiple simple cores. The result shows that ARMSS took 375 Cycles (x 1MIL) while in Bahurupi took 400 Cycles (x 1MIL) so, ARMSS gives Minimum Makespan.

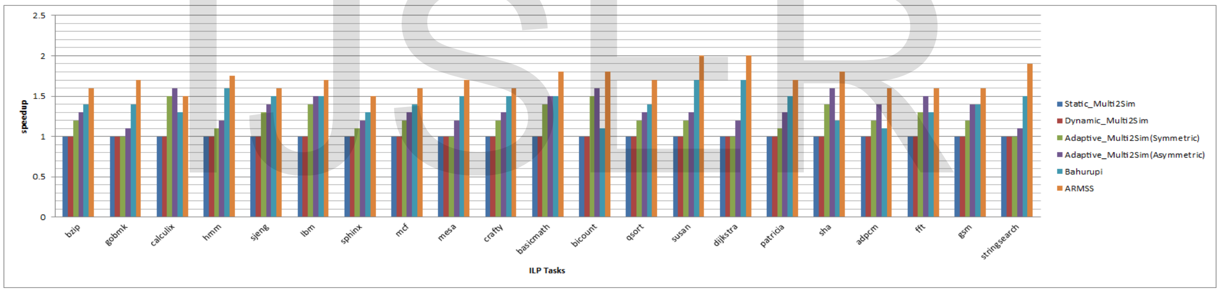

Fig. 6. Speedup of ILP tasks averaged across all task sets and normalized w.r.t. execution on a native 2-way core.

5.5. Processor Utilization

CPU utilization refers to a computer's usage of processing resources or the amount of work handled by a CPU. Actual CPU utilization varies depending on the amount and type of computing tasks. Certain tasks require heavy CPU time, while others require less because of non-CPU resource requirements. In another word CPU utilization is the proportion of the total available processor cycles that are consumed by each process.
Figure[8], reports the processor utilization (averaged across all tasks sets) for different architectures scheduling offline. The results show that the Offline-ARMSS has the best utilization (95%) and performance which, making it the most efficient architectures. In contrast, Bahurupi has a next higher utilization (92%) but low performance, making it the least efficient multi-core architecture. Static-Multi2Sim has low utilization (43%) and Dynamic-Multi2Sim also has low utilization (50%) as it can only exploit TLP from parallel tasks. The serial tasks keep only a subset of the cores busy. Adaptive-Multi2Sim(S1) has good utilization (72%) making it more efficient than Adaptive-Multi2Sim(A1).

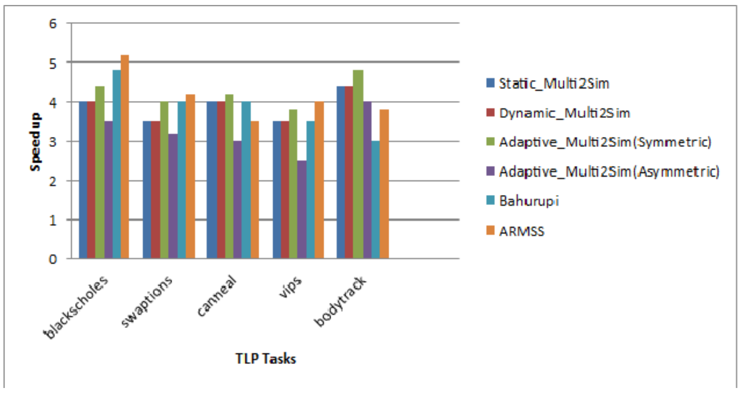

Fig. 7. Speedup of TLP tasks averaged across all task sets and normalized w.r.t. execution on one 2-way core

In Figure[9], Online-ARMSS shows very good efficiency with (81%) average utilization, while Bahurupi simulator shows the next beast efficiency with (76%) average utilization. Similarly, Static-Multi2Sim shows the lowest utilization (33%) due to the large number of simple cores that can only benefit the parallel applications, whereas the serial applications keep the occupied cores busy for a long time. The measured average competitive ratio between our online scheduler and an optimal online scheduler (obtained using strip packing) is 1.14.

5.6. Harmonize ILP and TLP

ARMSS Task Scheduler is successful in accelerating both ILP and TLP tasks. While symmetric architecture with a large number of simple cores is quite effective for TLP, it shows poor performance for serial tasks. Asymmetric architecture can perform well for ILP tasks due to the presence of a complex core but performs badly for TLP tasks. Among the static asymmetric configurations, the configuration asymmetric provides the best balance of ILP and TLP speedup, but is far behind adaptive multi-core architecture ARMSS.

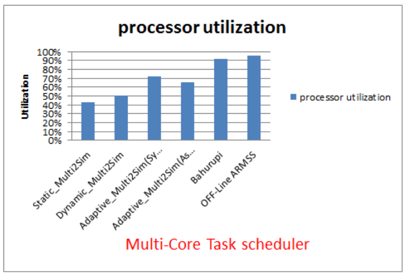

Fig. 8. Utilization of architectures in the offline schedule

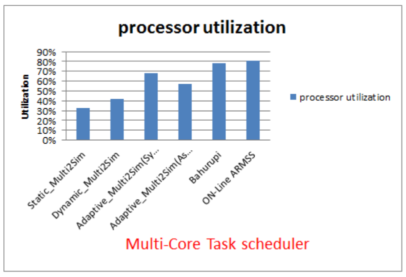

Fig. 9. Utilization of architectures in the online schedule.

6. Conclusions

We have presented a comprehensive quantitative approach to enhance the performance potential of adaptive multi-core architectures compared to static symmetric and asymmetric multi-cores. Scheduling Algorithm Optimization Criteria are measured by maximum utilization, maximum throughput and minimum makespan time. We employ an optimal algorithm that allocates and schedules the tasks on varying number of cores so as to achieve those criteria. Our study considers a mix of sequential and parallel workloads to observe the capability of adaptive multi-cores in exploiting both ILP and TLP. our research proposed an adaptive multicore architecture (ARMSS). This architecture supports for a dynamic workload consisting of a mix of ILP and TLP applications. two studying cases (offline and online) are considered. In both cases, ARMSS scheduler changes the numbers of cores allocated to each task dynamically taking into consideration that task utilization and the processor load. ARMSS's utilization is the highest where it achieved 95% in Offline and 82% in Online. The other Task schedule (Adaptive-Multi2Sim(A1), (Adaptive-Multi2Sim(S1), Bahurupi Static-Multi2Sim, Dynamic-Multi2Sim) achieved less result. The results show that ARSMM (online, and offline) achieve maximum throughput, speedup, utilization, and minimum makespane comparing to the other lecture scheduling algorithms.

References

[1] Yatish Turakhia, Guangshuo Liu, Siddharth Garg, and Diana Marculescu. 2017. Thread Progress Equalization: Dynamically Adaptive Power-Constrained Performance Optimization of Multi-Threaded Applications. IEEE Trans. Comput. 66, 4 (2017), 731744.

[2] Rakesh Kumar, Dean M Tullsen, Parthasarathy Ranganathan, Norman P Jouppi, and Keith I Farkas. 2004. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on. IEEE, 6475.

[3] Mihai Pricopi and Tulika Mitra. 2014. Task scheduling on adaptive multi-core. IEEE transactions on Computers 63, 10 (2014), 25902603.

[4] Ke Ning, Gabby Yi, and Rick Gentile. 2005. Single-chip Dual-core Embedded Programming Models for Multimedia Applications. ECN Magazine (2005).

[5] Juhi Saha, Hee Jun Park, Alex Kuang-Hsuan Tu, Thomas Andrew Morison, Tao Xue, and Ronald Frank Alton. 2017. Multi-core dynamic workload management using native and dynamic parameters. (July 11 2017). US Patent 9,703,613.

[6] Alan David. 2017. Scheduling Algorithms for Asymmetric Multi-core Processors. arXiv preprint arXiv:1702.04028 (2017).

[7] Ke Wang, Kan Qiao, Iman Sadooghi, Xiaobing Zhou, Tonglin Li, Michael Lang, and Ioan Raicu. 2016. Loadbalanced and locality-aware scheduling for data-intensive workloads at extreme scales. Concurrency and Computation: Practice and Experience 28, 1 (2016), 7094.

[8] B Ramakrishna Rau and Joseph A Fisher. 1993. Instruction-level parallel processing: history, overview, and perspective. The journal of Supercomputing 7, 1-2 (1993), 950.

[9] Dake Liu. 2008. Embedded DSP processor design: Application specific instruction set processors. Vol. 2. Elsevier.

[10] David Tarjan, Michael Boyer, and Kevin Skadron. 2008. Federation: Repurposing scalar cores for out-oforder instruction issue. In Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE. IEEE, 772775.

[11] Hongtao Zhong, Steven A Lieberman, and Scott A Mahlke. 2007. Extending multicore architectures

to exploit hybrid parallelism in single-thread applications. In High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on. IEEE, 2536.

[12] Konrad Malkowski. 2008. Co-adapting scientific applications and architectures toward energy-efficient high performance computing. The Pennsylvania State University.

[13] Jacek Blazewicz, Mieczyslaw Drabowski, and Jan Weglarz. 1986. Scheduling multiprocessor tasks to minimize schedule length. IEEE Trans. Comput. 35, 5 (1986), 389393.

[14] J Baewicz, Maciej Drozdowski, G Schmidt, and Dominique de Werra. 1994. Scheduling independent multiprocessor tasks on a uniform k-processor system. Parallel computing 20, 1 (1994), 1528.

[15] Guan-Ing Chen and Ten-Hwang Lai. 1988. Preemptive scheduling of independent jobs on a hypercube. Inform.Process. Lett. 28, 4 (1988), 201206.

[16] Jacek Blazewicz, TC Edwin Cheng, Maciej Machowiak, and Ceyda Oguz. 2011. Berth and quay crane allocation: a moldable task scheduling model. Journal of the Operational Research Society 62, 7 (2011),11891197.

[17] Behrooz A Shirazi, Krishna M Kavi, and Ali R Hurson. 1995. Scheduling and load balancing in parallel and distributed systems. IEEE Computer Society Press.

[18] Thomas L. Casavant and Jon G. Kuhl. 1988. A taxonomy of scheduling in general-purpose distributed computing systems. IEEE Transactions on software engineering 14, 2 (1988), 141154.

[19] Joshua R Bertram and Branden H Sletteland. 2017. Multicore adaptive scheduler. (May 2 2017). US Patent 9,639,401.

[20] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F Martinez. 2007. Core fusion: accommodating software diversity in chip multiprocessors. In ACM SIGARCH Computer Architecture News, Vol. 35. ACM, 186197.

[21] Hongtao Zhong, Steven A Lieberman, and Scott A Mahlke. 2007. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on. IEEE, 2536.

[22] David Tarjan, Michael Boyer, and Kevin Skadron. 2008. Federation: Repurposing scalar cores for out-oforder instruction issue. In Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE. IEEE,772775.

[23] Mark D Hill and Michael R Marty. 2008. Amdahls law in the multicore era. Computer 41, 7 (2008). Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F Martinez. 2007. Core fusion: accommodating software diversity in chip multiprocessors. In ACM SIGARCH Computer Architecture News, Vol. 35. ACM,186197.

[24] Jacek Blazewicz, Mikhail Y Kovalyov, Maciej Machowiak, Denis Trystram, and JanWeglarz. 2006. Preemptable malleable task scheduling problem. IEEE Trans. Comput. 55, 4 (2006), 486490.

[25] Mihai Pricopi and Tulika Mitra. 2012. Bahurupi: A polymorphic heterogeneous multi-core architecture. ACM Transactions on Architecture and Code Optimization (TACO) 8, 4 (2012), 22.

[26] Jeckson Dellagostin Souza, Luigi Carro, Mateus Beck Rutzig, and Antonio Carlos Schneider Beck. 2014. Towards a dynamic and reconfigurable multicore heterogeneous system. In Computing Systems Engineering (SBESC), 2014 Brazilian Symposium on. IEEE, 7378.

[27] Yizhuo Wang, Yang Zhang, Yan Su, Xiaojun Wang, Xu Chen, Weixing Ji, and Feng Shi. 2014. An adaptive and hierarchical task scheduling scheme for multi-core clusters. Parallel Comput. 40, 10 (2014), 611627.

[28] Quan Chen and Minyi Guo. 2014. Adaptive workload-aware task scheduling for single-ISA asymmetric multicore architectures. ACM Transactions on Architecture and Code Optimization (TACO) 11, 1 (2014),8.

[29] Hsin-Hao Chu, Yu-Chon Kao, and Ya-Shu Chen. 2015. Adaptive thermal-aware task scheduling for multicore systems. Journal of Systems and Software 99 (2015), 155174.

[30] Alan David. 2017. Scheduling Algorithms for Asymmetric Multi-core Processors. arXiv preprint arXiv:1702.04028 (2017).

[31] R Ubal, J Sahuquillo, S Petit, and P Lopez. 2007. Multi2sim: A simulation framework to evaluate multicoremultithread processors. In IEEE 19th International Symposium on Computer Architecture and High Performance computing, page (s). Citeseer, 6268.

[32] Xun Gong, Rafael Ubal, and David Kaeli. 2017. Multi2Sim Kepler: A detailed architectural GPU simulator. In Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on. IEEE, 269278.

[33] Joseph YT Leung. 2004. Handbook of scheduling: algorithms, models, and performance analysis. CRC Press.

[34] Bjorn Andersson and Jan Jonsson. 2003. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on. IEEE, 3340.

[35] Bjorn Andersson and Jan Jonsson. 2000. Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. In Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on. IEEE, 337346.

[36] MiBench benchmark http://vhosts.eecs.umich.edu/mibench.

[37] SPEC CPU Benchmarks http://www.spec.org/benchmarks.html.

[38] The Multi2Sim Simulation Framework http://www.multi2sim.org.

[39] P-E Bernard, Thierry Gautier, and Denis Trystram. 1999. Large scale simulation of parallel molecular dynamics. In Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings. IEEE, 638644.

[40] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In Proceedings of the 17th international conference on Parallel architectures and compilation techniques. ACM, 7281.