Online Scalability Characterization of Data-Parallel Programs on Many Cores

Younghyun Cho Surim Oh Bernhard Egger

Department of Computer Science and Engineering
Seoul National University, Seoul, Korea
{younghyun, surim, bernhard}@csap.snu.ac.kr

ABSTRACT

We present an accurate online scalability prediction model for data-parallel programs on NUMA many-core systems. Memory contention is considered to be the major limiting factor of program scalability as data parallelism limits the amount of synchronization or data dependencies between parallel work units. Reflecting the architecture of NUMA systems, contention is modeled at the last-level caches of the compute nodes and the memory nodes using a two-level queuing model to estimate the mean service time of the individual memory nodes. Scalability predictions for individual or co-located parallel applications are based solely on data obtained during a short sampling period at runtime; this allows the presented model to be employed in a variety of scenarios. The proposed model has been implemented into an open-source OpenCL and the GNU OpenMP runtime and evaluated on a 64-core AMD system. For a wide variety of parallel workloads and configurations, the evaluations show that the model is able to predict the scalability of data-parallel kernels with high accuracy.

1. INTRODUCTION

With the ongoing consolidation of more and more cores into a single chip, many-core multiprocessor systems are now readily available. While graphics processors have been integrating over a thousand cores for some time now, these days modern general-purpose servers easily integrate a few tens or even hundreds of cores. Machines with such a large number of cores typically comprise multiple physical CPU chips integrated in a multi-socket multi-core non-uniform memory architecture (NUMA) system. At the same time, data-parallel programming models such as OpenCL [20] promising for easy and scalable parallelization have risen in popularity, initially primarily on GPGPUs but recently also on modern general-purpose processors [32, 33, 31]. While in principle data parallel programs scale linearly, architectural realities such as the memory bottleneck limit the scalability as the number of available computing cores increases.

An important problem for runtime systems managing tens or hundreds of cores is allocation of processing cores to parallel applications. The state-of-the-art, static resource reservations or serializing parallel applications and devoting all physical cores to the running application, is not utilizing the computing resources to maximum capacity anymore. To exploit the full potential of the hardware, several parallel applications need to be executed simultaneously. In this context, the main question is how many of the available resources should be allocated to each application in order to achieve maximal system throughput. Scalability prediction models of parallel applications can provide resource managers with the necessary information to make these choices.

Predicting the performance scalability of parallel applications has long been an active topic of HPC research [4, 34, 41, 7, 36, 5, 22]. Sophisticated performance modeling techniques are used to aid application design and performance tuning for HPC systems. Existing techniques for at-runtime many-core management rely on additional efforts before applications are executed such as offline training [25] or machine learning [15, 14] to understand the applications' performance characteristics. If offline information is not available, the standard practice of prior research [27, 9] is to measure an application's throughput by executing it on different resource configurations to obtain information about its scalability. Such techniques not only require additional hardware resources but also a long reconfiguration time. Furthermore, existing techniques can not or only with great difficulty be used to predict the scalability of co-located parallel applications.

In this paper, we propose an accurate, at-runtime performance scalability model for data parallel programs running on commodity NUMA systems. The model is eventually to be employed by the resource scheduler of a runtime system for many-core systems which puts stringent constraints on the tolerable overhead of the model. Furthermore, we target malleable parallel programming models that can change the degree of parallelism during execution of a parallel section. The OpenCL model with work units distributed to worker cores naturally allows adaptation of the number of assigned cores. The standard OpenMP `parallel for` construct with its static even work distribution at the beginning of the loop does not provide malleability within a parallel section. Ways to render OpenMP applications malleable have been proposed [16].

The proposed model does neither require prior analysis, offline profiling, nor modifications of the parallel applications. The model is integrated directly into the parallel runtime, and the performance characteristics of the individ-

PACT '16, September 11 - 15, 2016, Haifa, Israel

ISBN 978-1-4503-4121-9/16/09...$15.00

DOI: http://dx.doi.org/10.1145/2967938.2967960

ual parallel sections are obtained at runtime. The negligible overhead of the brief sampling phase querying hardware performance counters renders the model an ideal candidate to be used in online resource allocation. Furthermore, the model is able to estimate an application's performance scalability in the presence of external workloads competing for the shared resources.

In the data parallel programming model, work units are independent of each other and synchronization only occurs at the end of a parallel section. The main limiting factor of scalability is thus memory contention. Reflecting the architectural design of modern many-core architectures with several cores sharing a last-level cache (LLC), termed `compute node`, and one or more NUMA memory nodes, contention is modeled both at the LLCs of the individual compute nodes and the memory nodes.

Modern NUMA systems by the major vendors for x86 hardware exhibit similar NUMA interconnects with AMD's HyperTransport (HT) and Intel's QuickPath Interconnect (QPI), respectively. Both vendors provide hardware performance counters that allow us to measure relevant NUMA-related performance events. We measure memory accesses per core, misses in the LLCs, and memory requests arriving at the different memory controllers during a brief sampling period initiated when the program enters a parallel section. The throughput of an application in dependence of the number of assigned cores is modeled based on the system-wide workload and memory contention. The presented model consists of a hierarchy of two stacked queuing models to compute contention at the individual memory controllers (MC) and the buses between the last-level caches and the MCs. Scalability predictions are made based hardware performance counters gathered at-runtime during a short sampling period. The model can adapt to different NUMA memory allocation policies and does not require any offline analysis of the data parallel code. This and the negligible overhead of the sampling make the model an ideal candidate for at-runtime resource management and performance tuning.

The model has been implemented into an open-source OpenCL and the GNU OpenMP runtime. OpenCL [20] is an obvious choice as it is one of the most prominent data parallel programming models these days. For OpenMP [10] code we only support the data-parallel `parallel for` loop constructs. The accuracy of the model is validated against measurements on a 64-core AMD multi-socket multi-core NUMA system with various OpenCL and OpenMP kernels. The model is evaluated both by predicting the scalability of standalone applications under different NUMA allocation policies and in co-located parallel applications scenarios. The experimental results show that the model is able to accurately predict the speedup of a data parallel kernel with an error of 6.8% and a coefficient of determination (R^2) of 0.9 for OpenCL and an error of 9.7% and R^2 of 0.88 in OpenMP benchmarks.

The remainder of this paper is organized as follows. Section 2 provides the necessary background on programming models and queuing theory. The scalability prediction model is detailed in Section 3. Sections 4 and 5 describe the implementation on the target architecture for validation and evaluation of the proposed model. Section 6 discusses related work. The paper concludes with Section 7.

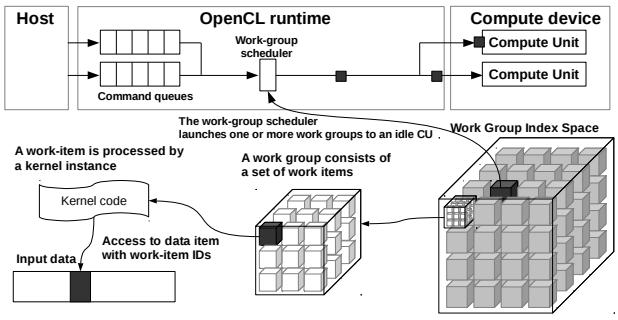


Figure 1: The OpenCL execution model.

2. BACKGROUND

2.1 Data Parallelism in OpenCL

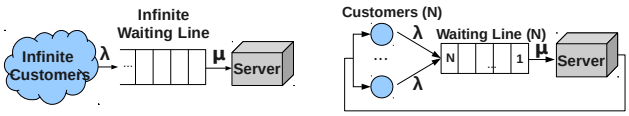
OpenCL [20] is one of the most popular data parallel programming models. An OpenCL application consists of a *host program* and one or several *kernels*. The kernels implement the data-parallel parts of the program while the host program provides initialization and glue between the parallel parts. Via the OpenCL runtime, the host processor dispatches work to the devices, each of which comprises one or more *compute units* (CUs). Each CU is divided into one or several *processing elements* (PEs) within which the computations on a device occur. In OpenCL terminology, a *work-item* refers to the smallest unit in a data decomposition. The minimal execution unit is a *work-group* comprising one or more work-items. All work-items within a work-group are executed concurrently on the PEs of a single CU. Figure 1 shows one possible data composition of a three-dimensional index space into work-groups and work-items.

Following the data parallel model, no synchronization between work-groups is provided. Synchronizations within work-groups and the synchronization of CUs at the end of the entire computation do not invalidate the assumptions of our model since work-groups constitute the smallest execution unit and are executed independently. An OpenCL scheduler can increase or decrease the number of processing CUs while executing a kernel. Performance scalability can thus be a useful indicator for at-runtime resource management and performance tuning for OpenCL programs.

In the context of many-core x86 processors, each processor core constitutes an OpenCL CU. That is, the work-items of a work-group are executed on one core. Most x86 OpenCL runtimes execute work-items serially to avoid the task switching overhead, but concurrent (task switched) execution is also possible. The proposed model is independent of the execution model since the CU is the smallest modeled entity.

2.2 Parallel Loops in OpenMP

OpenMP [10] is widely used for parallel programming on shared-memory systems. Based on the fork-join model, OpenMP provides a variety of functionalities (e.g., synchronization, nested parallelism, etc.) to ease parallel programming. Although OpenMP is not a data parallel programming model per-se, parallel loops in OpenMP executing different parts of the entire workload on assigned cores in parallel exhibit similar properties as data parallel models. In the OpenMP `parallel for` construct, there is no synchro-



(a) M/M/1 system (b) M/M/1/N/N system

Figure 2: Queuing systems (λ=arrival rate, μ=service rate).

nization or communication between loop iterations except implicit barriers at the end of the parallel loop. Application programmers can annotate the scheduling discipline of the loop and choose between static, dynamic, and guided scheduling. OpenMP parallel loops are not directly malleable with a static work distribution, but predicting the scalability of such loops can be useful if the application is executed more than once. In addition, several researches [16, 8] have presented techniques to render OpenMP programs malleable.

2.3 Queuing Models

Queuing theory is the mathematical study of waiting [35]. For a queuing network, a queuing model can predict the waiting time based on probabilistic methods. Queuing models are widely used for a variety of applications such as telecommunication, market services, or computer systems. In the context of computer systems, queuing models are especially well-suited to model resource contention.

We argue that queuing theory is a good candidate for performance prediction of data-parallel programs for the following reasons. First, there are no data dependencies or synchronization points between the work chunks. Instead, the major bottleneck affecting scalability is contention in the memory system. Second, the individual computing cores processing the chunks of work can be considered independent workers exhibiting similar access patterns to shared resources. These two properties allow us to treat each computing core as a flow-equivalent queuing customer and model memory contention using a queuing model. Employing queuing theory to model memory contention also has the additional benefit of being independent from the NUMA allocation policy as well as allowing us to model the performance of several parallel applications executing simultaneously.

The *M/M/1 model* is one of the simplest and most popular queuing models. An M/M/1 queuing system, illustrated in Figure 2 (a), assumes an infinite number of customers and one single server. The model can estimate the mean service time when input and service rates follow an exponential distribution. For an input rate λ and a service rate μ, $\mu > \lambda$, the expected mean service time is given by Little's Law as

$$r = \frac{1}{\mu - \lambda}$$

With a finite number of customers, the presence of more or fewer cores can have a strong effect on the distribution of arrivals which renders the model ineffective for accurate scalability estimation.

To deal with finite populations, the *M/M/1/N/N model*, also known as the "machine repair problem" can be applied [28, 35]. An M/M/1/N/N queuing system consists of N customers, a waiting line having N entries, and one server as shown in Figure 2 (b). Given an input rate of one customer, λ, and a service rate μ, the mean service time r is given by

$$r = \frac{1}{\mu}\left(\frac{N}{1 - 1/P(0)} - \frac{\mu}{\lambda}\right)$$

where $P(0)$, the probability that no customers request service, is computed as

$$P(0) = \sum_{k=0}^{N} \frac{N!}{(N-k)!}\left(\frac{\lambda}{\mu}\right)^k$$

The M/M/1/N/N model can be used to compute the contention overhead if the access pattern of the customers (the processing cores) to the shared resource (the memory controller) follows an exponential distribution and the access patterns of all involved customers (cores executing the same parallel program) are similar. Tudor *et al.* [39] have shown that parallel workloads in the NAS parallel benchmark [3] implementing the seven dwarfs of HPC [2] do not exhibit bursty memory access patterns unless the workload is small. Data parallel programs typically deal with big amounts of data; we therefore assume that the M/M/1/N/N queuing is adequate for modeling shared memory contention. The results in Section 5 validate this assumption.

3. SCALABILITY PREDICTION MODEL

The presented performance model facilitates the prediction of a data parallel program's scalability in dependence of the allocated number of processor cores. The prediction is based on a two-level M/M/1/N/N model estimating the memory contention at the bus-level as well as at each memory node. The model relies on the properties of data parallel programs, i.e., similar flow, no data dependencies, and no or very little synchronization between processor cores. Every bus from the last-level cache (LLC) of a CPU node to every memory node as well as each memory node are modeled separately. We compute flow-equivalent per-core memory accesses to obtain the contention on buses from the LLC to the memory nodes. Each memory node is treated as a separate queuing server, the customers are the buses arriving from the different LLCs.

We can only claim that the per-core memory access pattern of a data parallel program is stable when all cores behind a shared LLC execute the same kernel. In this paper, our model thus assumes that parallel applications are allocated to cores at the granularity of CPU nodes to avoid sharing of a LLC between several applications. The model parameters are obtained by querying the hardware performance counters during a short sampling period during execution of a parallel program kernel. The model then estimates how the program's performance changes in dependence of a varying number of CPU nodes. The proposed model is able to estimate memory contention and predict scalability of a parallel program in the presence of co-located parallel applications as long as each application is executed on disjoint CPU nodes.

3.1 Mapping NUMA System onto a Queuing System

The majority of all NUMA systems share the architectural characteristics that are important to map the proposed

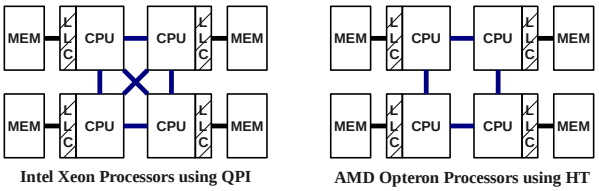


Figure 3: Intel and AMD NUMA systems.

model to a concrete system. CPU cores are grouped into so-called *CPU nodes*. All cores in a CPU node share a common last-level cache (LLC). Each CPU node is connected to its local and several remote memory nodes via a high-speed interconnection network. Figure 3 displays the organization of typical Intel and AMD NUMA systems. The Intel system on the left comprises four CPU chips, each with a local LLC and one memory controller. The CPU chips are connected by a full crossbar network. The AMD system uses a slightly different interconnection network where remote memory accesses can require up to two hops. The interconnection network allows simultaneous accesses from the LLCs to the different memory nodes. That is, the cores behind an LLC can issue memory requests independently and in parallel, but several simultaneous accesses to one memory node may get serialized by the LLC, resulting in a stall of the waiting core(s). Each memory node has a dedicated memory controller which also serializes and processes the incoming requests from the connected compute nodes.

The left hand of Figure 4 illustrates that contention can occur at two different locations: first, on the bus from one of the LLCs to a memory controller. Contention on the bus from one LLC to a given memory node occurs when memory requests by two or more cores behind the same LLC miss and trigger simultaneous requests to the MC that then need to be serialized. Second, at the individual memory controllers themselves. The proposed model reflects this by employing a two-level queuing model to model both bus contention and memory controller contention separately as illustrated on the right-hand side of Figure 4. To model the total contention overhead, we first compute the mean service time observed at each memory controller by using an M/M/1/N/N model. We then add the estimated mean service time of the memory controllers to the bus delay of each memory interconnection bus in an uncongested situation. The bus contention model is then used to compute the mean memory access latency (bus plus memory access latency) based on the measured arrival rates from each core. The following subsection explains the model in detail.

3.2 Scalability Prediction By Contention Modeling

The expected speedup for a number of cores to process a given, specific amount of work is computed as follows. Let W denote the total number of cycles required to complete the given work. In particular, W does not include stall cycles incurred at the two possible contention points. If there is no contention, we can expect linear speedup for a parallel section using N cores and thus divide the total work cycles W by N. Let $C(N)$ be the number of cycles required to serve all memory requests from a core when the core processes a given amount of work with N cores competing for shared

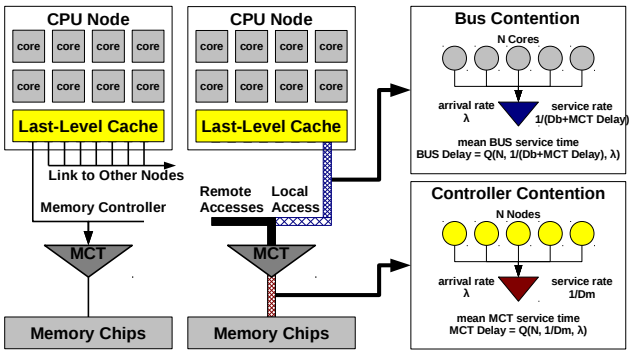


Figure 4: Mapping of NUMA to a queuing system.

Table 1: Model input parameters.

Program-specific parameters	
LLC_m	Per-core LLC miss rate for memory m
BUS_m	Per-core bus access rate for memory m
MCT_m	Per-node memory controller (MCT) access rate for memory m
$D_b[n][m]$	Uncongested bus delay from CPU n to memory m
D_m	Uncongested memory controller delay
Resource allocation	
$N^p = \{n_1^p, n_2^p, ...\}$	Set of reserved CPU nodes for program p
$M = \{m_1, m_2, ...\}$	Set of memory nodes on system
Queuing equation	
$Q(N, \mu, \lambda)$	MM1NN Queuing Equation: N = number of queuing customers, μ = service rate, λ = arrival rate

resources. The speedup is then given by

$$S(N) = \frac{W + C(1)}{W/N + C(N)}$$

To compute $C(N)$, we estimate the mean service time for each individual memory controller. $C(N)$ depends on the number of cores N because of potential contention at the LLCs and the memory nodes. The work per core for N cores (W/N) is multiplied by the average latency to access memory which is the sum of all latencies of requests (mean service request time, R_m) to the different memory controllers, LLC_m. LLC_m denotes the program's per-core miss rate from an LLC to a given memory controller m.

$$C(N) = \frac{W}{N} \cdot \sum_m LLC_m \times R_m$$

The mean service time R_m for memory m is computed by applying the M/M/1/N/N queuing equation as given in Section 2.3. Note that the bus access request rate, denoted BUS_m, differs from LLC_m because the bus access requests include memory write operations, whereas LLC_m only includes (missed) read accesses.

$$R_m = Q(\#\ of\ Cores\ in\ a\ Node, \mu, BUS_m)$$

While all system CPU nodes affect the mean service time of every memory controller, the number of queuing competitors

at the bus from an LLC to a memory controller is equal to the number of cores in a CPU node. The mean service rate μ is given by

$$\mu = 1/(Bus\ Delay + Controller\ Delay_m)$$

where $Bus\ Delay$ represents the mean bus delay without any congestion of the nodes allocated to the parallel application and $Controller\ Delay_m$ denotes the memory controller latency of memory m.

We compute $Bus\ Delay$ by taking the program's average delay of each involved processor node to all memory controllers. N^p contains the list of CPU nodes allocated to program p. The computation of the architectural parameters D_b and D_m is explained in Section 4.3.

$$Bus\ Delay = \frac{\sum_{n \in N^p, m \in M}(D_b[n][m] \times Req\ Ratio[m])}{|N^p|}$$

$$Req\ Ratio[m] = \frac{BUS_m}{\sum_{m' \in M} BUS_{m'}}$$

Finally, we compute the mean memory controller delay for each memory node. Unlike the other resources, memory controller contention can be affected by other applications. We thus compute the average per-node arrival rate λ_m for memory m from all running applications. P represents the set of all running programs in the system.

$$Controller\ Delay_m = Q(\sum_{p \in P} |N^p|, 1/D_m, \lambda_m)$$

$$\lambda_m = \frac{\sum_{p \in P}(MCT_m^p \times |N^p|)}{\sum_{p \in P} |N^p|}$$

The required parameters for computing the model are summarized in Table 1. We obtain the three memory access patterns (LLC_m, BUS_m and MCT_m) from an online profiler built into the parallel program execution runtimes.

3.3 Advantages and Limitations of the Model

Here we briefly discuss advantages and limitations of the proposed model. The major advantage of the model is its efficiency and versatility. The model requires only a very short at-runtime profiling phase to collect work cycles and memory access patterns. It is thus possible to employ the model in dynamic resource management to, for example, allocate cores to several running parallel applications such that the overall performance is maximized. The model is versatile because it is orthogonal to how the data is distributed to the system memories in the system and allows us to estimate speedup even in the presence of co-located workloads.

Nevertheless, the current model suffers from a number of limitations. Foremost, the model only predicts performance scalability on node-granularity. This is because the model requires knowledge about the memory access patterns of each processor node. Furthermore, the behavior of execution contexts in an application is only stable when executed on node granularity. We argue that such an allocation is sensible to prevent pollution of the LLC, however, if scalability needs to be estimated on a finer level than processor nodes, it may be necessary to apply other intra-node scalability prediction techniques independently. Second, our model assumes that the workloads exhibit similar memory access patterns independent of the core they are executed on. This condition is satisfied when a workload scheduler distributes the parallel chunks of work to idle cores at runtime without considering data locality. In general, well-known loop scheduling algorithms such as self or guided scheduling do not consider data-locality. However, if the application employs the first-touch NUMA policy and the work-group scheduler considers data locality when distributing the work, the memory access patterns will vary when executed on a different node. We can extend the model to support such a situation by providing additional information to the application runtime; this is part of future work.

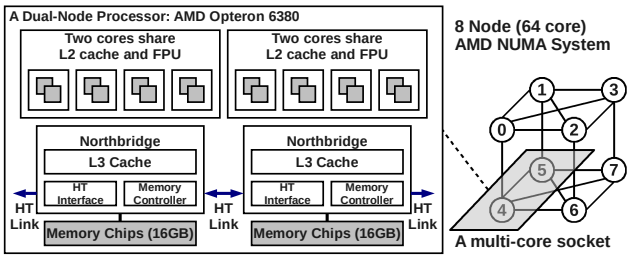


Figure 5: 64-core AMD multi-socket system.

4. MODEL VALIDATION

We validate the proposed model by integrating it into an open-source OpenCL framework and the GNU OpenMP runtime running on an AMD NUMA many-core architecture. This section describes the model's implementation.

4.1 Target NUMA Architecture

The NUMA architecture the proposed model is evaluated on is a 64-core AMD multi-socket NUMA system comprising a total of 8 processor nodes in four physical processor packagings (AMD Opteron 6380 [1]). Each node contains 8 computing cores that share an L3 (last-level) cache and a local memory node. The processor nodes are connected by AMD's hyper-transport with a maximum hop distance of two. Figure 5 shows the organization of the architecture.

4.2 Online Performance Measurement

The at-runtime performance measurement logic and scalability prediction model have been implemented into `SnuCL`, a freely available open-source framework [21] and the GNU OpenMP runtime. The OpenCL runtime profiles each individual OpenCL kernel during a short sampling period at the beginning of a kernel. The OpenMP runtime profiles parallel workloads when a parallel loop is launched.

The following hardware performance counters are measured at runtime (Table 2). On the CPU node, the number of CPU cycles are measured on a per-core basis. LLC miss counts, DRAM requests to the total eight memory nodes, and the local memory node requests are sampled on the north-bridge of each CPU node. While we implement and evaluate our prediction model on an AMD machine, the necessary performance counters are common in modern NUMA

Table 2: Measured performance counters [12].

CPU performance counters	
CPU Cycles PMCx076 - CPU Clocks not Halted	The number of core clocks that the CPU is not in a halted state
NorthBridge performance counters	
L3 Cache Misses NBPMCx4E1 - L3 Cache Misses	The number of L3 cache misses for accesses from each core
DRAM Requests NBPMCx1EO - CPU to DRAM Requests to Target Node	All DRAM reads and writes generated by cores on the local node to the targeted node in the coherent fabric
MCT Requests NBPMCx1F0 - Memory Controller Requests	The total number of read/write request commands sent to the DRAM controller

systems[1]. Since the AMD North-Bridge supports only up to four simultaneous performance measurements [12], several samples are necessary to obtain all values. One sample measures LLC miss counts, MCT requests, and DRAM requests to two of the eight memory nodes, i.e., a total number of four samples are required to measure the DRAM requests to all eight memory nodes. Details on the sampling rate and the number of full samples are given in Section 5.

4.3 Extracting the Model Parameters

To model the scalability of parallel workloads, a total of five application-specific parameters are required (Table 1): three shared-resource access rates (LLC misses, bus requests, and memory controller requests) and the latencies of both bus and memory controller in the uncongested state. The metrics of the shared resources are computed from the sampled performance counters as follows

$$LLC_M = \frac{L3\ Cache\ Misses\ to\ M\ /\ \#\ Cores\ in\ a\ Node}{Work\ Cycles}$$

$$BUS_M = \frac{DRAM\ Requests\ to\ M\ /\ \#\ Cores\ in\ a\ Node}{Work\ Cycles}$$

$$MCT_M = MCT\ Requests\ to\ M\ /\ Work\ Cycles$$

We cannot directly measure the number of L3 cache misses going to a specific node ($L3\ Cache\ Misses\ to\ M$); the measured L3 cache misses include accesses to all memories. We compute the value by dividing the measured total number of L3 cache misses by the memory access ratio obtained to a specific memory node M from the DRAM request counters to all memory nodes.

A non-trivial problem is the computation of work cycles on AMD processors. Work cycles refer to the number of CPU cycles required to perform the requested work *excluding* stall cycles caused by memory contention. AMD processors do not support performance counters providing the necessary

[1]For example, Intel's Xeon family also provides the performance counters corresponding to each counters on the AMD system: `LONGEST_LAT_CACHE:MISS` for L3 Cache Misses, `DATA_FROM_QPI_TO_NODEx` for DRAM Requests, and `UNC_M_CAS_COUNT:ALL` for MCT Requests [17, 18].

Table 3: Architecture-specific model parameters.

Parameter	Hops	Latency
D_b	0	3.0
	1	5.5
	2	8.0
D_m	-	12.0

data[2]. We thus approximate the number of work cycles W as follows. Let T be the total number of cycles required to perform a certain amount of work including contention. Then, T is W plus the stall cycles incurred by LLC cache misses. The average number of LLC stall cycles is obtained by multiplying the number of accesses from the LLC to each of the different memory nodes, denoted L_m, by the service time of each memory controller, R_m. This value is computed by the queuing model (Section 3), which in turn requires the input request rate given by L_m/W.

$$T = W + \sum_m L_m R_m = W + \sum_m L_m Q(L_m/W) \quad (1)$$

In order to solve equation 1 for W, we apply an iterative method to compute an approximated value of W where the initial input (W_0) is set to T. Since $T \geq W$, W_i converges towards the actual number of work cycles. In our implementation, we use the value of the 5^{th} iteration, W_5, to approximate the number of work cycles.

$$W_{i+1} = T - \sum_m L_m Q(L_m/W_i) \quad (2)$$

The model also requires the architecture-specific parameters $D_b[n][m]$ and D_m (Table 1). $D_b[n][m]$ is the latency incurred when accessing memory node m from the LLC of CPU node n in the uncongested state. Accessing a memory node m from a given CPU node n incurs between 0 (local node) and 2 hops (remote nodes) on our target architecture (i.e., D_b only contains three distinct values). D_m, on the other hand, represents the latency of an uncongested memory controller. On the chosen architecture, there are no performance counters to obtain the required values. We perform a regression analysis on the predicted and the measured speedups of all kernels to compute the values for $D_b[n][m]$ and D_m. Once the values are determined, the values are used for every application. Table 3 lists the values for our AMD target architecture.

5. EVALUATION

5.1 Experimental Setup

The proposed model is evaluated with two programming models, and a variety of applications and configurations. To make a prediction for 8, 16, 24, 32, 40, 48, 56, and 64-core configurations, the first execution of a kernel/loop is briefly sampled on a given configuration (typically one node, i.e., 8 cores). The baseline is obtained by running the parallel application with a different number of assigned

[2]The AMD Opteron architecture provides latency measurement performance counters at the NorthBridge, for example, `CPU Read Command Latency/Requests to Target Nodes` (from NBPMCx1E2 to NPBMCx1E7). However, these counters are not accurate when L3 speculative prefetching is enabled (as is the default) [13].

cores. One sample consists of four sub-samples that extract the different hardware counters as outlined in Section 4.1. We use a sampling frequency of 200 Hz and collect 10 full samples for a prediction, i.e., the entire sampling period takes $4 * 10 * 5ms = 200ms$. We evaluate the model in the following configurations: memory configuration (using 2, 4, and 8 memory nodes), scheduler configuration (static, self, and Li; see next paragraph), and standalone or co-located execution. All experiments were performed three times, and we report the average of the runs.

5.2 Scheduling Policies

The policies for scheduling work units regarding load balancing and dispatch overhead can have a significant effect on the performance scalability of an application.

In the case of OpenCL, scheduling of work-groups is performed by the OpenCL runtime. We test kernels with three scheduling strategies: static scheduling, dynamic (self) scheduling, and guided scheduling. Static scheduling divides and assigns the work-groups of a parallel section equally between the available cores. The dispatch overhead is very small, but the policy may suffer from load imbalances since work-groups are never reassigned. In dynamic scheduling, work-groups are assigned to idle cores at runtime. For the experiments, our dynamic (self) scheduler assigns one work-group at a time. This policy provides good load balancing but suffers from a high dispatch overhead. Guided scheduling assigns several work-groups in one chunk in order to keep the dispatch overhead low yet allow for load balancing. Li's guided scheduling [23], the default in `SnuCL`, assigns $\lceil wg/2N \rceil$ work-groups to an idle core where wg represents the remaining number of work-groups and N the available worker cores. Guided scheduling typically yields the best performance on many-core systems. For OpenMP the programmer can choose between the static, dynamic, and guided scheduling policies by program annotation.

5.3 Target Applications

For the evaluation, we select the kernels of six applications (BT, CG, EP, FT, MG, SP) from SNU-NPB3.3 [30] which provides both OpenCL and OpenMP implementations [29] of the NAS parallel benchmarks [3]. Table 4 lists the characteristics of the target applications.

Our model predicts the scalability of each parallel section individually. The parallel kernels are evaluated with the biggest available input data set. Three classes of kernels are excluded from the evaluation for the following reasons. First, kernels appearing in more than one benchmark application have only been evaluated once. Second, kernels with very short runtimes ($< 0.5sec$) when executed on one CPU node are not considered. Last, we have skipped most of the kernels performing data initialization because their performance either shows linear speedup or saturated performance depending on their memory-boundness. One exception is the `init_ui` kernel with the highest write-intensity of all kernels. We employ the `init_ui` kernel when evaluating the co-located scenario. The selected kernels along with their number of work-groups and work-items are shown in the first three columns of Table 5.

5.4 Evaluation Metrics

The proposed model is evaluated in terms of the accuracy of the predicted scalability. Two metrics, the mean absolute percentage error (MAPE) and the coefficient of determination (R^2), are used to validate our prediction model. MAPE is computed by taking the arithmetic mean of the percentage errors based on the difference between the measured and the predictive value. It is indicated by

$$M = \frac{1}{n} \sum_{k=1}^{n} \left| \frac{a_k - p_k}{a_k} \right|$$

where a_k represents the actual and p_k the predicted value.

The coefficient of determination R^2 indicates the extent to which the speedup can be predicted from the profiled data. R^2 is defined as the square of the correlation between the predicted and the actual speedup.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

Here, n represents the number of data points, y_i the actual and $\hat{y}_i$ the predicted speedup. $\bar{y}$ is the mean value of the observed speedups ($\bar{y} = 1/n \cdot \sum_{i=1}^{n} y_i$). An R^2 value of 1 indicates that the prediction perfectly matches the measured speedup. Values < 0 appear when $\bar{y}$ provides a better fit than the predicted values. The MAPE metric shows how close the prediction is to the actual value, whereas an R^2 value close to one indicates a good match between the predicted scalability trend and the measurement.

Table 4: Target applications.

App	Class (Problem Size)	Description (Workload Characteristics)
BT	D (408×408×408)	Block Tri-diagonal solver (memory intensive; scalable)
CG	D (1500000)	Conjugate Gradient (irregular memory access patterns)
EP	B (2^{30} random number)	Embarrassingly Parallel (CPU intensive)
FT	C (512×512×512)	discrete 3D fast Fourier Transform (all-to-all communication)
MG	D (1024×1024×1024)	Multi-Grid on a sequence of meshes (long and short distance communication)
SP	D (408×408×408)	Scalar Pentra-diagonal solver (memory intensive)

5.5 Scalability Prediction of Standalone OpenCL Kernels

We first analyze the proposed model in the standalone scenario for OpenCL kernels, i.e., except for the program under test, no other (parallel) user programs are running on the server.

Table 5 shows the error (MAPE) and R^2 of each individual kernel for the default system configuration: $200ms$ of sampling using Li's guided scheduling, and memory data allocated to all available eight memory nodes in an interleaving manner. The first three columns list the selected kernels along with their input data settings. The next column labeled `scal.` shows the measured actual speedup of the kernel when running on 8 CPU nodes (64 cores) in relation to the baseline (one CPU node, 8 cores). The last two columns, `err` and R^2, contain the results of predicting the speedup for 64 cores. The average estimation error (geometric mean) and average R^2 (arithmetic mean) across all kernels are shown in the final row of the table.

The results show that, in general, the proposed model accurately predicts scalability with an error of 6.8%. The low

Table 5: Scalability prediction accuracy for the default system configuration.

WGs: # of work-groups, WIs: # of work-items, scal.: scalability

Selected Kernel	WGs	WIs	Prediction		
			scal.	err(%)	R^2
BT.z_solve	406	406	7.21	4.22	0.97
CG.conj_grad_2	640	640	4.52	9.85	0.92
CG.conj_grad_6	640	640	4.04	28.19	0.55
EP.embar	1024	16384	7.71	1.6	1.0
FT.init_ui	2101248	134479872	3.38	19.3	0.53
FT.compute_indexmap	4096	512X512	6.51	2.58	0.95
FT.compute_initial_cond	512	512	7.06	3.59	0.97
FT.cffts1	4096	512X512	4.09	11.47	0.9
FT.cffts2	4096	512X512	5.14	4.34	0.99
FT.cffts3	4096	512X512	5.56	3.48	0.99
SP.exact_rhs2	2842	448X406	4.11	9.64	0.93
SP.exact_rhs3	2842	448X406	5.54	5.19	0.95
SP.exact_rhs4	2842	448X406	5.92	3.77	0.98
SP.compute_rhs1	2856	448X408	5.16	7.55	0.94
SP.compute_rhs2	2856	448X408	4.02	12.45	0.88
SP.compute_rhs3	2842	448X406	4.37	6.2	0.97
SP.compute_rhs4	2842	448X406	6.33	3.64	0.98
SP.compute_rhs5	2842	448X406	7.42	2.18	0.99
SP.z_solve	2842	448X406	4.52	6.93	0.96
SP.tzetar	2842	448X406	4.06	9.52	0.9
MG.kernel_resid	4096	1024X1024	4.13	12.03	0.9
MG.kernel_rprj3	4096	512X512	4.08	25.52	0.6
MG.kernel_interp_1	16384	1024X1024	3.54	11.44	0.86
MG.kernel_psinv	4096	1024X1024	4.22	9.22	0.92
Average			**4.96**	**6.84**	**0.90**

Table 6: Accuracy for different sampling periods.

Kernels having at least 1% differences of MAPE for longer samples.

Kernels	200ms		400ms		Entire run	
	err(%)	R^2	err(%)	R^2	err(%)	R^2
MG.kernel_rprj3	25.52	0.6	19.43	0.78	17.75	0.82

error confirms that the proposed two-level queuing model focusing on congestion in the memory system is adequate to model data-parallel applications. For CPU-intensive kernels such as `BT.z_solve`, `EP.embar`, `FT.compute_indexmap`, and `FT.compute_initial_cond` that rarely access memory and have high scalability, the model predicts speedup within 5% error. Data-intensive (non-trivial) kernels such as `FT.cffts1-3`, `SP.exact_rhs2-4`, `SP.compute_rhs1-5`,`SP.z_solve` and `SP.tzetar` show an accurate scalability prediction with an error of 2-13%. On the other hand, `CG.conj_grad_6` and `MG.kernel_rprj3` exhibit slightly bigger prediction errors. We suspect that the irregular memory access patterns of `CG` and `MG` are the cause for this behavior. Unlike the regular `BT`, `FT` or `SP` benchmarks, `CG` and `MG` exhibit irregular (or varying) memory access patterns (Table 4). Although guided scheduling can mitigate the influence of irregular memory accesses between work-groups, the scheduler may still fail to distribute memory accesses evenly to the available cores. In these cases, the short default sampling period can cause a larger error because the kernel's memory access patterns cannot be fully captured.

The purpose of the experiments in Table 6 is to determine whether a longer sampling phase than the relatively short 200ms at the beginning of a kernel execution allows for better scalability predictions. We compare the default configuration, 200ms, to sampling 400ms and the entire run. Table 6 displays only the kernels which show at least 1% of differences in terms of MAPE with different sampling periods. We observe that the only kernel that significantly profits from a longer profiling phase is `MG.kernel_rprj3` where the accuracy of the prediction increases significantly as the sampling period is prolonged (200ms: 26%, 400ms: 19%, entire-run: 18%). This result confirms that most kernels of the NPB benchmark suite are relatively stable and a sampling period of $200ms$ is sufficient.

Table 7: Scalability prediction accuracy for different work schedulers.

Kernels	Self			Static		
	scal.	err(%)	R^2	scal.	err(%)	R^2
BT.z_solve	7.19	4.21	0.97	7.15	5.14	0.96
CG.conj_grad_2	3.95	24.36	0.7	3.63	31.51	0.11
CG.conj_grad_6	4.14	22.11	0.64	3.87	19.93	0.72
EP.embar	7.63	1.82	0.99	7.97	1.36	1.0
FT.init_ui	0.92	426.77	-1235.12	2.9	26.99	0.54
FT.compute_indexmap	5.84	4.27	0.88	5.2	9.1	0.76
FT.compute_initial_cond	6.76	4.4	0.94	7.17	3.74	0.97
FT.cffts1	6.6	13.89	0.79	2.77	33.81	0.49
FT.cffts2	5.53	4.85	0.97	2.87	34.96	0.44
FT.cffts3	5.83	3.77	0.98	4.15	16.67	0.8
SP.exact_rhs2	4.23	23.34	0.59	4.3	11.32	0.9
SP.exact_rhs3	5.57	6.39	0.93	5.71	7.53	0.92
SP.exact_rhs4	6.05	5.83	0.95	6.28	5.45	0.94
SP.compute_rhs1	2.59	78.38	-9.67	5.91	3.47	0.98
SP.compute_rhs2	5.1	11.75	0.77	4.33	15.78	0.83
SP.compute_rhs3	4.36	6.02	0.97	4.46	6.25	0.97
SP.compute_rhs4	5.84	3.23	0.99	6.29	2.88	0.99
SP.compute_rhs5	6.56	3.01	0.98	7.52	2.04	0.99
SP.z_solve	3.3	44.15	-1.13	4.77	6.97	0.96
SP.tzetar	4.08	8.77	0.91	4.07	9.07	0.9
MG.kernel_resid	3.97	20.77	0.54	1.07	66.14	0.29
MG.kernel_rprj3	3.72	23.92	0.44	1.27	116.85	0.32
MG.kernel_interp_1	3.77	6.19	0.94	2.91	24.07	-0.42
MG.kernel_psinv	3.67	6.38	0.96	3.62	9.55	0.92
Average	**4.54**	**10.47**	**-51.17**	**4.14**	**10.92**	**0.72**

Table 7 shows the results for self- and static scheduling. Self- and static scheduling suffer from scalability issues or load imbalances and thus exhibit higher error rates of 10-12%. For example, `FT.init_ui` and `SP.compute_rhs1` display a very high error (427 and 78%) in Table 7 compared to Li or static scheduling because the central scheduler becomes the bottleneck when delivering work-groups to an increasing number of worker cores. Similarly, `MG.kernel_resid` and `MG.kernel_rprj3` show a high error (66 and 117%) in the static scheduler because of load imbalances caused by the static work distribution as can be seen from the scalability (speedup) values for the different scheduling techniques in Table 5 and Table 7. This is an expected limitation of the proposed model as we assume memory contention to be the only limiting factor and do not account for bottlenecks in the scheduler or imbalanced workloads.

In the next two experiments, the number of available memory nodes are reduced from originally eight to four and two (`Memory-0123` and `Memory-01`, respectively) in order to see how the model performs under increasing memory contention. We observe that the error of the estimation increases as the number of memory nodes decreases. `Memory-01` in particular diverges a lot from the predicted value with an average MAPE of 14.7% and an R^2 of -10.3. With only a small number of available memory nodes, memory-intensive kernels often do not scale at all, i.e., the scalability curve is (almost) flat. Even if the model captures the trend of a kernel's scalability well, modest prediction errors can lead to large percentage errors and negative R^2 values. This is, however, rather a limitation of the evaluation metrics than of the model as the visualization of the predicted versus the actual speedup reveals: the prediction of, for example, `FT.init_ui` on two memory nodes has a percentage error of 42% and

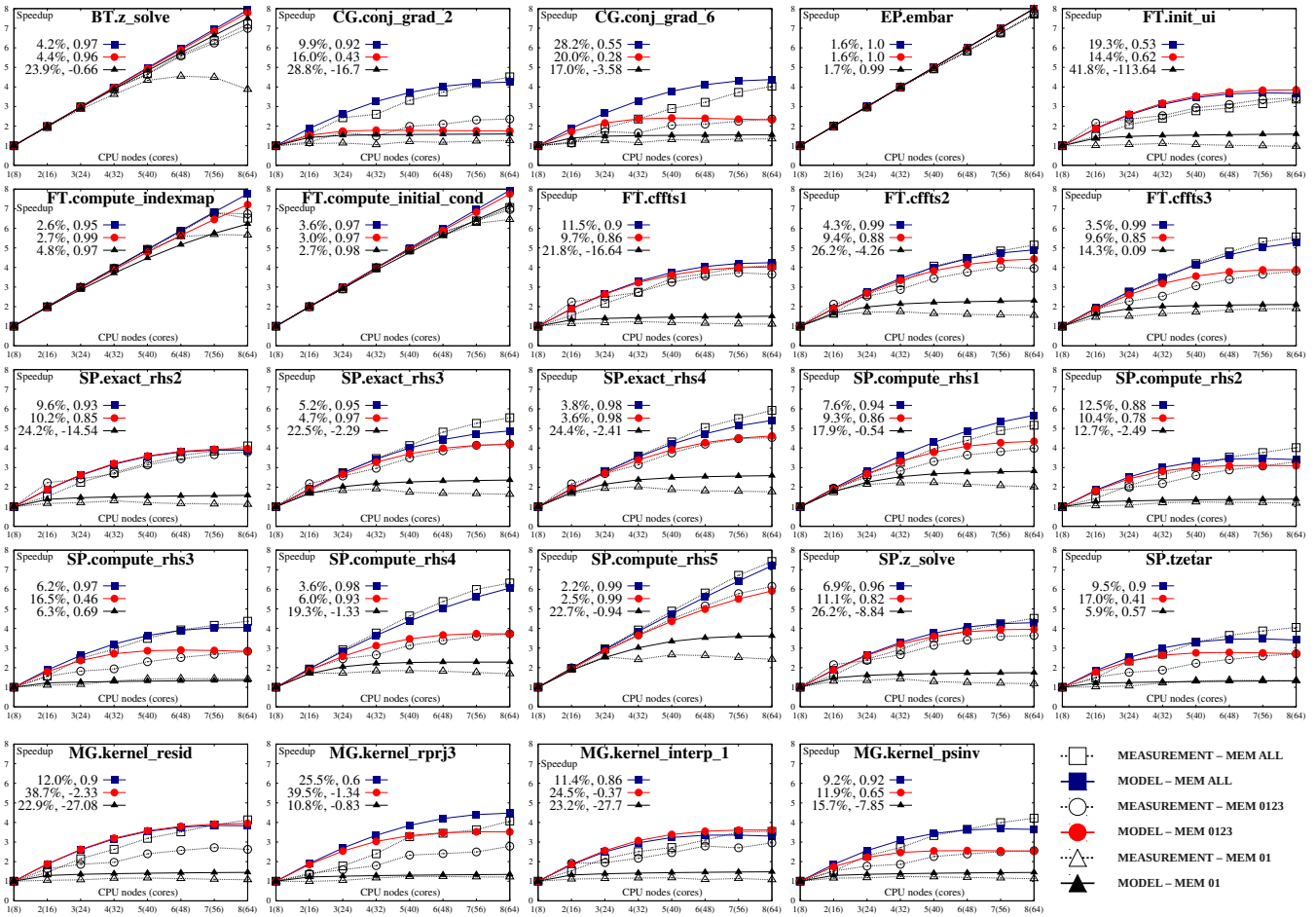


Figure 6: Scalability prediction curves of standalone OpenCL kernels. Each prediction shows MAPE(%) and R^2 values on the graphs.

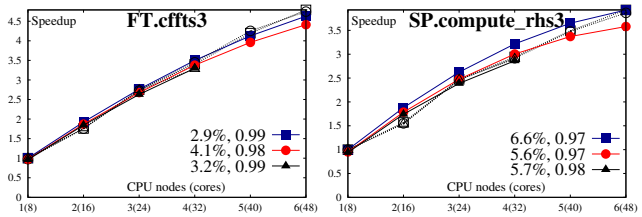
Table 8: Prediction accuracy for varying memory configurations.

Kernels	Memory 0-3			Memory 0-1		
	scal.	err(%)	R^2	scal.	err(%)	R^2
BT.z_solve	6.99	4.43	0.96	3.87	23.88	-0.66
CG.conj_grad_2	2.36	15.96	0.43	1.27	28.77	-16.7
CG.conj_grad_6	2.37	20.0	0.28	1.35	16.97	-3.58
EP.embar	7.73	1.55	1.0	7.68	1.73	0.99
FT.init_ui	3.43	14.44	0.62	0.97	41.84	-113.64
FT.compute_indexmap	6.74	2.71	0.99	5.65	4.77	0.97
FT.compute_initial_cond	6.96	3.04	0.97	6.45	2.71	0.98
FT.cffts1	3.66	9.75	0.86	1.11	21.81	-16.64
FT.cffts2	3.95	9.39	0.88	1.56	26.2	-4.26
FT.cffts3	3.81	9.57	0.85	1.9	14.35	0.09
SP.exact_rhs2	3.8	10.17	0.85	1.12	24.22	-14.54
SP.exact_rhs3	4.22	4.71	0.97	1.64	22.5	-2.29
SP.exact_rhs4	4.53	3.62	0.98	1.76	24.37	-2.41
SP.compute_rhs1	3.99	9.26	0.86	2.01	17.86	-0.54
SP.compute_rhs2	3.3	10.39	0.78	1.18	12.7	-2.49
SP.compute_rhs3	2.84	16.51	0.46	1.42	6.33	0.69
SP.compute_rhs4	3.73	5.97	0.93	1.69	19.31	-1.33
SP.compute_rhs5	6.15	2.55	0.99	2.43	22.68	-0.94
SP.z_solve	3.64	11.1	0.82	1.18	26.17	-8.84
SP.tzetar	2.71	17.01	0.41	1.34	5.92	0.57
MG.kernel_resid	2.63	38.66	-2.33	1.08	22.93	-27.08
MG.kernel_rprj3	2.78	39.54	-1.34	1.22	10.82	-0.83
MG.kernel_interp_1	2.96	24.5	-0.37	1.07	23.22	-27.7
MG.kernel_psinv	2.56	11.93	0.65	1.13	15.67	-7.85
Average	**3.82**	**9.00**	**0.52**	**1.75**	**14.70**	**-10.33**

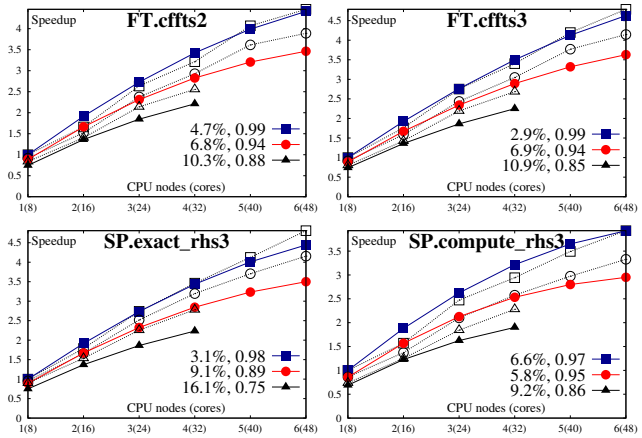
an R^2 of -114 even though the model catches the scalability trend well (Figure 6, top-right graph, triangle data series).

Figure 6 visualizes the measured and the predicted speedup for 24 OpenCL kernels and memory allocation policies under Li's guided scheduling and three memory configurations for eight different core configurations ranging from 8 to 64 cores with an increase of 8 cores. Square, round and triangle data series represent the configurations `Memory-ALL`, `Memory-0123`, and `Memory-01`, respectively. The filled shapes show the predicted, and the empty shapes the measured speedup. The predictions are based on sampling the kernel for 200ms on one CPU node (8 cores).

As online performance prediction relies solely on performance monitoring at runtime and work-group scheduling has a strong influence on the speedup, the model is not always able to predict the scalability with high accuracy. We observe three special cases in Figure 6. First, the short sampling-based prediction may fail to accurately capture the memory access patterns for irregular applications such as `CG` or `MG`. For some of these kernels a longer sampling period leads to a better prediction (e.g., `MG.kernel_rprj3` in Table 6). Second, our model predicts almost linear speedup for CPU-intensive kernels such as `BT.z_solve`, `FT.compute_indexmap` while the actual speedup can be significantly non-linear as the number of CPU node increases. The reason



(a) Co-location with a CPU-bound kernel (`EP.embar`)



(b) Co-location with a memory-bound kernel (`FT.init_ui`)

■ standalone execution, ●/▲ co-located workloads executed on 2/4 nodes (16/32 cores), respectively. Empty shapes represent measured, solid shapes predicted values.

Figure 7: Scalability prediction curves of OpenCL kernels with co-located kernels.

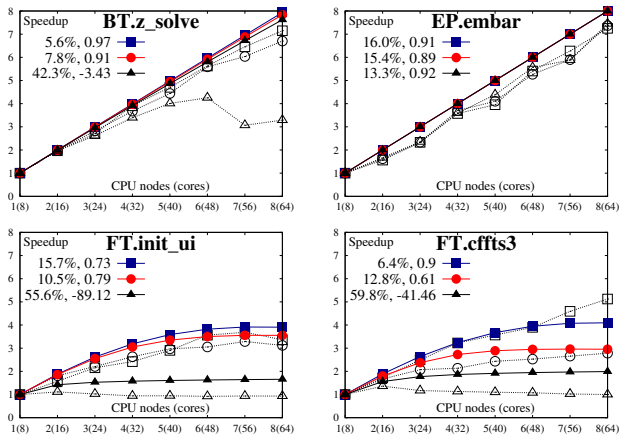
is mainly that these kernels have a small number of work groups (Table 5) compared to the number of system cores. In such a case, guided scheduling can fail to distribute the workloads well. In addition, the work-group scheduler overhead plus other operating system tasks executed with the OpenCL kernel can affect its performance when the kernel is executed on all available nodes. Lastly, some measurement have show irregularities such as `SP.compute_rhs3` and `SP.tzetar` where performance remains stable from 24 to 32 but jumps for 40 or more cores. Here, the additional physical processor can mitigate some congestion and the additional remote LLCs can have a positive effect on performance for certain workloads and node counts. Pressure at some memory links can also be alleviated if more nodes are activated. Our model focuses on major contention points and does not capture all memory-related traffic, but the graphs show that, in general, the proposed model captures scalability trends very well for most kernels.

In the standalone scenario, the average error of 6.8% of the scalability prediction for both trivial and non-trivial kernels shows that the proposed model is a viable candidate to accurately estimate both the expected speedup and the scalability trend for a wide variety of kernels despite the very short at-runtime sampling period.

5.6 Scalability Prediction of OpenCL Kernels with Co-located Workloads

We now analyze how the presence of other concurrently running workloads affects the scalability of a kernel. For the experiments, we run one of the following two kernels as an external workload: `EP.embar` representing a CPU-intensive kernel and `FT.init_ui` as a representative of a memory-intensive kernel. The co-located workloads are executed in an infinite loop to produce a stable test environment. The kernel under test is then executed on a varying number of CPU nodes. We selected four non-trivial kernels that show good prediction accuracy in the standalone scenario (`FT.cffts2/3`, `SP.exact_rhs3`, and `SP.compute_rhs3`).

Figure 7 visualizes the results. The speedup is normalized to the turnaround time of the target kernel executing standalone on one CPU node. To visualize the performance degradation caused by co-located workloads more clearly, the y axis is scaled to the measured speedup on 6 nodes. As expected, co-location with the CPU-intensive `EP.embar` does not cause a performance degradation of the kernel under test (Figure 7 (a)). On the other hand, the memory-intensive `FT.init_ui` causes a visible degradation of performance as shown in Figure 7 (b). The graphs show that the proposed model is able to accurately predict the scalability trend of a single kernel in the presence of co-located parallel applications and can thus be used for dynamic performance tuning or resource management in many-core operating systems.



For the legend refer to Figure 6.

Figure 8: Scalability prediction curves of OpenMP parallel loops.

5.7 Scalability Prediction of OpenMP Parallel Loops

The proposed model can also be applied to OpenMP parallel loops. As for OpenCL, the scalability of a loop is predicted based on a 200ms sampling period at the beginning of a loop. We selected four parallel loops (`BT.z_solve`, `EP.embar`, `FT.init_ui` and `FT.cffts3`) from `SnuNPB` for the evaluation (Figure 8). The OpenMP applications implement the same algorithm as their OpenCL counterparts and are tested with the same input data. Compared to the OpenCL scheduler, the OpenMP runtime applies different load balancing and thread management mechanisms which can affect the scalability of the parallel workloads. The OpenMP parallel loops were executed with a static workload distribution. Since our model assumes perfectly balanced workloads, we expect a reduced prediction accuracy for OpenMP loops. For the OpenMP kernels tested, however, the results show that our model still captures the scalability trend well.

6. RELATED WORK

Performance Prediction on HPC Systems.

A number of performance prediction and modeling techniques have been proposed to aid application design and performance tuning for HPC systems. Barnes *et al.* [4] obtain performance data by running an application on different resource configurations and perform multiple linear regressions. Susukita *et al.* [34] predict an application's performance by macro-level simulation tracing the execution flow of the parallel kernels without performing any real computation. Zhai *et al.* [41] use a single node of the target platform for the acquisition of real computation time and predict the performance by observing the computation behavior of the application. Calotoiu *et al.* [7] automatically find asymptotic performance limitations on large-scale machines based on empirical performance modeling and kernel refinements. Bhattacharyya *et al.* [5] generate performance models based on static information refined by performance data obtained during the program's execution. Palm [36] requires source code annotations of certain parameters to ease the generation of an analytic performance model. COMPASS [22], finally, automatically extracts performance parameters from a source-level analysis and predicts the performance on various target platforms from sequential CPU, over GPUs, to heterogeneous systems. In contrast to the techniques above, the presented model can predict the scalability of an application for modern multi-/many-core single- or multi-node NUMA machines. Another distinction is that all of the techniques require offline program analysis, while our model relies solely on performance counters obtained at runtime during a very short sampling period.

Memory Performance on NUMA Systems.

Addressing the issue of shared resource contention on NUMA systems is also an active area of research. Zhuravlev *et al.,* [42], Blagodurov *et al.* [6] and Majo *et al.* [24] focus on scheduling techniques that consider mitigating contention and improving overall system performance through a proper task-to-core mapping. A holistic framework for NUMA resource management is proposed by Dashti *et al.* [11]. These techniques aim to improve system performance with consideration of shared memory contention, but they do not focus on an application's scalability.

Queuing models have been applied for modeling of memory performance [19, 37], but these works are based on simulation. Tudor *et al.* [39, 38] propose analytic models to understand memory contention in multi-threaded programs based on M/M/1 queuing model. In contrast to their work that only accesses memory nodes local to active CPU nodes, the presented model can predict contention for any number of local and remote memory nodes.

A recent work from Wang *et al.* [40] proposes a model that predicts memory bandwidth usage and optimal core allocation of multi-threaded applications on NUMA systems. Integer programming is used to predict the optimal core allocation based on application profiling. Our work is orthogonal to their work as we predict applications' scalability on varying core resources.

Many-core Resource Management.

Moore *et al.* [25] characterize an application's scalability by employing a utility prediction model based on offline training. The model is used to compute the proper number of threads in order to efficiently execute several multi-threaded applications. Grewe *et al.* [15] and Emani *et al.* [14] predict the proper number of threads for an application in the presence of external workloads. The prediction is based on various performance features fed into machine learning models. These researches show that the performance scalability of applications is an important indicator to optimally schedule parallel programs in a many-core environment, but the techniques require offline training obtained.

A number of techniques have been proposed that do not require offline information. Sasaki *et al.* [27] perform scalability based spatial partitioning of multiple multi-threaded applications. To estimate the scalability factors, applications are executed with varying core allocations while measuring the overall throughput. Similarly, Creech *et al.* [9] estimate the scalability of OpenMP parallel sections by creating and concurrently running a serial process executing the same code as the parallel section in order to determine an application's performance scalability. Both techniques estimate the parallel efficiency of a parallel application but require additional resources or additional reconfiguration time. Sasaki *et al.* [26] also perform power and performance optimization with consideration of the scalability of multi-threaded applications by observing per-core CPU utilization as a scalability indicator. In contrast to our work, these approaches do not consider performance degradation caused by shared resource contention or performance inference between multiple applications.

7. CONCLUSION

This paper presents an efficient at-runtime scalability prediction model for data parallel programs running on many-core NUMA systems. The model employs an M/M/1/N/N queuing model to accurately predict contention both at the individual NUMA memory controllers as well as in the buses connecting the last-level caches with the memory controllers. The prediction is made at-runtime based on performance counters obtained during a short $200ms$ sampling period.

The proposed model has been implemented in an open-source OpenCL and the GNU OpenMP runtimes and evaluated on a 64-core NUMA system. Results with a wide variety of parallel kernels and different configurations show that the model, on average, is able to predict the speedup of an OpenCL kernel with an error of only 6.8% in the most common usage scenario. More importantly, the model captures the scalability trend of individual parallel sections which makes it an ideal candidate for online dynamic resource management in a multi-core scheduler.

The model currently does not work well with data-locality-aware schedulers. This issue is part of future work.

Acknowledgments

This work was supported in part by BK21 Plus for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU) funded by the National Research Foundation (NRF) of Korea (Grant 21A20151113068), the Basic Science Research Program through NRF funded by the Ministry of Science, ICT & Future Planning (Grant NRF-2015K1A3A1A14021288), and by the Promising-Pioneering Researcher Program through Seoul National University in 2015. ICT at Seoul National University provided research facilities for this study.

REFERENCES

[1] AMD. AMD Opteron 6300 Series Processors. http://www.amd.com/en-us/products/server/opteron/6000/6300. [online; accessed March 21, 2016].

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[4] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. De Supinski, and M. Schulz. A regression-based approach to scalability prediction. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 368–377. ACM, 2008.

[5] A. Bhattacharyya and T. Hoefler. Pemogen: automatic adaptive performance modeling during program runtime. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 393–404. ACM, 2014.

[6] S. Blagodurov and A. Fedorova. User-level scheduling on NUMA multicore systems under Linux. In *Proc. of Linux Symposium*, 2011.

[7] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf. Using automated performance modeling to find scalability bugs in complex codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 45. ACM, 2013.

[8] A. Collins, T. Harris, M. Cole, and C. Fensch. Lira: Adaptive contention-aware thread placement for parallel runtime systems. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, page 2. ACM, 2015.

[9] T. Creech, A. Kotha, and R. Barua. Efficient multiprogramming for multicores with scaf. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 334–345. ACM, 2013.

[10] L. Dagum and R. Enon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[11] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: a holistic approach to memory placement on NUMA systems. *ACM SIGPLAN Notices*, 48(4):381–394, 2013.

[12] A. M. Devices. BIOS and kernel developer's guide (BKDG) for AMD family 15h models 00h-0fh processors, 2012.

[13] A. M. Devices. Revision Guide for AMD Family 15h Models 00h-0Fh Processors, 2014.

[14] M. K. Emani, Z. Wang, and M. F. O'Boyle. Smart, adaptive mapping of parallelism in the presence of external workload. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10. IEEE, 2013.

[15] D. Grewe, Z. Wang, and M. F. O'Boyle. A workload-aware mapping approach for data-parallel programs. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 117–126. ACM, 2011.

[16] T. Harris, M. Maas, and V. J. Marathe. Callisto: co-scheduling parallel runtime systems. In *Proceedings of the Ninth European Conference on Computer Systems*, page 24. ACM, 2014.

[17] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, 2015.

[18] Intel. Intel Xeon Processor E5 and E7 v3 Family Uncore Performance Monitoring Reference Manual, 2015.

[19] H. Jonkers. Queueing models of parallel applications: the glamis methodology. In *Computer Performance Evaluation Modelling Techniques and Tools*, pages 123–138. Springer, 1994.

[20] Khronos Group. The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/. [online; accessed March 21, 2016].

[21] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 341–352. ACM, 2012.

[22] S. Lee, J. S. Meredith, and J. S. Vetter. Compass: A framework for automated performance modeling and prediction. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 405–414. ACM, 2015.

[23] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik. Locality and loop scheduling on NUMA multiprocessors. In *null*, pages 140–147. IEEE, 1993.

[24] Z. Majo and T. R. Gross. Memory management in NUMA multicore systems: trapped between cache contention and interconnect overhead. In *ACM SIGPLAN Notices*, volume 46, pages 11–20. ACM, 2011.

[25] R. W. Moore and B. R. Childers. Using utility prediction models to dynamically choose program thread counts. In *ISPASS*, pages 135–144, 2012.

[26] H. Sasaki, S. Imamura, and K. Inoue. Coordinated power-performance optimization in manycores. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 51–62. IEEE Press, 2013.

[27] H. Sasaki, T. Tanimoto, K. Inoue, and H. Nakamura. Scalability-based manycore partitioning. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 107–116. ACM, 2012.

[28] A. L. Scherr. *An analysis of time-shared computer systems*, volume 71.

[29] S. Seo, G. Jo, and J. Lee. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *Workload Characterization (IISWC),*

2011 IEEE International Symposium on, pages 137–148. IEEE, 2011.

[30] S. Seo, J. Kim, G. Jo, J. Lee, J. Nah, and J. Lee. SNU NPB Suite. http://aces.snu.ac.kr/software/snu-npb/, 2011. [online; accessed March 21, 2016].

[31] S. Seo, J. Lee, G. Jo, and J. Lee. Automatic opencl work-group size selection for multicore cpus. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*, pages 387–398. IEEE, 2013.

[32] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu. Performance gaps between openmp and opencl for multi-core cpus. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 116–125. IEEE, 2012.

[33] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu. An application-centric evaluation of opencl on multi-core cpus. *Parallel Computing*, 39(12):834–850, 2013.

[34] R. Susukita, H. Ando, M. Aoyagi, H. Honda, Y. Inadomi, K. Inoue, S. Ishizuki, Y. Kimura, H. Komatsu, M. Kurokawa, et al. Performance prediction of large-scale parallell system and application using macro-level simulation. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 20. IEEE Press, 2008.

[35] J. Sztrik. Basic queueing theory. *University of Debrecen: Faculty of Informatics*, 2011.

[36] N. R. Tallent and A. Hoisie. Palm: easing the burden of analytical performance modeling. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 221–230. ACM, 2014.

[37] T.-F. Tsuei and W. Yamamoto. Queuing simulation model for multiprocessor systems. *Computer*, 36(2):58–64, 2003.

[38] B. M. Tudor and Y. M. Teo. A practical approach for performance analysis of shared-memory programs. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 652–663. IEEE, 2011.

[39] B. M. Tudor, Y. M. Teo, and S. See. Understanding off-chip memory contention of parallel programs in multicore systems. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 602–611. IEEE, 2011.

[40] W. Wang, J. W. Davidson, and M. L. Soffa. Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale numa machines. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 419–431. IEEE, 2016.

[41] J. Zhai, W. Chen, and W. Zheng. Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node. In *ACM Sigplan Notices*, volume 45, pages 305–314. ACM, 2010.

[42] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 129–142. ACM, 2010.

APPENDIX

A. ARTIFACT DESCRIPTION

A.1 Abstract

The software package provided to evaluate the proposed model is composed of several software modules.

- **snucl-model**: the SnuCL framework with an implementation of our online scalability model.
- **gomp-model**: the GNU OpenMP runtime with an implementation of our online scalability model.
- **rte**: a core resource manager (RTE) which manages the core resources for applications (space-sharing). This software is used to execute an application on a specific number of CPU nodes with or without co-located applications.
- **benchmark**: target applications (SNU-NPB) and tools for validating our performance model for both OpenCL and OpenMP applications. The tools include scripts to run the experiments, log analyzers, and plot/table generators.
- **pact16-results**: all experimental results used in the paper are provided here.

All benchmark and visualization tools are written as open-sourced python scripts. The OpenCL and OpenMP runtimes with our online performance model are not open-sourced for now. Instead, we provide pre-compiled binaries of the application runtime libraries.

Our software framework currently runs only on AMD Opteron architectures as it requires access to architecture-specific performance counters. We provide access to our experimental machine, a 64-core AMD server through a guest account. The software package is provided via our private web server. All prerequisites are already installed, on the target architecture. Experiments can be performed immediately without any experimental environment setup.

We also provide detailed installation and benchmark guides in the artifact package. There are two important steps to correctly perform experiments. First, runs requiring access to the hardware performance counters must be executed with super-user privileges. Second, the environment setup as provided in PACT16AE_HOME/env.sh must be maintained. As a consequence, after switching to super-user the environment variables need to be activated (# source env.sh).

A.2 Description

A.2.1 Checklist (artifact meta information)

- **Algorithm**: we present a model for predicting performance scalability. The model is implemented directly into the application runtimes. Performance modeling is performed at runtime and the results are printed when execution of an application ends. It is also possible to model scalability offline by using the collected performance data during application execution. The modeling logic is the same with the online model but scalability predictions can be reproduced efficiently.
- **Program**: we execute SNU-NPB applications. For the experiments we manually adjusted the problem sizes in order to even out the execution time across different benchmarks, but different problem sizes also work.
- **Compilation**: the software modules do not need any special compilers. SnuCL and OpenMP runtimes use gcc/g++, and the benchmark tools require python.
- **Run-time environment**: the software framework runs only on Linux because we use the Linux *perf* interface to query the hardware performance counters.
- **Hardware**: our technique can be applied to other NUMA systems, but the current framework only runs on AMD Opteron NUMA systems.
- **Run-time state**: executing other workloads while collecting performance features of a target applications can negatively affect prediction accuracy.
- **Output**: applications emit log files which include collected performance counter data and predicted speedups values. We analyze the obtained log data to generate graphical outputs.
- **Experiment workflow**: all required experimental steps are explained in a separate file (BENCH) in the software package.
- **Publicly available**? the framework is not publicly available for now.

A.2.2 How delivered

You can download the software package through our private Git server. For the experiments, we also provide a guest account on our evaluated architecture, a 64-core AMD Opteron server. Users requiring access to the software package or the target machine are welcome to contact Younghyun Cho (`younghyun@csap.snu.ac.kr`).

A.2.3 Hardware dependencies

While the performance model is currently only implemented on our target architecture, the technique can be applied to other NUMA systems if such systems support the required performance counters.

A.2.4 Software dependencies

The framework is tested on a Linux operating system (Ubuntu, kernel 3.19). Programs and tools required by our framework (`numactl`, `gnuplot`, and `python` including the `numpy` module) have already been installed in the experimental environment. For more information about software settings and installation please refer to the `INSTALL` file in the software package.

A.2.5 Datasets

While the complete set of experimental results used in the paper is provided in `pact16-results/`, it is also possible to regenerate the same set of experimental data using the provided scripts. The provided data includes raw experimental data, predicted and measured speedups, scalability curve graphs, and table data (Tables 5-8).

In `pact16-results` there are three sub-directories (`10`, `20`, and `full`) in the `plot` directory. The three directories refer to $200ms$, $400ms$ and full execution sampling of a kernel. In addition, in those three directories additional sub-directories labeled `0`, `5`, etc. contain results for different (cache) warm-up periods. The directory `5` contains predictions that ignore the first 5 samples (100ms), and so on. We use 0 warm-up time across all evaluations. We also provide experimental

results for different scheduling algorithms (Static, Self, and LI's guided scheduling) where applicable.

For the co-located experiment scenario, there are two sub directories (`embar_1024`, `init_ui_2101248`). The name refers to the co-located workloads. Lastly, the table data for Tables 5-8 is located in the directory `table/`.

Please refer to the BENCH file in the artifact package for more details.

A.3 Installation

While all required programs and tools have already been installed on the target machine, some software modules (GNU OpenMP runtime, SnuCL runtime, target applications) of our experimental framework need to be installed separately for each guest account. The following software modules must be installed:

1. **SnuCL-Model**: pre-compiled binaries for SnuCL runtime are provided. The SnuCL compiler (OpenCL code translation and compilation tools) must be re-installed.
2. **GNU-OpenMP-Model**: for GNU-OpenMP, performance modeling is implemented through dynamic library interpositioning. The required library code is pre-compiled and provided in binary form. Before execution the library needs to be registered in the system library path.
3. **Target applications (SNU-NPB)**: the source code of all application benchmarks is available.
4. **Core resource manager (RTE)**: pre-compiled binaries are provided.

After installation of all required software modules the experiments can be performed. Detailed explanations about the software packaging and the installation steps can be found in the `INSTALL` file in the artifact package.

A.4 Experimental Workflow

Three experiments can be executed with the provided software package.

1. **SnuCL-Model**: scalability prediction of standalone OpenCL kernels.
2. **SnuCL-Model**: scalability prediction of OpenCL kernels with co-located workloads.
3. **OpenMP-Model**: scalability prediction of standalone OpenMP parallel-sections.

Required experimental steps (environmental setup, benchmark scripts, and visualization tools) are explained in the `BENCH` file of the artifact package.