Nearest Neighbor Affinity Scheduling In Heterogeneous Multi-Core Architectures

Fadi N. Sibai
College of Information Technology, UAE University
Al Ain, United Arab Emirates

ABSTRACT

Asymmetric or heterogeneous multi-core (AMC) architectures have definite performance, performance per watt and fault tolerance advantages for a wide range of workloads. We propose a 16 core AMC architecture mixing simple and complex cores, and single and multiple thread cores of various power envelopes. A priority-based thread scheduling algorithm is also proposed for this AMC architecture. Fairness of this scheduling algorithm vis-a-vis lower priority thread starvation, and hardware and software requirements needed to implement this algorithm are addressed. We illustrate how this algorithm operates by a thread scheduling example. The produced schedule maximizes throughput (but is priority-based) and the core utilization given the available resources, the states and contents of the starting queues, and the threads' core requirement constraints. A simulation model simulates 6 scheduling algorithms which vary in their support of core affinity and thread migration. The simulation results that both core affinity and thread migration positively effect the completion time and that the nearest neighbor scheduling algorithm outperforms or is competitive with the other algorithms in all considered scenarios

Keywords: asymmetric multiprocessors, multi-core architectures, thread scheduling.

1. INTRODUCTION

Asymmetric multi-core (AMC) and symmetric multi-core architectures are gaining ground [1-3]. The motivation behind the multi-core (MC) architecture is as follows. Higher performance single cores are getting more complex and harder to design and validate. Complex features have been added with diminishing returns on performance. Higher performance by increasing frequency implies higher power envelopes. Higher power envelopes with diminishing circuit geometries imply higher power densities and more hot spots on the chip. These have significant heat sink or cooling costs and reliability implications. Moreover, low end MC architectures with 4 or less cores are easy to design by cutting and pasting and making enhancements to cache and memory bandwidths. Larger MCs require higher scalability in their interconnect and memory subsystems. On top of that, operating systems, applications, libraries, and background tasks all demand computation requirements that are satisfied by MCs. A symmetric MC architecture comprises identical CPU cores with identical capabilities and features. They suit workloads with roughly equivalent threads with similar computation (e.g. vector) requirements. Unfortunately, realistic workloads with applications, background tasks, and operating system kernel all running simultaneously are typically composed of threads of different computation and time requirements, some running fine with simple low-power ALUs while others needing more advanced power-consuming vectorized ALUs and predictors. Some threads run better solo while others run better together when scheduled on the same simultaneous multithreading (SMT) processor [4].

Making the MC architecture asymmetric brings great benefits. First of all, the mixture of low-power simple processor cores with high-power complex processor cores fit realistic workloads with a basket of low computation threads and high computation threads. Second, the argument for symmetric MC architectures with simple low-power cores would crumble next to workloads with one or more single threads that run solo and would benefit from a high-power and high instruction level parallelism (ILP) core. The argument for symmetric MC architectures with complex high-power but a limited number cores would crumble next to workloads with a large number of cooperating threads that require each simple CPU computation. Therefore a mixture of cores provided by AMC architectures would satisfy either scenario while cutting down on the power consumption of symmetric MC architecture with high-power complex cores. Functional and performance validation of AMC architectures however still presents a formidable challenge. Compared to a single complex high-frequency and high-power core with ILP-rich features, AMCs provide higher throughput and better performance per watt on multiple thread workloads.

In this paper, we review the latest scheduling and related cache partitioning schemes for multi-cores in section 2. We propose a 16 core AMC architecture in section 3. We detail a priority-based scheduling algorithm for that AMC architecture in section 4. A detailed example that illustrates the working of this scheduling algorithm is also discussed in section 4. Section 5 describes the simulation model and presents simulation results. We conclude the paper in section 6.

2. MC SCHEDULING AND CACHE PARTITIONING ALGORITHMS

MC processors or Chip Multiprocessors (CMP) have been proposed to improve performance and power requirements. In MC processors, the pursuit of higher frequency designs is replaced by the integration of more processors in a single package reducing latencies, and improving the sharing of resources such as second level (L2) cache memories and reducing power consumption and heat dissipation per unit area. CMP [5] refers to the implementation of a shared memory multiprocessor on a single chip. Several commercial CMPs are available on the market [6-8] ranging from 2 to 16 cores per chip.

While hardware features and compiler optimizations may greatly benefit a program's performance, a scheduling algorithm tailored for the architecture it targets can bring even higher benefits. Scheduling a thread with another thread on a dual core processor can be disastrous if the threads both simultaneously contend for insufficiently

available shared resources. Even worse expectations can result from the operating system scheduler scheduling lower priority tasks before higher priority ones. Some of the important issues relevant to MC architectures that have been recently tackled by researchers are single thread migration, shared resource partitioning among co-scheduled threads and cache fair scheduling. Constantinou et al [9] studied the impact of single thread migration in multi-cores with a shared L2 cache on the system performance, and highlighted the performance benefits from migrating the thread to the core it previously ran on and from cores remembering their predictor state since their previous activation, as better performance results from caches and predictors being warmed up. Shaw [10] studied the migration of data and threads in CMPs using vectors that convey locality and resource usage information. Migration time of threads has been measured in Windows-based clusters [11] and network of workstations [12]. In MC processors, thread migration was estimated to take under 200 processor cycles [9] although ideally it should take close to no time. Another critical issue in SMT and MC CPUs is allocation of shared resources to competing threads. [13] points to a strong link between L2 cache contention and thread performance on SMT processors. [2] found that asymmetry hurts performance predictability and scalability of commercial server workloads on asymmetric cores and recommended making both the operating system kernel and the application aware of the hardware asymmetry in order to circumvent these issues.

In CMPs, fair cache sharing and partitioning [14] was found to optimize throughput. Fairness measures the performance slow down (e.g. thrashing) of parallel threads due to cache sharing. [14] assumes that the operating system enforces thread priority by giving more time slices. This is problematic as the operating system when assigning time slices assumes that all threads get equal resources but that is not the case with parallel threads contending to L2 cache space. With cache partitioning methods that optimize cache fairness among the parallel threads, the operating system scheduler becomes fair. Chandra [15] evaluated 3 models for predicting the impact of cache sharing on co-scheduled threads. Another CMP cache fair scheduling algorithm idea [16] is to give larger time slices to co-scheduled threads that suffer more from extra L2 cache misses due to being scheduled with other threads. The application user is expected to specify the thread priority and the thread class. Their algorithm helped the performance of applications with low cache requirements but hurt the performance of applications with large cache requirements. The overall response times of co-schedules threads that they considered was not impressive, however the fairness criterion was met.

In the next section, we propose an asymmetric multi-core architecture and detail a thread scheduling algorithm for it based on core utilization and availability, and core and thread affinities, and discuss its hardware requirements. We then study the effectiveness of this scheduling scheme compared to non-migratory scheduling schemes and other scheduling schemes which allow migration within the class of the core affinity.

3. AMC ARCHITECTURE

We propose an AMC architecture that mixes cores belonging to the following classes or bins:

1. Class A: high power, high ILP, complex predictors, vector execution units (e.g. SSE/MMX), large L2 cache;
2. Class B: medium power and ILP, vector execution units, medium L2 cache;
3. Class C: low power and ILP, small L2 cache; and
4. Class D: special purpose cores (media codecs, encryption, I/O processor).

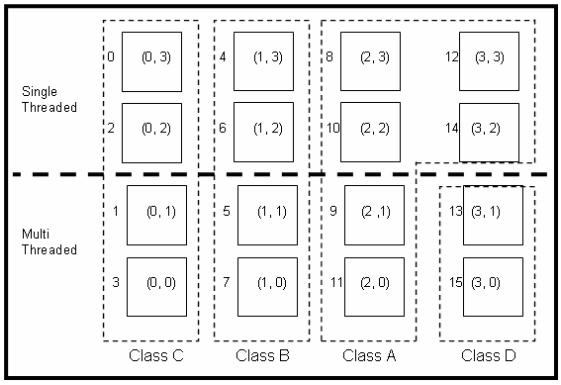

Fig. 1. AMC Architecture

It is assumed that cores are interconnected by a 2D mesh interconnect with Manhattan-style routing. Fig. 1 shows the 16 core AMC architecture with its 4 classes. Each of the first 3 bins is divided in half between single threaded processors (with even numbers) and multi-threaded processors (with odd core numbers). A 4-bit ID identifies a core and the least significant bit identifies it as singled threaded or multi-threaded. By mixing cores with various power requirements and computational capabilities, it is intended to maximize the probability of good mapping of wide workloads into the AMC's cores. Note that cores in the lower classes (e.g. C) miss some of the functionality of cores in the higher classes (e.g. A). Contrary to what is pictured in Fig. 1, the core areas of the various classes are unequal and class A cores occupy much larger areas that class C cores. Needless to say, when a thread is scheduled for the 1st time, if the thread only requires a single threaded class C core (cores 0 and 2) and neither is available, then the scheduling algorithm (that we'll discuss in the next section) will select an available core in the nearest higher class possible, and specifically in the following order: multi-threaded class C (cores 1 or 3), single-threaded class B, multi-threaded class B (cores 5 or 7), single-threaded class A (cores 8, 10, 12, or 14), and finally multi-threaded class A (cores 9 or 11). Note that class D cores (13 and 15) have special functions and normal threads are not mapped to them but special operations are assigned to class D cores. However if a thread requiring a multi-threaded class B core finds none to be available, then the scheduler will attempt to schedule it to a single threaded class A core if one is available. If none are available, the scheduler cannot make an assignment to an available core in the lower class C as these do not support some required functionality (e.g. vector units) and so the thread is requeued and not scheduled. On a 2nd or later attempt to schedule a thread, the scheduler attempts to assign a thread to run on the core on which it ran the last time it got scheduled thus satisfying the core affinity of the thread to minimize inter-core thread state update overhead penalties.

In mesh-connected AMCs, it is desirable to schedule cooperating threads to as close cores as possible in order

to minimize the communication time. The distance form $core_i$ to $core_j$ is given by

$$Distance(core_i, core_j) = |x_j - x_i| + |y_j - y_i| \quad (1)$$

where $core_i$= CPU core i's number, and its 2D coordinates $(x_i, y_i)$are given by

$$x_i = \lfloor core_i / 4 \rfloor$$

$$y_i = (core_i \bmod 4)$$

$$\text{if } y_i \in \{0, 3\} \text{ then } y_i = (y_i + 3) \bmod 6 \quad (2)$$

As these may involve three costly divisions, it is desirable to create a table of inter-core distances for each core that includes cores in the same class or in higher classes. For instance for core 5, 1-hop cores include cores in {7, 6, 9}, 2-hop cores include core in {4, 11, 10}, 3-hop cores include cores in {8}.

4. SCHEDULING SCHEME

Scheduling algorithms attempt to deliver schedules which optimize metrics such as maximum throughput, minimum response time, minimum waiting time, or maximum CPU utilization [17]. Several algorithms exist such as shortest job first, round robin, etc, each with its advantages and drawbacks. Since tasks have different priorities, some that need urgent attention while others have more tolerance for waiting, it makes sense for the scheduling algorithm to be priority-based. While optimal schedules are desired, it is also important to avoid excessive data collection and intensive schedule computation in order to keep the scheduling overhead time under control. This means that near-optimal schedules are acceptable.

In a priority-based scheduling scheme, each thread is assigned a priority either by the programmer or the operating system. Several scheduling queues exist one for each priority. The scheduler attempts to schedule threads waiting in the highest priority queue 1 first, followed by those in priority queue 2, etc. Priority-based schedulers can cause starvation for the lower priority threads, and starvation avoidance policies can be enforced to remedy these situations. Some options are enforcing aging which increases the priority of threads as time progresses thus each queued thread will eventually reach highest priority if not scheduled, or allocating time slices to each queue which distributes according to its policy its allotted time slices among its threads, this way no queue will be left behind. Reducing the time slice increases the number of context switches which can improve more threads' chances to progress at the expense of a larger total context switch overhead time. Our scheduling algorithm is centralized and preferably runs on the same (class C) core. The scheduler maintains the structures of Fig. 2, the CPU Core Assignment Board (CCAB) which holds information of which core or logical processor (if multithreaded) is busy, and the Thread Board (TB) which contains thread relevant information including: the thread's state, *previous_CPU* (PC) or *core affinity* (CA) which holds the core number on which the thread ran last, *good_fit* which indicates if the core assignment is good (1) or can be improved (0), *Thread Affinity* (TA) which indicates the desire to be in proximity to the core hosting thread TA (ideally on the same core but on different logical processor), the thread's priority, and its class which reflects it core functionality and power requirements. Note that PC depends on the thread scheduling history and has nothing to do with the programmer, while TA may be intentionally specified by the programmer, or by the operating system -- if unspecified by the programmer-- as the operating system knows which threads collectively belong to the same process and thus may benefit by running together.

CPU Core Assignment Board (CCAB)

Core #	LP0	LP1
Core 0	1 (occupied)	
Core 1	0 (available)	1 (occupied)
…		
Core 8	1 (occupied)	
Core 9	1 (occupied)	1 (occupied)
…		

Thread Board (TB)

Thread #	State	Previous_ CPU (PC) / Core Affinity (CA)	Good_ fit	Thread Affinity (TA)	Priority	Class (core and power requirements)	System Time when Queued Last
0	Sleeping	4	1	0	1	A	91099223
1	Active	9	0	1	3	B	91110277
2	Waiting	5	1	2	2	C	86583234
…	…						

Fig. 2. Relevant Structures

Fig. 3 presents our proposed scheduling algorithm for the AMC architecture. Initially all queues are cleared and relevant structures are also cleared. The algorithm goes by each queue starting from highest to lowest priority and schedules each thread to its *previous_CPU* in order to satisfy core affinity if the *previous_CPU* assignment is a good fit (same class), or if the class to which *previous_CPU* is not utilized by more than *upc* % where *upc* is initialized to 75% and can be later changed by the operating system. If the *previous_CPU* is unavailable or belongs to a not-best-fit class with over 75% utilization, the thread may have to migrate to a core nearer (defined by equations (1) and (2)) to the *previous_CPU* in the same class, or if one is unavailable to a core nearer to *previous_CPU* in the next higher class available. If no such cores are available in the same class or all higher classes, the thread is requeued in the same priority queued it was popped off thus implementing Round Robin policy within the same priority level. When a core is assigned to the thread, a thread entry is queued into the dispatch queue for that core so that it can be executed.

Note that priority inversion is avoided by scheduling first from higher priority queues. So there is no chance for slower priority threads to make faster progress than the higher priority ones except possibly temporarily when the lower priority threads have been starved and their priority has been temporarily boosted by the operating system.

In order to maintain fairness, our scheduler implements the following fairness policy. Periodically each $t_{UpdatePriority}$ time slices, a timer triggers an alarm which boosts the priority of all queued tasks that have not been scheduled for the past $t_{starvation}$ time quanta by 1 priority level in all queues of priority 2 and above. The parameters $t_{UpdatePriority}$ and $t_{starvation}$ can be initially set to 3 and 9 and can be adaptively fine tuned by the operating system or manually by the system administrator. This way all lower priority threads will eventually reach priority 1 level and Round Robin policy in that level assures that they will get scheduled. This policy requires that the system time when the thread was last queued to be stored in the Thread Board (Fig. 2).

-Initialization:
For all i
{clear queue$_i$}
For all j
{set all *previous_CPU*$_i$ to *none* and all *good_fit*$_{ij}$ to 0}
Set upc to 75% //utilization of class to which previous CPU belongs

-Global scheduling algorithm:
While not all queues empty {
For i=1 to num_queues {
While queue$_i$ is not empty {
Pop thread$_k$ from queue$_i$ and assign thread$_k$ to run on core number (in the following decreasing preference order)
-*previous_CPU*$_k$ **if** available **and** (*good_fit*$_{ik}$=1 **or** (*good_fit*$_{ik}$=0 **and** *utilization* of class of *previous_CPU*$_k$ < upc))
// no migration
-another available core in thread$_k$'s class such that it is *nearest* to the core corresponding to thread$_k$'s *thread_affinity*$_k$ and set *good_fit*$_{ik}$ to 1
-*nearest* available core in nearest higher class and set *good_fit*$_{ik}$ to 0
//else if all above cores are unavailable then just requeue
If thread is not completed then requeue thread$_k$ at end of queue$_i$
// implements Round Robin in same priority
}
}
}
-Fairness policy of global scheduler:
Each $t_{\text{Update Priority}}$ time quanta, in all priority queues at priority 2 and above, boost the priority of all queued tasks that have not been scheduled for the past $t_{\text{starvation}}$ time quanta by 1 priority level.

-Local dispatch:
Pop thread from core's queue and run it on available logical processor in core.

Fig. 3. Scheduling Algorithm for AMC

	cycle 1							Priority		cycle 2						
Thread #		12	11	10	3	2	1	0	1			11	10	3	2	1	0
Class		D	D	B	A	A	A	A				D	B	A	A	A	A
TA		11	11	10	0	0	0	0				11	10	0	0	0	0
PC/CA		-	-	-	-	-	-	-				13	4	14	12	10	8
17	16	15	9	8	7	6	5	4	2	17	16	15	5	4	9	8	7	6
B	B	B	A	A	A	A	A	A		B	B	B	A	A	A	A	A	A
17	15	15	6	6	6	6	4	4		17	15	15	4	4	6	6	6	6
-	-	-	-	-	-	-	-	-		7	5	6	11	9	-	-	-	-
						18	14	13	3							18	14	13
						C	A	A								C	A	A
						18	14	13								18	14	13
						-	-	-								0	-	-
									. . .									
	cycle 3							Priority		cycle 4						
Thread #				10	3	2	1	0	1				10	3	2	1	0
Class				B	A	A	A	A					B	A	A	A	A
TA				10	0	0	0	0					10	0	0	0	0
PC/CA				4	14	12	10	8					4	14	12	10	8
			7	6	5	4	9	8	2			9	8	7	6	5	4
			A	A	A	A	A	A				A	A	A	A	A	A
			6	6	4	4	6	6				6	6	6	6	4	4
			11	9	11	9	-	-				11	9	11	9	11	9
						18	14	13	3							14	13
						C	A	A								A	A
						18	14	13								14	13
						0	-	-								-	-
									. . .									
	cycle 5							Priority		cycle 6						
Thread #							10	1									
Class							B										
TA							10										
PC/CA							4										
			5	4	9	8	7	6	2							5	4
			A	A	A	A	A	A								A	A
			4	4	6	6	6	6								4	4
			11	9	11	9	11	9								12	14
							14	13	3							14	13
							A	A								A	A
							14	13								14	13
							-	-								-	-
									. . .								

Fig. 4. Contents of the Priority Queues

To illustrate how this scheme works, we go over an example. For simplicity, the illustration assumes that all threads request single threaded cores and that $t_{\text{starvation}}$ is 8 so the fairness policy is not involved in this particular example. At the start of scheduling cycle 1, 19 threads are waiting in 3 priority queues to be scheduled into the 16 core as shown in cycle 1 of Fig. 4. Each thread entry in these queues holds information that includes the thread number, its class, thread affinity (TA), and *previous_CPU* or *core affinity* (PC, CA). The algorithm pops the queues and allocates threads 0-3 onto cores 8, 10, 12, and 14, thread 10 to core 4, threads 11-12 to cores 13 and 15, threads 4-5 to cores 9 and 11, threads 15-16 to cores 6 and 5, thread 17 to core 7, and thread 18 to core 0 as shown under the time slice 1 column of the Gantt chart of Fig. 5. Note that for simplicity, we omit showing delays due to schedule computations. Also note how threads 1 and 2 are assigned to cores 10 and 12, only 1 hop away from core 8 to which thread 0 is assigned in order to meet thread affinity requirements of threads 1 and 2. For the same reason, thread 16 is assigned to core 5, the nearest available core in class B to core 6. After running, the thread entries are requeued onto their respective queues in the same order if the threads do not finish execution in this scheduling cycle as long as they are not blocked. In this example, all these threads are requeued except for thread 12 which terminates execution at the end of cycle 1 as indicated under the time slice 2 column in Fig. 5. Threads 6-9 and 13-14 do not get scheduled in this cycle and remain in the queues as they request class A core none of which is available.
Cores 1-3 remain idle as there is no sufficient demand for class C cores. At the start of scheduling cycle 2, the queue contents are as shown in cycle 2 of Fig. 4. The main difference in this cycle is that the order of waiting threads in the priority 2 queue is different from the previous cycle as not all threads in this queue got

Thread #	Time slice 1	Time slice 2	Time slice 3	Time slice 4	Time slice 5	Time slice 6	Time slice 7	Time slice 8
0	8	8	8	8	Finished			
1	10	10	10	10	Finished			
2	12	12	12	12	Finished			
3	14	14	14	14	Finished			
4	9			9	14	14	Finished	
5	11			11	12	12	Finished	
6		9			9	Finished		
7		11			11	Finished		
8			9		10	Finished		
9			11		8	Finished		
10	4	4	4	4	4	Finished		
11	13	13	Finished					
12	15	Finished						
13						8	8	Finished
14						10	Finished	
15	6	6	Finished					
16	5	5	Finished					
17	7	7	7	Finished				
18	0	0	0	Finished				

Fig. 5. Example's Gantt Chart

scheduled in the previous cycle. Round Robin policy schedules now threads 6-7 to cores 9 and 11 as shown in Fig. 5 while thread 4-5, 8-9, and 13-14 wait in their queues. In Cycle 3, (time slice 3) it is now the turn of threads 8-9 to be scheduled onto cores 9 and 11. In cycle 4, it is now the turn of threads 4-5 to be rescheduled onto cores 9 and 11. Note how *previous_CPU* information is used by the scheduler to satisfy core affinity, and how TA is used to schedule threads near the core running the thread corresponding to their thread affinity. In cycle 5, threads 6 then 7 get scheduled first so they are assigned to their *previous_CPU*'s 9 and 11. Thread 8 then get scheduled and is assigned core 10, the nearest available core in class A to core 9 to which thread 8's TA (thread 6) is assigned. For the same reason, threads 9, 4 and 5, get assigned to cores 8, 14, and 12 respectively. In cycle 6, threads 4-5 get assigned to their *previous_CPU*'s 14 and 12, leaving room for cores 8 and 10 to be assigned to threads 8 and 9. Thread 13 gets assigned to core 8 in cycle 7 and completes at the end of cycle 7.
The produced schedule is optimum with respect to the throughput (but is priority-based) and the CPU utilization given the available resources, the states and contents of the starting queues, and the thread core requirement constraints.

5. SIMULATION EXPERIMENTS AND ALGORITHM EVALUATION

In this section, we study the effectiveness of this scheduling scheme compared to non-migratory scheduling schemes and other schemes which allow migration within the class of the core affinity. For simplicity, we assume that each class in the AMC architecture has only multithreaded cores and no general core is single-threaded. We also assume that once a thread migrates to a new core, its *good_fit* is 1 or its CPU utilization is always lower than *upc*. In other words, the first choice of the scheduling algorithm is always the previous core, *previous_CPU*, on which the thread ran in the last quantum in which it was active. We also assume that $t_{UpdatePriority}$ and $t_{starvation}$ are very large such that the fairness policy is never involved. It is also assumed that if a thread is scheduled to a core, that core remains idle and unavailable to other threads until the in-transit thread assigned to it starts on it. This core only opens up to the other threads upon completion of 1 cycle –time slice-- by the assigned thread. In order to quantify the benefits of thread affinity, which seeks to schedule a thread to the same core in which it ran in the last time quantum it was active, and thread migration, which allows the scheduling of a thread to a core different from the core on which it ran in the last time quantum it was active due to the unavailability of this latter core, we consider and compare the following six scheduling algorithms.

A. **NAM** (No Affinity- Migration allowed): a variation of the proposed algorithm with no concept of affinity which attempts to schedule the thread within its class if possible according to a list of increasing core numbers, and then looks for an available core in the next higher class, following a sequential order of increasing core number;

B. **NAMWC** (No Affinity- Migration allowed Within Class): a variation of A except that the scheduler attempts to schedule the thread within its class following a sequential order of increasing core number, and if no cores are available within the same class, it does not seek to schedule the thread to an available core in the next higher classes but requeues the thread to the end of the priority queue.

C. **AMNN** (Affinity- Migration allowed according to Nearest Neighbor): a small variation of the proposed algorithm as the one proposed in the previous section except that if the previous CPU is not available then the scheduler looks to schedule the thread to cores in the vicinity of *previous_CPU* (and not the TA as in Fig. 3) and if this is not possible, it schedules on the nearest available core in the next higher classes;

D. **AML** (Affinity- Migration allowed within List in increasing core number order): same as NAM except that the scheduler attempts first to schedule the thread to the same *previous_CPU* core if available.

E. **AMWC** (Affinity- Migration allowed Within Class): same as NAMWC except that the scheduler attempts first to schedule the thread to the same *previous_CPU* core if available; and

F. **ANM** (Affinity- No Migration): a non-migratory scheduling algorithm that only attempts to reschedule an unfinished thread to the same *previous_CPU* core if available. Otherwise, it requeues the thread to the end of the priority queue if the *previous_CPU* core is unavailable.

For that purpose, a model of the thread scheduling algorithm and its queue infrastructure was developed in the C programming language in Microsoft Visual .net Studio 2003. A time slice after which the operating system scheduler starts a new scheduling cycle is assumed to consume a time duration which we refer to as 1 *cycle.* Thus one *cycle* in this section refers to one time slice or tens of processor cycles. Threads are assumed to be very light weight threads with very short durations and even shorter switching times. For simplicity, it is assumed that thread migration from a core to another core adjacent to it, referred to by 1 hop, takes CPH (Cycles Per Hop) cycles to complete, where CHP varies between 1-2 *cycles*. Using this terminology, thread migration from core 8 to core 1 in Fig. 1 will take 4 hops or (4 x CPH) *cycles* to complete. At the start of each of the 100 simulation runs, for each of the 3 priority queues, random number generating functions are called to generate: i. the number of tasks in each queue, ranging from 0 to 20; ii. the duration of each task, ranging from 1 to *dur cycles*, where the maximum thread duration, *dur*, is allowed to vary from 3, 6, 9, 18, 30, 120, 480, up to 960 *cycles*; and iii. the *previous_CPU* core number of the thread, ranging from 0 to15;

For each of the threads, two associated numbers are initialized to 0, the completion time of the thread, and the penalty in *cycles* or hops incurred due to thread migration from the start of the simulation run at time 0 till the time when the thread fully completes execution. Simulation then proceeded as in Figures 3-4 until all threads in all 3 priority queues finished execution, updating in each run the number of *cycles* it took for all the threads to complete execution and the total number of penalties (in hops or *cycles*) incurred by all migrating threads. Note that the total number of *cycles*, TNOC, represents the time in *cycles* of the last thread that completed its run and that the completion times of the other threads overlap with TNOC. The total number of penalty *cycles*, TNOP, is accumulative and adds the penalty *cycles* incurred by all threads. In the next simulation run, the number of threads in priority queues and their characteristics are randomly generated as described above and the procedure repeats until all 100 simulation runs each representing a different ensemble of threads' scenarios complete. The final TNOC it took for all threads to complete after the 100th and final run, adds up all the *cycles* from all 100 runs and is accumulative. The final TNOP incurred by all threads in all 100 simulation runs adds up also the individual penalty *cycles* from each simulation run and is also accumulative. Next, we present the number of *cycles* per simulation run, the average number of penalty *cycles* per simulation run, for all 6 algorithms, and for various CPH and *dur* values. Note that the averages are the total numbers of *cycles* divided by 100, the total number of simulation runs. The TNOP for the ANM algorithm is 0 as this algorithm is non-migratory and reassigns the thread to its *previous_CPU* whenever available so no migration-related penalty *cycles* are incurred. Precisely, we plot the normalized ratios $TNOC_{algorithm}/ TNOC_{AMNN}$ and $TNOP_{algorithm}/ TNOP_{AMNN}$ for all 6 algorithms.

1-Cycle Per Hop

Fig. 6 plots the TNOCs for all 6 algorithms normalized to the TNOC of the AMNN scheduling algorithm for the case of a 1-*cycle* hop duration. The x coordinate is *dur* in *cycles*. When CPH is 1, AMNN is best for *dur* of 3 and 9.

For *dur*>=18, there is no notable difference in the performance in total number of *cycles* between the Affinity algorithms AMNN, AML and AMWC. Non-affinity algorithms NAMWC and NAM perform worse than AMNN by 13%-38% but relatively improve in performance with increasing *dur*. Non-migratory algorithm ANM performs worse than AMNN by 3.6%-22.5% and degrades further with increasing *dur*. It is clear that thread affinity and migration are both helpful to the total schedule completion time.

Fig. 7 plots the normalized TNOPs for all 6 algorithms normalized to the TNOP of the AMNN scheduling algorithm for the case of a 1-cycle hop duration. ANM is best with no penalty *cycles* followed by AMWC which limits migration within the same class thereby containing migration costs, followed by AMNN, and AML, respectively. AMWC (AML) generates fewer and fewer penalty *cycles* as *dur* increases, and handily beats AMNN in that domain when *dur*>=9 (480). The non-affinity algorithms NAM and NAMWC generate 2.5x-13.8x AMNN's penalty *cycles* with increasing number of penalty cycles with increasing *dur*. It is important for the reader to keep in mind that the TNOP *cycles* overlap with the schedule completion time and are not the ultimate decider of the best scheduling algorithm but help in comparing them with respect to migration costs. For instance, algorithm ANM performs the worst under large *dur* values but yet incurs the fewest penalty *cycles* among all 6 algorithms.

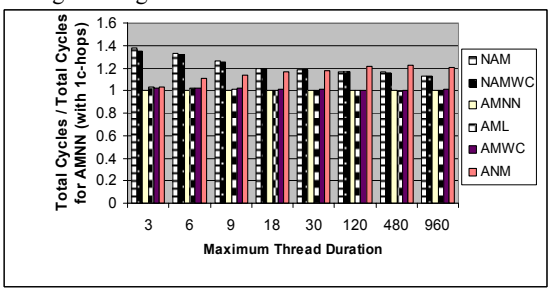

Fig. 6. TNOCs Normalized for AMNN with CPH=1

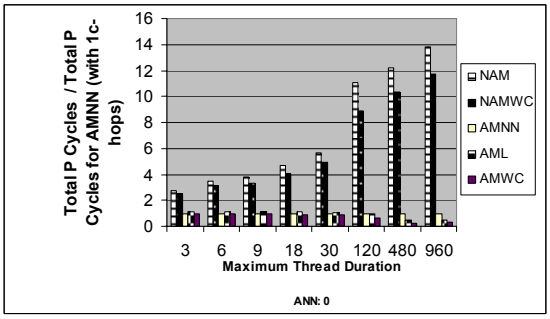

Fig. 7. TNOPs Normalized for AMNN with CPH=1

2-Cycle Per Hop

Inter-core communication can slow down due to a variety of reasons including longer distances and wires, higher resource contention, or bigger traffic and longer wait times. Fig. 8 plots the normalized TNOCs for all 6 algorithms normalized to the TNOC of the AMNN scheduling algorithm for the case of a 2-*cycle* hop duration. When inter-core communication slows down and the duration of a hop is increased to 2 *cycles*, AMNN is best for a *dur* in the 9-30 range, followed respectively by AML and AMWC. It is also observed that AMNN is very competitive for a *dur* of above 30. For small *dur* values of 6 *cycles* or below, ANM is best (2.7%-17% better than AMNN) followed by AMNN in the second place. Short duration tasks with longer migration penalties seem to favor the Affinity but non-migratory scheme. For *dur*>=120 cycles, AMWC and AML are best (2-3% better than AMNN) followed by AMNN which remains competitive. A quick look at Fig. 9 reveals that this is attributed to more penalty *cycles* generated by AMNN than AMWC or AML when *dur*>=120. Higher contention to the *previous_CPU* cores of AMNN (as compared to AMWC or AML) is the most logical reason for the larger penalty cycles generated by AMNN. In other words, when *previous_CPU* is unavailable, scheduling a new core following a list of increasing core numbers seems to reduce contention slightly more to scheduling a new core in the vicinity of *previous_CPU* as achieved by AMNN, with the randomly generated thread scenarios of our simulation experiments. As for the worst performing algorithms, the non-Affinity algorithms NAM and NAMWC are 35%-62.7% worse than AMNN but improve in relation to AMNN as *dur* increases. For *dur* values of 30 or above, ANM performs 14%-16% worse than AMNN.

Fig. 9 plots the normalized TNOPs for all 6 algorithms normalized to the TNOP of the AMNN scheduling algorithm for the case of a 2-*cycle* hop duration. ANM still generates 0 penalty *cycles*. When *dur* <=30, AMWC generates 3%-4% fewer penalty *cycles* than AMNN. As expected the most penalty *cycles* are generated by the non-affinity algorithms NAM and NAMWC which generate 2x-7x more penalty *cycles* than AMNN. The algorithms generating the fewest penalty *cycles* are AMWC and ANM when *dur* is 120 or above, generating 1/3-1/2 of AMNN's penalty *cycles*.

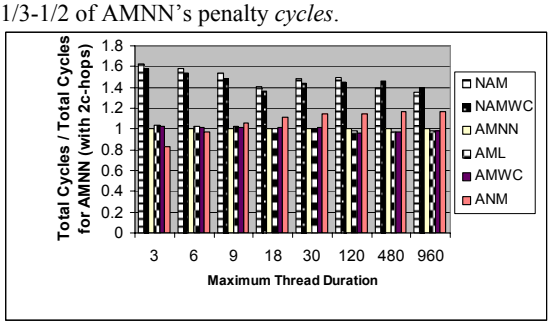

Fig. 8. TNOCs Normalized for AMNN with CPH=2

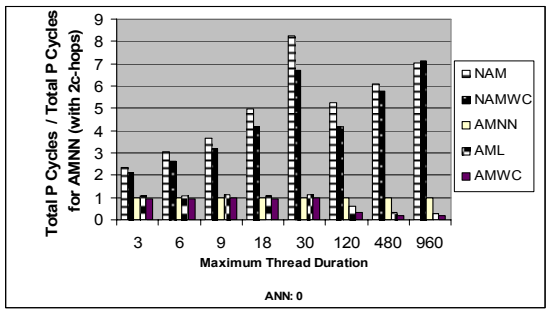

Fig. 9. TNOPs Normalized for AMNN with CPH=2

Effect of Hop Duration

Fig. 10 displays compares the TNOPs as CPH is doubled from 1 to 2 *cycles*. As ANM is non-migratory the hop duration has no effect on its performance. This ratio is highest for non-affinity algorithms NAMWC & NAM. In the Affinity algorithms, it is observed that the $TNOC_{CPH=2}/TNOC_{CPH=1}$ ratio goes down as *dur*

increases. This is because longer thread durations dilute the increases in hop duration and communication time. Non-Affinity algorithms are most sensitive to doubling the CPH. Increasing the hop duration is most detrimental to the Non-Affinity algorithms which very likely incur migration costs on every thread re-scheduling, costs which become heftier with longer hop durations.

Fig. 11 compares the TNOPs for the same scheduling algorithms as CPH is doubled from 1 to 2 *cycles*. Not surprisingly, the trend of the $TNOP_{CPH=2}$ / $TNOP_{CPH=1}$ ratio is often increasing with increasing *dur*. The longer the hop duration, the higher the total number of incurred penalty *cycles*. This ratio is highest for AMNN when *dur* >=120. Starting with a *dur* value of 30 cycles, the Affinity algorithms appear to be the most sensitive to doubling the CPH, and in particular AMNN, which attempts to keep the thread in the neighborhood of its *previous_CPU* as much as core availability permits.

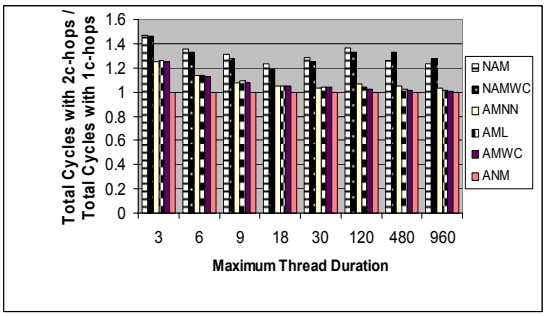

Fig. 10. Effect of Doubling the Hop Duration on TNOC

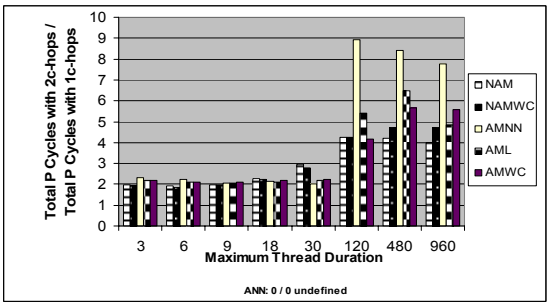

Fig. 11. Effect of Doubling the Hop Duration on TNOP

6. CONCLUSIONS

We presented a 16 core asymmetric multi-core architecture comprised of 4 core classes. Our AMC architecture combines high-power complex cores with large L2 caches to low-power low-ILP cores with small L2 caches and a few special purpose cores in the same chip. We also presented a priority-based scheduling algorithm for this AMC architecture. Our algorithm addresses priorities of threads, and incorporates a fairness policy to avoid thread starvation. It also attempts to schedule threads to their previous cores if possible to minimize state migration overhead time, and when not available to cores nearest to their thread affinities in their requested class if available, or in the nearest class where a core is available. As such, it maximizes throughput and core utilization.

We developed a C simulation model which simulates the thread scheduling on the 16-core architecture and considered 6 scheduling algorithms. Simulation results revealed that the proposed affinity- and migration-based nearest neighbor scheduling algorithm which considers both thread affinity and thread migration in its scheduling decisions outperforms the other algorithms for small thread durations. For large thread durations, affinity- and migration-based scheduling algorithms outperform the non-affinity algorithms and the non-migratory algorithm, but there is insignificant difference in the performance of the affinity- and migration-based algorithms. In that case, core selection policy, be it nearest neighbor, within a class, or across classes, makes little difference. As for the worst performing algorithms, when CPH is 1 and for small thread durations, or when CPH is 2 irrespective of *dur*, non-affinity algorithms perform worse than non-migratory ones. When CPI is 1 and *dur* is large, non-migratory ones perform worse than non-affinity algorithms.

This scheduling scheme can be combined with cache partitioning and cache fairness policies [14, 16] that partition the L2 cache memory and other shared resources adequately and fairly among the co-scheduled threads. Future work includes fine tuning the scheduling scheme's parameters with real workloads and exploring other thread schedule scenarios.

6. REFERENCES

1. P., Dubey, CMP Challenges, ICCD Panel, *IEEE Int. Conf. on Computer Design*, 2005.
2. S. Balakrishnan et al., The Impact of Performance Asymmetry in Emerging Multicore Architectures, Proc. of *32nd ISCA*, 2005.
3. R. Kumar et al., Heterogeneous Chip Multiprocessors, *IEEE Computer*, 2005.
4. F. N. Sibai, Dissecting the PCMark05 Benchmark and Assessing Performance Scaling, *Proc. of 3rd IEEE Conf. on Innovations in Info. Tech.,* 2006.
5. L. Hammond et al., A Single-Chip Multiprocessor, *IEEE Computer*, Volume 30, No. 9, 1997.
6. R. Kalla et al., IBM POWER5 Chip: A Dual-Core Multithreaded Processor, *IEEE Micro*, 2004.
7. P. Kongetira et al., Niagara: A 32-way Multithreaded SPARC Processor, *IEEE Micro*,2005.
8. C. McNairy and R. Bhatia, Montecito: A Dual-Core, Dual-Thread Itanium Processor, *IEEE Micro*, 2005.
9. T. Constantinou et al., Performance Implications of Single Thread Migration on a Chip Multi-Core, *ACM Comp. Architecture News,* Vol. 33 (4), 2005.
10. K. Shaw et al., Migration in Single Chip Multiprocessors, *Comp. Arch. Letters*.Vol. 1, No. 3, 2002.
11. H. Abdel-Shafi et al. Efficient User-Level Checkpointing and Thread Migration in Windows NT Clusters, *3rd Usenix Windows NT Symp.*, 1999.
12. R. Avnur et al, Thread Migration in the River Dataflow Environment, University of California Berkeley CS Dept. Technical Report.
13. S. Hily et al., Contention on 2nd Level Cache May Limit The Effectiveness of Simultaneous Multithreading, *INRIA Report # 1086,* 1997.
14. S. Kim et al., Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture, *Proc of the 13th IEEE PACT04*, 2004.
15. D. Chandra, et al., Predicting Inter-thread Cache Contention on a Chip Multi-Processor Architecture, *Proc of the IEEE HPCA-11*, 2005.
16. A. Fedorova et al., Cache-Fair Thread Scheduling for Multicore Processors, Technical Report TR-17-06, Harvard University, 2006.
17. A. Silberschatz et al, *Operating Systems Concepts*, John Wiley and Sons, 2004.

Received: Feb. 2008. Accepted: Sep. 2008.