Task Scheduling for Many-Cores with S-NUCA Caches

Anuj Pathania, Jörg Henkel
Chair of Embedded System (CES), Karlsruhe Institute of Technology, Germany
Corresponding Author: anuj.pathania@kit.edu

***Abstract*—A many-core processor may comprise a large number of processing cores on a single chip. The many-core's last-level shared cache can potentially be physically distributed alongside the cores in the form of cache banks connected through a Network on Chip (NoC). Static Non-Uniform Cache Access (S-NUCA) memory address mapping policy provides a scalable mechanism for providing the cores quick access to the entire last-level cache. By design, S-NUCA introduces a unique topology-based performance heterogeneity and we introduce a scheduler that can exploit it. The proposed scheduler improves performance of the many-core by 9.93% in comparison to a state-of-the-art generic many-core scheduler with minimal run-time overheads.**

I. Introduction

A many-core processor may comprise a large number of processing cores on a single chip [1]. The cores along with their private caches and a bank (or slice) of shared Last-Level Cache (LLC) are physically distributed all over the many-core. A core also contains a router through which it can communicate with other cores using a Network on Chip (NoC) and reach Dynamic Random-Access Memory (DRAM) controllers on the periphery for getting access to the main DRAM memory. Figure 1 shows an abstract block diagram of the many-core with L2 cache being the last-level cache and the cores arranged in a 2D lattice pattern.

Since the last-level cache is physically distributed, access to any of its banks for any of the cores does not have uniform access latency. Therefore, such caches are called Non-Uniform Cache Access (NUCA) caches. The NUCA caches are designed primarily for NoC-based many-cores as the bus-accessed physically consolidated uniform access caches used in multi-cores do not scale up well in many-cores [2].

Several memory-to-cache address mapping policies have been proposed for many-cores [3]. Static-NUCA (S-NUCA) is a fixed address mapping policy for NUCA caches, wherein mapping of memory addresses to NUCA cache banks is done statically at design-time; generally interleaved over the available cache lines. S-NUCA due to its rigidity is not as efficient at run-time as the Operating System (OS) managed flexible address mapping policies such as Dynamic-NUCA (D-NUCA). S-NUCA on the other hand can be efficiently implemented in the hardware independent of the OS with very low overheads [4].

Figure 2 shows how the NUCA cache accesses are distributed amongst the different S-NUCA cache banks in the different cores of a 64-core many-core when an instance of four-threaded *streamcluster* benchmark is executed on the many-core in isolation. We observe that the execution results in access of all the NUCA cache banks of the many-core without any regard to which core the threads are pinned. This NUCA cache access pattern is also not fixed for a given task in a multi-program execution as the NUCA cache banks accessed by the task not just depends upon interaction of the task with memory but also on the interactions of other tasks with the memory. This makes profiling or predicting the frequency of NUCA cache accesses of a task across all the NUCA cache banks difficult in practice [5].

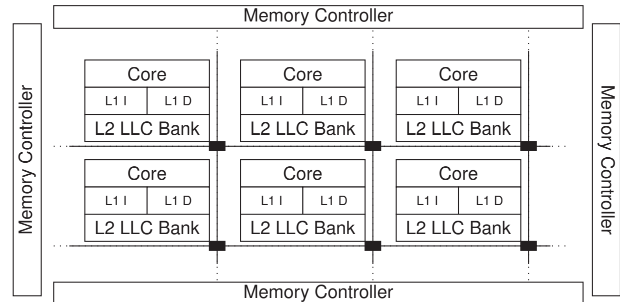


Fig. 1: An abstract block diagram of a many-core with L2 cache being the physically distributed LLC.

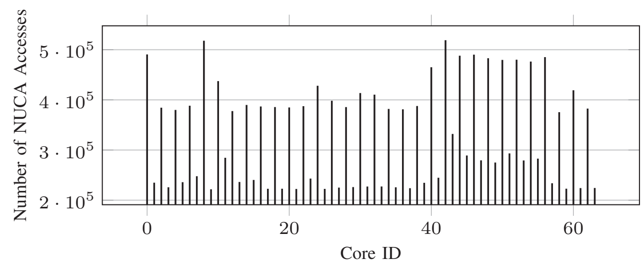


Fig. 2: Distribution of NUCA cache accesses under S-NUCA for an isolated execution of four-threaded *streamcluster* benchmark amongst cores of a 64-core many-core.

The common approach for mapping a task's threads on a many-core is to allocate them in a spatially compact square-like shapes [6]. This approach is inspired from similar resource allocation problems in grid computing [7], where focus is to minimize the network hops resulting from the shapes of the resource allocations on a 2D-grid [8]. Authors in [9] have shown that this approach can be successfully extended to many-cores with D-NUCA caches. Unfortunately, this approach of

978-3-9819263-0-9/DATE18/©2018 EDAA

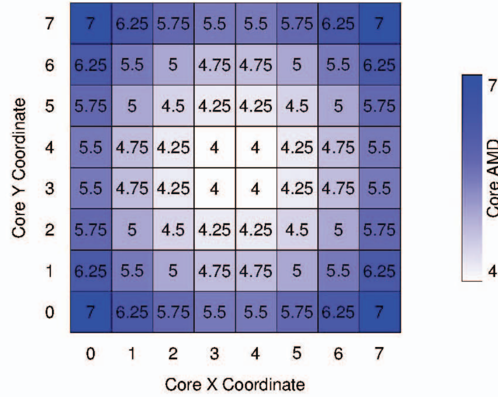


Fig. 3: The design-time AMDs associated with different cores of a 64-core (8x8) many-core.

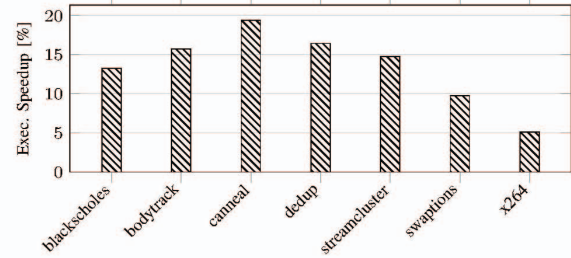


Fig. 4: The observed speedups in execution time of four-threaded instances of different benchmarks in isolated executions when their four threads are pinned across cores with minimum AMD in comparison to when they are pinned across cores with maximum AMD of a 64-core (8x8) many-core.

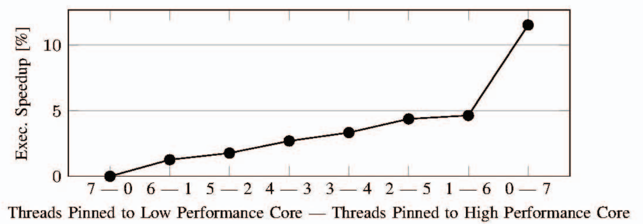


Fig. 5: The observed speedups in execution time of eight-threaded instances of *streamcluster* benchmarks in isolated executions when their seven slave threads are pinned in different combinations on high-performance (low AMD) cores and low-performance (high AMD) cores.

compacting the shape of resource allocations does not work on the many-core with S-NUCA caches, where optimizing for physical proximity within the threads from the same task can potentially have no observable benefits.

Even though it is difficult to determine the set of cores to pin a task's threads under S-NUCA that can potentially result in the best performance for the task, the probability that a core on an average would result in a higher performance is higher if a core has a lower Average Manhattan Distance (AMD) associated with it than its peers. AMD of the core is defined as the average of all rectilinear distances between the core and every other core in the many-core. This behavior under S-NUCA introduces an inherent design-time performance heterogeneity in different cores of the many-core based on their spatiality even if all the cores themselves are perfectly homogeneous. A scheduler oblivious to this heterogeneity would inadvertently make poor thread mapping decisions.

Our Novel Contributions: We characterize the performance heterogeneity in cores of a many-core with S-NUCA caches while executing multi-threaded workloads. Based on our observations, we introduce a scheduler called *SNSched* that exploits this performance heterogeneity to extract more performance from the many-core in comparison to a state-of-the-art generic many-core scheduler while executing multi-threaded multi-program workloads.

II. Performance Characterization for Many-Cores with S-NUCA caches

We begin by characterizing the performance of multi-threaded tasks on a many-core with S-NUCA caches. Observations made in this section will form the foundation for the design of our proposed scheduler in the next section.

By topological design, not all cores of a many-core are equidistant from each other and inherently some of the cores have lower AMDs than the others. Figure 3 shows AMDs of all cores in a 64-core (8x8) many-core. It can be observed from the figure that the cores closer to the center by design have lower AMD then the cores farther away. Figure 4 shows the speedups obtained for four-threaded instances of different benchmarks in isolated executions when all their threads are pinned to the cores with the lowest AMD of 4 vis-a-vis when all their threads are pinned to the cores with the highest AMD of 7. Results show that all the tested benchmarks complete execution significantly faster on the cores with lower AMD compared to the cores with higher AMD.

It is also important that all the concurrently executing threads of a multi-threaded task which synchronize over a barrier experience near-equal performance, otherwise the slowest thread will become a bottleneck limiting the speedup gains. This behavior can be observed in master-slave thread design of *PARSEC* [10] benchmarks which represent tasks from embedded and high-performance computing domains. In a *PARSEC* benchmark, a master thread is invoked first which then spawns multiple slave threads. Once all the slave threads finish, only then the master thread terminates itself to complete the benchmark execution. Figure 5 shows the speedups observed in eight-threaded instances of *streamcluster* benchmark when seven of its slave threads are distributed between the low-performance (high AMD) cores and the high-performance (low AMD) cores in different combinations. Figure 5 shows that there is a statistically significant jump in the speedup for the benchmark only when all its slave threads are executed on the high-performance (low AMD) cores simultaneously.

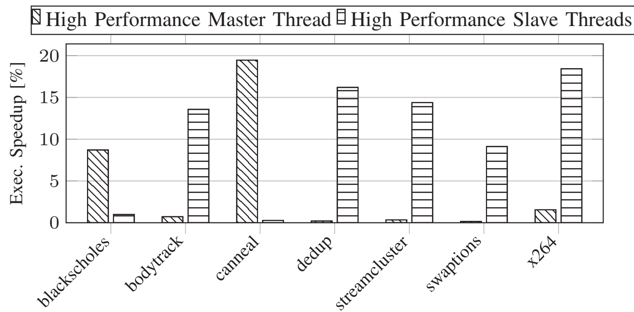


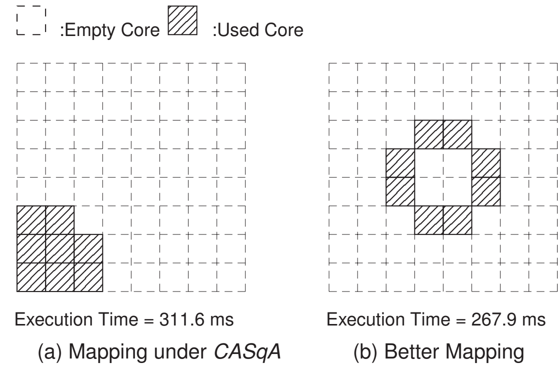
Fig. 6: The execution speedups in four-threaded instances of different benchmarks in isolated executions when their master thread is pinned on low-performance core or their three slave threads are pinned on low-performance cores against a baseline when all their threads are pinned on low-performance cores.

The relationship between the single master thread and multiple slave threads is of a different nature but is equally important. The master and slave threads are joined together in a sequential relationship. Speeding up only any one of the two can potentially restrict the overall speedup of the benchmark. The part which is not sped up becomes the bottleneck limiting the gains as described by Amadhl's law [11]. Figure 6 shows the speedups of different four-threaded instances of benchmarks when only their master thread is placed on the high-performance cores and when only their slave threads are placed on the high-performance cores. Speedup is measured against the baseline performance when all their threads are placed on the low-performance cores. It can be seen from Figure 6 that the benchmarks perform differently based upon the importance of the master thread and slave threads in their overall performance. Amongst the seven tested benchmarks capable of producing four-threaded instances, *blackscholes* and *canneal* benchmarks are most sensitive to the performance of the master thread. All the remaining benchmarks are most sensitive to the performance of the slave threads.

Summary: Based on our observations, we conclude that performance of the cores is negatively correlated to their topology induced AMDs. We also conclude that to prevent performance degrading bottlenecks, it is best to pin all the slave threads of a multi-threaded benchmark to the cores that have the same or near-similar AMD. Depending upon the sensitivity of the master and slave threads to the overall performance they can be mapped on cores with different AMDs. It is best to map all the master and slave threads to the cores with near-similar AMD if the sensitivity information is not available to prevent any bottleneck formation.

III. Shortcoming of State-of-the-art on Many-Cores with S-NUCA Caches

CASqA [12] is a state-of-the-art generic many-core scheduler when operating with rigid tasks executing under a one thread per-core model [13]. A rigid task's thread mapping must be determined before it starts execution and the mapping cannot be modified using thread migration thereafter.



(a) Mapping under *CASqA* (b) Better Mapping

Fig. 7: An example showing a better performing thread mapping for an eight-thread instance of *Streamcluster* benchmark on an idle 64-core many-core with S-NUCA caches in comparison to a thread mapping assigned under state-of-the-art generic many-core scheduler *CASqA*.

CASqA attempts to map the threads of an incoming rigid task in a contiguous square-like shape around an initial node selected using a stochastic hill climbing algorithm. Furthermore, it prefers mapping close to edges of the many-core to minimize the fragmentation which can reduce potential for future compact contiguous mappings. *CASqA* allows spatial contiguity to be broken when there are not enough contiguous cores available to perform a square-like non-contiguous allocation to improve throughput by increasing system utilization. The relaxation in the contiguity can be adjusted using a user-defined parameter; but we only cover *CASqA* version that permits unlimited non-contiguous solutions in this work. Furthermore, *CASqA* is designed to operate with the tasks that come with a task-graph. We modify *CASqA* to operate with the master-slave applications such as *PARSEC* benchmarks.

We now show the short-comings of *CASqA* on a many-core with S-NUCA caches using an illustrative example in Figure 7. Figure 7a shows how an eight-threaded instance of *streamcluster* benchmark could be possibly mapped by *CASqA* in a square-like shape on an idle 64-core many-core using its stochastic algorithm. Figure 7b shows an alternative thread mapping more suitable for the many-core with S-NUCA caches for eight threads of the *streamcluster* benchmark in the same scenario. Experiments show that *streamcluster* benchmark's execution time decreased by 14.02% under the non-compact non-contiguous thread mapping shown in Figure 7b in comparison to the compact contiguous thread mapping shown in Figure 7a.

The performance gain can be explained by the observations made in Section II. The performance of *streamcluster* benchmark in both mappings is determined by its slowest performing thread (bottleneck thread), which in turn is determined by the core with highest AMD allocated to the benchmark. The highest AMD of a core allocated in the thread mapping shown in Figure 7a is 7 whereas all the cores allocated in the thread mapping shown in Figure 7b have the same AMD value of 4.25. Hence, *streamcluster* benchmark instance executes faster

with the thread mapping shown in Figure 7b than the thread mapping shown in Figure 7a.

IV. S-NUCA Aware Many-Core Scheduler

We present a scheduler called *SNSched* for scheduling of rigid tasks on a many-core with S-NUCA caches. *SNSched* uses a classic Branch and Bound (BnB) algorithm [14] to determine the thread mapping for an incoming task on the many-core. A complete BnB algorithm would have been computationally infeasible on the many-core at run-time because of the large problem size and NP-hard complexity of the thread mapping problem under consideration [15]. Still, we can efficiently deploy it in this work because observations made in Section II substantially reduce the search-space. Without our observations, each core of the many-core is potentially a unique design point for the BnB algorithm to evaluate. With the help of our observations, we can classify the cores into classes based on their AMDs. For example, 64 cores of the many-core shown in the Figure 3 can be divided into 9 unique classes - 4 to 7 - based on their AMDs. Since all the cores in each class are potentially equivalent for *SNSched* in respect to their performance potential, the worst-case search depth of BnB algorithm is reduced to 9 from 64. The BnB depth of 9 is still computationally feasible at run-time on the many-core.

System Model: We assume an open system [16], where random rigid tasks arrive into the system for execution at non-initial time with uniform distribution. The tasks arriving in the arrival queue of the open system are executed based on a First-In-First-Out (FIFO) policy. Let N be a core requirement of a task in front of the queue. Since the many-core operates with one thread per-core model, the task in our system can only be mapped when the number of free cores in the system is more than the task's core requirement N.

Let the cores of the many-core be classified into a set of classes C based on their AMDs indexed by C_i with $|C_i|$ representing the number of free cores available of the class C_i. Let $\alpha(C_i)$ represent a numerical AMD value associated with cores of the class C_i.

Let M be a set of subsets of C indexed by M_j representing all possible thread mappings using different combinations of classes in C that can satisfy the core requirement N.

$$M_j \in M \iff \left(\sum_{C_i \in M_j} |C_i|\right) \geq N \tag{1}$$

The set M theoretically contains all the possible thread mapping solutions that can satisfy the task in the front of the queue though in practice we do not need to enumerate them all at run-time. We still need to guarantee that we find the best amongst them. We measure a quality of mapping using two different functions. The first function is represented by the symbol ρ which captures the performance of a mapping. Based on the observations made in Section II, performance of the mapping is negatively correlated to the AMD of a core with highest AMD amongst all cores contained in the mapping.

$$\rho(M_j) = 1/\max_{C_i \in M_j} \alpha(C_i) \tag{2}$$

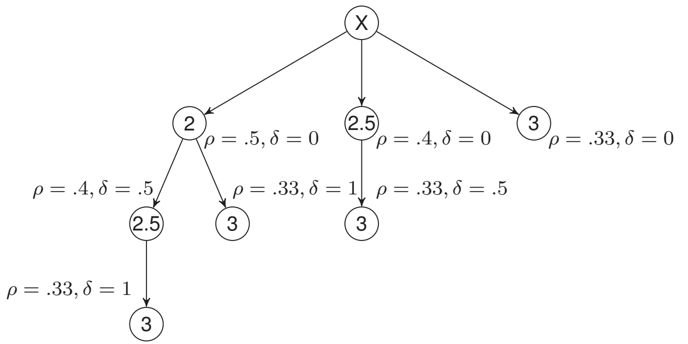


Fig. 8: A search-space tree for a 16-core (4x4) many-core with 3 unique AMD-based core classes.

The second function represented by the symbol δ captures the dispersion (fragmentation) caused by the thread mapping. It is mathematically defined as the difference between the maximum and minimum AMD of cores contained in the mapping. The dispersion δ has a value zero if the cores contained in the mapping are all from the same class.

$$\delta(M_j) = \max_{C_i \in M_j} \alpha(C_i) - \min_{C_{i'} \in M_j} \alpha(C_{i'}) \tag{3}$$

We choose to prioritize the performance ρ over dispersion δ. Therefore, we give preference to a task ready for execution over a task that is yet to be executed. An advance heuristic can also only partially prioritize the performance ρ over dispersion δ to derive more overall system performance. The performance $\rho(M_j)$ and dispersion $\delta(M_j)$ default to values $-\infty$ and ∞, respectively if the thread mapping M_j is empty. Under *SNSched*, the mapping M_j is called superior to another mapping $M_{j'}$ if former has either higher performance or has lower dispersion with the same performance than the latter.

$$M_j > M_{j'} \iff \rho(M_j) > \rho(M_{j'}) \vee \big(\delta(M_j) < \delta(M_{j'}) \wedge \rho(M_j) = \rho(M_{j'})\big) \tag{4}$$

Algorithm: We now present the BnB algorithm used in *SNSched*. The algorithm's goal is to find the best thread mapping for an incoming task with the core requirement N given the thread mappings of the tasks which are already scheduled. Figure 8 shows the search-space tree of an empty 16-core (4x4) many-core with cores divided into three unique AMD-based classes – 2, 2.5 and 3. The value of a node in the search-tree represents the numeric value of the AMD class being explored for containment in a thread mapping solution. The path to the node represents a thread mapping solution (partial or complete) in which at least one core from the class represented by the node is used and at least one core from the class represented by all its parent is also used. Each node in the search-space tree shown in Figure 8 is accompanied by a respective value of the performance function ρ and dispersion function δ if the thread mapping solution is to include the node and its parents but not its children.

In the search-space tree shown in Figure 8, by design the child nodes always go from lower AMD values to higher

Algorithm 1 Proposed algorithm called *SNSched*.

Input: *SNSched* $(N, M^{\#})$;
Output: M^*; ▷ M^* is a global variable initialized to NULL.
1: Stack $S = \{\}$;
2: **for** $C_i \in C\ \nabla \alpha(C_i)$ s.t. $|C_i| \neq 0 \wedge \alpha(C_i) > \max_{C_{i'} \in M^{\#}}$ **do**
3: Push C_i into S;
4: **end for**
5: **while** S is not empty **do**
6: $M^{\#} = M^{\#} \cup$ Pop S;
7: **if** $\rho(M^{\#}) > \rho(M^*) \vee \big(\delta(M^{\#}) < \delta(M^*) \wedge \rho(M^{\#}) = \rho(M^*)\big)$ **then**
8: **if** $\sum_{C_i \in M^{\#}} |C_i| \geq N$ **then** ▷ A valid best solution yet.
9: $M^* = M^{\#}$;
10: **else**
11: *SNSched* $(N, M^{\#})$; ▷ Recursively search deeper for a solution.
12: **end if**
13: **end if** ▷ Search depth bounded by no further recursive exploration.
14: Pop $M^{\#}$
15: **end while**
16: return M^*;

AMD values. The AMD of the parent node is always lower than all its children nodes in the search-space tree. As a result, the performance ρ always decreases and dispersion δ always increases as we traverse down the search-space tree. Therefore, the search for a better solution in the search-space tree can be bounded by not exploring a node's children if the performance ρ of the node falls below a known solution of higher performance (Equation 4). The search can also be bounded if the performance of the node is same as the previous best performing solution but has higher dispersion δ associated with it than the best performing solution.

Algorithm 1 shows the pseudo-code for *SNSched* that uses a Depth First Search (DFS) to find a solution (M^*) through recursively extending a partial solution ($M^{\#}$), which gets bounded by the value of the performance ρ function and dispersion δ function as soon as the first valid solution is found. Only AMD classes that have at least one free core need to be enumerated in the search-space tree. The lower performing cores from the class with a higher AMD in M^* are allocated first to the incoming task till N cores are allocated if the solution M^* has more than N cores.

V. Empirical Evaluation

We need to evaluate multi-threaded multi-program workloads on a many-core with full modeling of shared resource contentions to demonstrate the efficacy of *SNSched*. *Sniper* [17] interval simulator allows for such evaluations under reasonable time-constraints. *Sniper* uses S-NUCA by default when simulating NUCA architectures. We simulate a 64-core many-core with a 2D NoC with cores arranged in an 8x8 grid. The NoC is configured to use XY routing and has a latency of 4 cycles per-hop with a link bandwidth of 256 bits per-cycle. Each core is an out-of-order core with *Intel Gainestown* microarchitecture and *x86-64* Instruction Set Architecture (ISA). Each core has a 4-way associative 16 KB L1 instruction and data cache. A 16-way associative 2 MB S-NUCA cache acts as a Last Level Cache (LLC) with each core holding 32 KB slice of the physically distributed LLC. All caches use Least Recent Used (LRU) page replacement policy and are kept coherent using Modified Shared Invalid

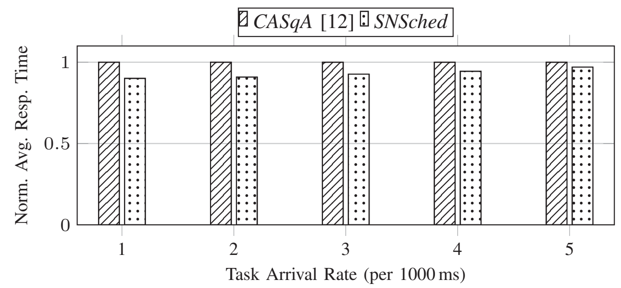


Fig. 9: Performance of a 64-core many-core with S-NUCA caches in an open system with a multi-program workload arriving at different arrival rates under different schedulers.

(MSI) protocol. The hit latencies for L1 cache and LLC are set at 3 cycles and 8 cycles, respectively. Each core also contains a dedicated NoC router. An off-chip 1 GB DRAM acts as the main memory with hit latency of 80 cycles. There are 4 DRAM controllers providing main memory access; one on each side of the many-core.

PARSEC [10] multi-threaded benchmarks are used as software. We execute *PARSEC* benchmarks with *sim-small* inputs as the *sim-small* inputs are large enough to meaningfully stress the caches of simulated many-core but small enough for the multi-program simulations to finish in a reasonable time. We use 9 out of the 13 benchmarks of *PARSEC* benchmark suite – *blackscholes*, *bodytrack*, *canneal*, *dedup*, *fluidanimate*, *streamcluster*, *swaptions*, *x264* – to create workloads comprising of twenty randomly selected benchmark instances. Two benchmarks *freqmine* and *vips* were discarded due to the unresolvable binary instrumentation errors while using *PIN* tool in *Sniper*. Two benchmarks *facesim* and *raytrace* were not used due to lack of *sim-small* input for them making them unreasonably large in comparison to other benchmarks.

We implemented the S-NUCA aware scheduler *SNSched* and its comparative state-of-the-art baseline scheduler *CASqA* in *C++* by forking the original code of the *pinned* scheduler shipped with *Sniper*. Both schedulers operate at a granularity of 10 ms; same default granularity as the *Linux* scheduler [18].

Performance: We execute same multi-program workloads on a 64-core many-core with S-NUCA caches deployed in an open system under the two schedulers *CASqA* and *SNSched* to evaluate their efficacies. The workloads are repeated with different values of arrival rate parameter to simulate different levels of system load. The load induced by an open workload on the many-core increases with the increase in the arrival rate. The performance of a scheduler is measured by the average response time experienced by the tasks composing the workload when managed by the scheduler. Lower average response time reflects higher scheduler performance.

Figure 9 shows the performance under *SNSched* is superior to *CASqA* under all loads and *SNSched* can results in up to 9.93% increase in the performance. We observe that performance gains under *SNSched* over *CASqA* decrease with the increase in the system load. As the system's load increases, the low-performance cores even under *SNSched* must be allocated

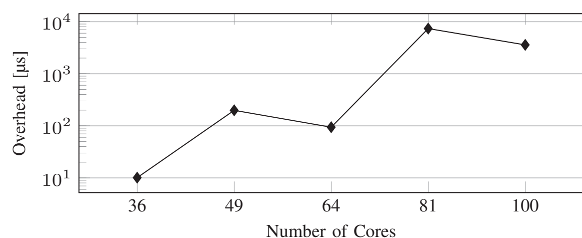


Fig. 10: The measured worst-case task scheduling overheads for *SNSched* on varisized many-cores.

to prevent system under-utilization and hence potential to improve the overall average response time decreases. The experiments corroborate our claim that a scheduler aware of S-NUCA design can improve the performance of the many-core with S-NUCA caches.

Overhead: Figure 10 shows the *worst-case* scheduling overheads of *SNSched* in real-world time for 6x6 36-core many-core to 10x10 100-core many-core on a logarithmic scale. The average overhead in real-world would be much smaller than the reported worst-case overheads. The worst-case problem-solving time of *SNSched* on a 64-core (8x8) many-core is a minuscule 94 μs translating into a worst-case overhead of just .09% at run-time for a scheduling epoch of 10 ms. The worst-case problem-solving time rises sharply on an 81-core (9x9) many-core to 7364.1 μs translating into an unsustainable worst-case overhead of 73.64% at run-time. Scheduling using light-weight heuristics [19] would be more suitable for the 81-core many-core than the BnB algorithm used in this work. This supports our argument that *SNSched* is practical on the 64-core many-core but still it maybe not be fast enough on many-cores of larger size. The overhead of *SNSched* on the 64-core many-core is less than the overhead on a 49-core many-core because of the smaller search-space as the number of unique AMD classes in an 8x8 many-core is 9 against 10 in a 7x7 many-core.

VI. Conclusion

In this work, we proposed a scheduler called *SNSched* for task scheduling on a many-core with S-NUCA caches. We characterized the performance heterogeneity introduced in the cores of the many-core by executing multi-threaded workloads on them. We then presented a BnB algorithm that enables *SNSched* to exploit this heterogeneity for extracting up to 9.93% more multi-program performance in comparison to a state-of-the-art generic many-core scheduler on a 64-core many-core. A proof-of-concept simulation predicts that *SNSched* will operate efficiently in practice on the 64-core many-core with negligible 0.09% worst-case scheduling overhead. In future, we will extend *SNSched* to also incorporate the next layer of design-time performance heterogeneity introduced in the cores of the many-core due to the presence of multiple memory controllers in the many-core [20].

Acknowledgment

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

References

[1] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hübner, R. K. Pujari, A. Grudnitsky, J. Heisswolf, A. Zaib, B. Vogel *et al.*, "Invasive Manycore Architectures," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2012.

[2] R. Balasubramonian, N. P. Jouppi, and N. Muralimanohar, "Multi-Core Cache Hierarchies," *Synthesis Lectures on Computer Architecture*, 2011.

[3] S. Das and H. K. Kapoor, "Exploration of Migration and Replacement Policies for Dynamic NUCA over Tiled CMPs," in *International Conference on VLSI Design (VLSID)*, 2015.

[4] H. Kim, P. Ghoshal, B. Grot, P. V. Gratz, and D. A. Jiménez, "Reducing Network-on-Chip Energy Consumption Through Spatial Locality Speculation," in *International Symposium on Networks-on-Chip (NOCS)*, 2011.

[5] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry *et al.*, "Scheduling Threads for Constructive Cache Sharing on CMPs," in *Symposium on Parallel Algorithms and Architectures (SPAA)*, 2007.

[6] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: survey of current and emerging trends," in *Design Automation Conference (DAC)*, 2013.

[7] M. A. Bender, D. P. Bunde, E. D. Demaine, S. P. Fekete, V. J. Leung, H. Meijer, and C. A. Phillips, "Communication-Aware Processor Allocation for Supercomputers: Finding Point Sets of Small Average Distance," *Algorithmica*, 2008.

[8] E. D. Demaine, S. P. Fekete, G. Rote, N. Schweer, D. Schymura, and M. Zelke, "Integer Point Sets Minimizing Average Pairwise L1 Distance: What is the Optimal Shape of a Town?" *Computational Geometry*, 2011.

[9] A. Pathania, V. Venkataramani, M. Shafique, T. Mitra, and J. Henkel, "Defragmentation of tasks in many-core architecture," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2017.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[11] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Spring Joint Computer Conference*, 1967.

[12] M. Fattah, P. Liljeberg, J. Plosila, and H. Tenhunen, "Adjustable Contiguity of Run-Time Task Allocation in Networked Many-Core Systems," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014.

[13] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y.-h. Dai *et al.*, "Corey: An Operating System for Many Cores," in *Operating Systems Design and Implementation (OSDI)*, 2008.

[14] E. L. Lawler and D. E. Wood, "Branch-and-Bound Methods: A Survey," *Operations research*, 1966.

[15] E. D. Demaine and M. L. Demaine, "Jigsaw Puzzles, Edge Matching, and Polyomino Packing: Connections and Complexity," *Graphs and Combinatorics*, 2007.

[16] D. G. Feitelson and L. Rudolph, "Metrics and Benchmarking for Parallel Job Scheduling," *Job Scheduling Strategies for Parallel Processing*, 1998.

[17] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[18] V. Pallipadi and A. Starikovskiy, "The Ondemand Governor," in *The Linux Symposium*, 2006.

[19] E. Carvalho, N. Calazans, and F. Moraes, "Heuristics for Dynamic Task Mapping in NoC-based Heterogeneous MPSoCs," in *International Workshop on Rapid System Prototyping (RSP)*, 2007.

[20] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis, "Handling the problems and opportunities posed by multiple on-chip memory controllers," in *Parallel Architectures and Compilation Techniques (PACT)*, 2010.