Projecting Performance for PIUMA using Down-Scaled Simulation

Stijn Eyerman, Wim Heirman, Yigit Demir, Kristof Du Bois, Ibrahim Hur — Intel Corporation

***Abstract*—Programmable Integrated Unified Memory Architecture (PIUMA) is Intel's novel graph analysis optimized processor architecture, targeted at efficiently executing graph algorithms on very large graphs. Simulation is used to project its performance for various algorithms before the system is built. However, simulators are limited in the number of cores and threads they can simulate, because of their low simulation speed, high resource usage and poor scalability. Therefore, it is practically impossible to simulate PIUMA at its full system scale.**

In this paper, we present downscaled simulation, a technique to project performance of a large scale system using a small scale simulation. We apply the technique to PIUMA, showing how to configure the downscaled system in order to accurately reflect the characteristics of the full system. We evaluate downscaled simulation on a set of graph applications, showing that it accurately tracks simulation results of small scale simulations, as well as the projections to large systems made by an analytical model.

I. Introduction

Simulating a large scale system (e.g., a multi-node supercomputer) is a challenging task. Detailed simulation of computer systems is very resource intensive: it requires executing thousands of simulator instructions on the host to simulate the performance of a single instruction on the simulated system. As a result, detailed simulators execute more than 4 orders of magnitude slower than the actual hardware. Furthermore, parallel timing simulation requires tight synchronization to accurately model the interactions between cores, network and memory at small time granularity, which limits the potential speed benefit of parallel simulation.

On the other hand, detailed simulation can be the only option to reach the required level of accuracy. Analytical models [8] or spreadsheet models only model high-level characteristics, ignoring lower level details that can have a significant impact on performance, such as network contention or timing variability. Decoupled simulation, e.g., simulating a single node in detail and separately simulating the interconnection network [13], ignores the interactions between the components: network contention has an impact on node performance, which on its turn impacts the network injection rate, etc.

In this paper, we propose an efficient simulation methodology for large-scale systems, such as PIUMA (Programmable Integrated Unified Memory Architecture, formerly known as PUMA) [9]. PIUMA is a graph-oriented instruction set processor, consisting of many multi-threaded cores to hide memory latency, a hardware distributed global address space (DGAS) for efficient random remote accesses, and offload engines for common memory-intensive operations, such as gathers and barriers. A multinode PIUMA systems consists of thousands of cores and up to millions of threads, making it impossible to fully simulate to assess its performance.

Our simulation technique scales down the system at each level (e.g., core, node and multi-node system), ensuring that we simulate each component of the system in detail (cores, memory controllers, on-chip network, inter-node network, etc.), but on a smaller scale to limit the time and resources used by the simulation. This approach ensures that we model the impact of all components and their interactions, which is not feasible using high-level models or decoupled simulation.

In particular, this paper makes the following contributions:

- We present our novel methodology to simulate a large system using a downscaled system with extrapolation.
- We apply this methodology to project performance of a multi-node graph processor, and show how the downscaled system is configured to represent the full system as closely as possible.
- We compare the projected performance to that of a small scale simulation and an analytical model, showing that downscale simulation provides very similar results, and likely even more accurate results as it models more aspects of the system than an analytical model.

II. Related Work

Fast, accurate and scalable processor simulation has been a long-lasting subject of research. Prior work has mainly focused on reducing the workload to be simulated (application sampling), accelerating the simulation model by making it less complex, and using parallel simulation to increase simulation speed. Our proposal adds a new dimension: sampling the system to reduce the scale of the simulation, and using extrapolation to project performance for a large scale system. We discuss prior work on the former dimensions in the next sections.

A. Application sampling

Realistic applications consist of billions of instructions to be executed, which takes days or weeks to simulate even for a small system on a fast simulator. Sampling the application to reduce the instruction count has been an active research area. Best known are the SMARTS technique [24], which performs periodic statistical sampling, and SimPoints [21], which detects phase behavior in applications, and uses one representative sample per phase to cover the behavior of the full application. Carlson et al. [3] propose a sampling methodology for multi-threaded applications, which is more complicated due to memory operation interleaving and synchronization events. Gonzalez et al. [11] describe a sampling technique for distributed MPI applications.

B. *Fast simulation models*

A way to speed up simulation is to abstract away certain details of the processor microarchitecture that are assumed to have no impact on performance in normal circumstances. Instruction window centric simulation [5] abstracts away many components, assuming they are designed in balance with the instruction window size and pipeline width, resulting in an order of magnitude speedup while limiting simulation error.

Other proposals use an analytical processor model instead of detailed simulation [10], or simulate small parts of the application and use these results to estimate the timing of the rest of the application [12]. Large scale network simulators often use very simple processor models, such as a one-instruction-per-cycle (one-IPC) model [18], or use a synthetic network traffic simulator [14]. The latter two models are very fast, but the traffic they model does not resemble that of a real application, resulting in questionable validity and usefulness.

Traces of real applications, containing the network operations (e.g., MPI calls) with their timing [13], [20], model traffic more realistically, but are tied to a specific system setup (number of nodes, number and type of cores on a node) and these traces quickly become too large to store practically. Furthermore, these models assume clearly separated computation and communication phases, which is usually the case in bulk synchronous MPI applications. For more fine-grained communication, e.g., in a distributed global address space (DGAS) where each memory access can potentially cause communication, compute and communication are highly intertwined, making it hard to model accurately with completely separated compute and network models. Our technique simulates both the compute (cores, memory) as the communication (network) concurrently in detail, so it models the combined effect more accurately.

C. *Parallel simulation*

A third way to speed up simulation is to run parallel simulation threads [4]. However, to achieve accurate simulation in shared components (e.g., shared cache, network on chip, memory controller), the simulated time of all threads should be synchronized, which significantly reduces the parallel efficiency and simulation speed. Distributed simulation across multiple simulation hosts [18], [19] offers even more parallelism, at the cost of a higher implementation complexity, lower accuracy due to loose synchronization and/or low simulation speed due to superlinear synchronization delays.

While all techniques described here have the same goal as our proposal, namely speeding up simulation, none of them targets the main issue we tackle in this paper: the scale of the simulated system. Our down-scale simulation method is orthogonal to these prior techniques: it can be combined with any of them to further reduce simulation overhead.

III. PIUMA ARCHITECTURE

The main feature of PIUMA is that the full system has distributed shared memory, i.e., all memory locations can be accessed from any point in the system, even across nodes. Crucial for full-system shared memory supporting sparse accesses is a high-bandwidth low-latency interconnection network, alleviating the need for partitioning the data. Thereto, our graph processor configuration uses a hierarchical network configuration, with all-to-all network connections at each level (HyperX network topology [1]). This topology significantly limits the number of hops to reach a distant memory location, and it has a high exclusive bandwidth between each pair of cores.

The system has 5 hierarchic levels: pipeline, core, tile, node and system. The pipeline is the basic execution unit, executing instructions from multiple threads. Because graph analysis applications are mainly memory bound, the pipeline is a multi-threaded single-issue in-order pipeline. To completely fill the pipeline, it supports 16 concurrent threads, providing a hardware thread context for each thread. Each pipeline has a small private cache. Due to the absence of locality in most graph applications, PIUMA has no higher level or shared caches.

Four pipelines form a core. Each core has one memory controller, that addresses its part of the shared memory system. Eight cores make up a tile. Cores on a tile are connected using an all-to-all network-on-chip.

A node consists of 16 tiles. Each tile is connected to each other tile on the same node. Each tile also has inter-node links, for creating a multinode system. The HyperX network topology enables to scale out to any number of nodes, by adding levels. An example 32 node system has 16 threads per pipeline, times 4 pipelines per core, times 8 cores per tile, times 16 tiles per node, times 32 nodes in the system, or a total of 256 K threads.

IV. DOWN-SCALING A PIUMA SYSTEM

The motivation of our method is that it is impossible to simulate all components of a full PIUMA system, because that would require too many resources and would take too much time. Therefore, we *sample* the system by simulating only part of it. However, simulating, for example, a single tile of a few cores means that we do not model the impact of the inter-tile and inter-node network, which is crucial for projecting performance to the full system. Therefore, our downscale method samples sparsely: a few cores on a few tiles on a few nodes. That enables modeling all components of the system: cores, tiles, nodes and the corresponding network and memory controllers.

Table I shows the hierarchical configuration of a 32 node system and the choices we made for downscaling the configuration. The idea behind the downscaled numbers is that we reduce the count at each level by a factor of 4 (1 pipeline per core instead of 4, 2 cores per tile instead of 8, etc.). This ensures that the relative counts at each level remain the same. Note that we do not scale down the number of simultaneous

TABLE I
FULL SYSTEM AND DOWN-SCALED HIERARCHY

Level	Full system	Down-scaled
Pipeline	16 threads	16 threads
Core	4 pipelines	1 pipeline
Tile	8 cores	2 cores
Node	16 tiles	4 tiles
System	32 nodes	8 nodes

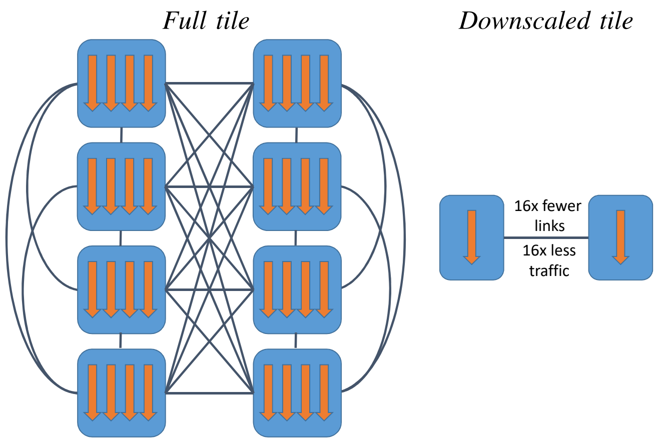

Fig. 1. Tile and on-tile network in the full system (left) and in the downscaled system (right). Arrows represent pipelines, boxes are cores.

threads in a pipeline, because the effect of multithreading a pipeline has no linear scaling. Scaling down the thread count to 4 threads instead of 16 could reduce the generated memory traffic (data accessed per time unit) by a factor of 4 if performance is memory bound, i.e., the pipeline is mainly stalled on outstanding memory operations: there are 4 instead of 16 outstanding misses. However, if performance is compute bound, i.e., bound by the compute capacity of the pipeline, then the generated memory traffic will be similar with 4 threads as with 16 threads. To faithfully model the impact of pipeline multithreading on performance for all types of behavior, we therefore keep the thread count per pipeline at 16. Because each pipeline has its own cache and there are no other caches, we do not need to scale caches.

Next, we need to scale memory and network bandwidth, in order to faithfully model contention in the memory controller and network links. There is one memory controller per core. The core is down-scaled with a factor of four (1 pipeline instead of 4), meaning that there are four times fewer threads than on the real system, generating four times less memory traffic. Therefore, we need to scale down the bandwidth per memory controller with a factor of 4. Because the number of memory controllers in the system is equal to the number of cores, we do not need to scale the memory bandwidth further as the system size increases: for every downscaling of the core count, the number of memory controllers also scales proportionally.

For downscaling the network bandwidth, we use the method of *equal bisection bandwidth usage* at each level of the network hierarchy. Bisection bandwidth is the sum of the bandwidths of all links that cross the boundary between two halves of a system. For example, consider a tile that has 8 cores, interconnected with all-to-all connections, see Figure 1. Half a tile has 4 cores, and each of these cores has 4 links going to the 4 cores on the other half that cross the bisection. So the total bisection bandwidth is the bandwidth of 16 links. Our downscaled tile has only 2 cores, with one link between them. That means that in order to have the same bisection bandwidth, this link should have 16× higher bandwidth than the inter-core links in the full system. However, the number of threads that generate network traffic is also smaller. Assuming that bisection network traffic is proportional to thread count, the total amount of traffic generated on one halve of the downscaled system is 16× lower than on the full system: 4× fewer pipelines per core and 4× fewer cores on a tile. As a result, with 16× fewer links and 16× less traffic across the bisection, we can use the link bandwidth of the actual system to have the same bandwidth usage as in the real system.

Table II shows the bandwidth downscale factors for each level of the system. Although the total system is downscaled with a factor of 256, bandwidth numbers only need to scale down with a maximum factor of 16, because of the reduction in link count when the system size is reduced.

Regarding latency, we keep the uncontended link latencies the same as in the full system, because the way the system is downscaled ensures that the number of hops taken by each request is the same as in the full system. The bandwidth scaling ensures that bandwidth usage is modeled correctly, which means that if bandwidth usage is high, extra queuing latency will be added by the simulator's memory and network model, as would occur in the full system.

A. Applications and Extrapolation

Next to downscaling the system, we also need to downscale the application running on the system, i.e., reducing its thread count but retaining the same behavior as if they execute on the full system. Fortunately, most large scale applications running on these large scale manythreaded systems are homogeneous, i.e., each thread executes the same code on different data. If the available thread count increases, the work (the data to process) is simply divided among more threads, while each individual thread keeps executing the same code but on a smaller part of the data. The applications evaluated in this paper are all homogeneous applications that scale well from a few to many threads, with homogeneous behavior in all threads.

After simulating the downscaled configuration, we need to extrapolate the results to the full system. The most straightforward way to extrapolate is to assume that the application scales perfectly with increasing thread count, and thus to divide the simulated execution time of the downscaled system by the total downscale factor, e.g., 256 for the 32-node system (perfect strong scaling).

However, even with a well scaling homogeneous application, it is potentially unsafe to assume perfect scaling with such a high scaling factor. For example, increasing the thread count with 256× using the same input data size might lead

TABLE II
NETWORK LINK BANDWIDTH DOWNSCALE FACTOR CALCULATION FOR THE DOWNSCALED SYSTEM (TH = THREADS, C = CORES, T = TILES, N = NODES).

	# threads per halve			# bisection links			total
Level	full system	downscaled	factor	full system	downscaled	factor	factor
Tile	64 th/c$\times$ 4 c	16 th/c$\times$ 1 c	16	4×4	1	16	$16/16 = 1$
Node	64 th/c $\times$ 8 c/t $\times$ 8 t	16 th/c $\times$ 2 c/t $\times$ 2 t	64	8×8	2×2	16	$16/64 = 1/4$
System	64 th/c $\times$ 128 c/n $\times$ 16 n	16 th/c $\times$ 8 c/n $\times$ 4 n	256	16×16	4×4	16	$16/256 = 1/16$

to load imbalance, where some threads have less work to do than others because the data cannot be evenly divided. This is not modeled by our downscaled system, because of the much smaller thread count. Therefore, it is usually safer to extrapolate using weak scaling: for a downscaled simulation that has f times fewer threads than the full system, the full system will have the *same execution time* as the simulated downscaled system on *f times larger input data* (assuming that the input can be scaled linearly). For example, if we use a graph with 1 million vertices to simulate the downscaled system, the full system will have the same execution time processing a graph with 256 million vertices. Scaling the input set with the downscale factor ensures that the amount of work per thread remains constant, avoiding the need to profile or model strong scaling limitations.

V. EXPERIMENTAL SETUP

To validate the downscale methodology for large scale system simulation, we use an adapted version of the Sniper multicore simulator [4] to simulate an PIUMA-like architecture described above. We also port four graph kernels and applications to the PIUMA architecture: Random Walks, Graph Search, Breadth First Search and Application Classification.

A. Evaluated Applications

Random Walks (RW) [16] samples a graph by selecting a set of source vertices and performing walks by selecting a random neighbor of the current vertex as the next step. Graph Search (GS) [23] looks for vertices with a large attribute value. From a source vertex, it selects the neighbor with the largest attribute value to walk to, and proceeds from there with the same algorithm (steepest ascent).

Breadth First Search (BFS) [2] starts from a single source vertex and traverses the graph breadth-first, i.e., all of a vertex's children before the children of these children. Application Classification (AC) [17] detects patterns of malware network traffic from a large traffic graph by performing subgraph matching: the malware graph pattern is matched to the full graph to detect occurrences of this pattern.

We selected these four applications because they have different behavior (latency, bandwidth or compute bound) and they represent common graph operations (fetching neighbors, attributes, etc.), which is the designated domain of the PIUMA processor design. A common characteristic of all four applications is that they have a random uniform access pattern: each memory operation accesses each memory controller in a uniform way. In other words, there is no locality, which means most accesses go to another memory controller than the one closest located to the core it is executing.

To evaluate new applications in our infrastructure, we ported them to the SPMD programming model for PIUMA, and optimized them to efficiently support large thread counts, system-wide shared memory and user-level scratchpads. We run all applications on synthetic RMAT graphs [6] with scale 24 (16 million vertices). For Application Classification (AC), we use an RMAT scale 14 graph as the data graph, and a pattern graph of 10 vertices.

B. Analytical model

Because we cannot practically simulate full systems, we cannot compare our downscale method against full system simulation results, nor can we compare against machine measurements, because the system we model does not yet exist. However, because the applications are relatively simple and the pipelines are single-issue in-order processors, it is possible to predict the performance pretty accurately using an analytical model. The model takes into account the performance of the innermost loop, by counting the number of instructions and memory operations, and also taking into account memory and network bandwidth and latency.

The general idea of the model is that performance is either limited by the pipelines, memory bandwidth or network bandwidth. It calculates the minimum execution time of all iterations of the innermost loop in the pipeline, by adding the latency of all instructions of the loop, including the average latency of the memory operations. This time is divided by the total thread count, assuming perfect workload balance across threads.

Next, the amount of memory accessed is calculated by multiplying the memory accesses and sizes in the innermost loop with the iteration count. This number is divided by the aggregate memory bandwidth, i.e., adding the bandwidth of all memory controllers, to get the minimum time required to perform the memory operations.

We also estimate the amount of network traffic by assuming a uniform random distribution of remote memory accesses, which is a good approximation for most graph applications. Because the lower levels of the HyperX network topology are more densely connected than the higher levels, and network traffic is uniform, the network bottleneck is usually situated at the highest level (e.g., the inter-node network level for multiple nodes). Therefore, we calculate the bisection bandwidth on the highest level, i.e., the total bandwidth between two equal halves of the system. By dividing the estimated bisection traffic

TABLE III
CONFIGURATIONS SIMULATED AND PROJECTED (“*w*N *x*T *y*C *z*P” MEANS *w* NODES, *x* TILES PER NODE, *y* CORES PER TILE AND *z* PIPELINES PER CORE).

Dense	Downscale		
	simulated	extrapolated to	factor
1n 1t 1c 4p	1n 1t 1c 1p	1n 1t 1c 4p	4×
1n 1t 2c 4p	1n 1t 2c 1p	1n 1t 2c 4p	4×
1n 1t 4c 4p	↪	1n 1t 4c 4p	8×
1n 1t 8c 4p	↪	1n 1t 8c 4p	16×
1n 2t 8c 4p	1n 2t 2c 1p	1n 2t 8c 4p	16×
	1n 4t 2c 1p	1n 4t 8c 4p	16×
	↪	1n 8t 8c 4p	32×
	↪	1n 16t 8c 4p	64×
	2n 4t 2c 1p	2n 16t 8c 4p	64×
	4n 4t 2c 1p	4n 16t 8c 4p	64×
	8n 4t 2c 1p	8n 16t 8c 4p	64×
	↪	16n 16t 8c 4p	128×
	↪	32n 16t 8c 4p	256×

by this bandwidth, we obtain the minimum time needed to transfer the data across the network.

Finally, we assume that pipeline latency, memory access time and network time are overlapping, so the final execution time estimation equals the maximum of these three timings. The analytical model has several assumptions that might or might not be valid for the evaluated workload, such as assuming perfect balance, uniform access patterns and limiting the application to the innermost loop only. It also only takes into account averages, ignoring the impact of traffic bursts. Furthermore, it involves a detailed study of the assembly instructions of the inner loop of the application, to determine the instruction count, types and memory behavior. The downscale simulation method requires a specific configuration of the simulator, but once that it is determined, it can simulate most applications unchanged, making its applicability and ease of use better than that of the analytical model. It also more accurately simulates the effect of individual events, such as temporary imbalance and bursts.

VI. VALIDATION

In this section, we use our experimental setup to evaluate and validate our downscale method. We compare against two other methods:

- Full simulation: we simulate all threads, pipelines, cores, tiles and nodes of a part of the system. Given the high overhead of full simulation, we could simulate up to 2 nodes for RW and GS. For AC and BFS, we simulate up to 2 and 4 tiles respectively.
- Analytical model: described in the previous section.

The downscale simulation and the analytical model are validated on a small scale system by the full simulation. Although there is no golden reference for the large scale system, if both the analytical model and the downscale simulation method agree, the confidence in both techniques is high.

In order to have multiple points of comparison and to show that the downscale model is able to accurately track scaling trends, we perform multiple simulations or projections per application, starting from one core up to 32 nodes. Table III

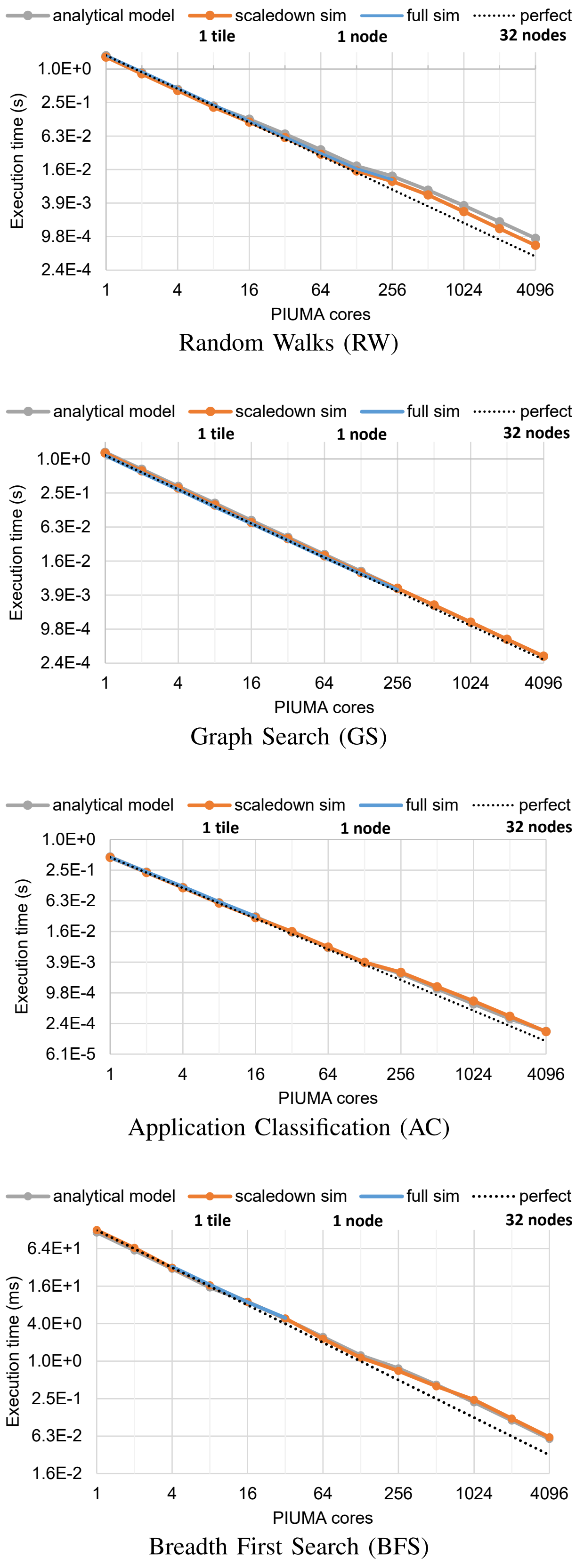

Fig. 2. Comparison between full simulation, downscaled simulation, analytical model and perfect scaling model.

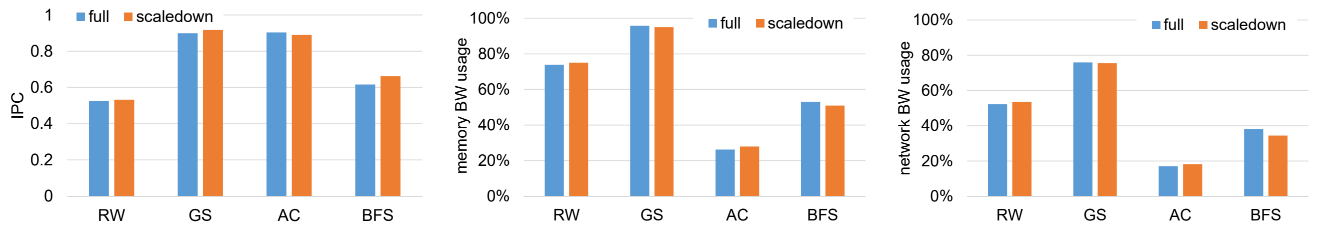

Fig. 3. Comparison between full 2-tile simulation and scaledown simulation, in terms of IPC, memory bandwidth usage and network bandwidth usage.

shows the configurations evaluated for the dense sampling and for the downscale simulations, including how they are extrapolated. The scaling factor in the table is the factor by which the execution time is divided or input is scaled up for the extrapolation.

Figure 2 shows the results of the validation study, comparing the full simulation, downscale simulation method and the analytical model from 1 to 4,096 cores (or 32 nodes). Note that BFS has no results for the dense sampling at 1 and 2 cores because the core local storage is too small to execute the algorithm on 1 and 2 cores. For comparison, we also add a perfect scaling curve (execution time equals the execution time of one core divided by core count), to show that both the downscale model and the analytical model track the scaling effects in a more detailed way than a simple perfect scaling model.

For the configuration for which we have full simulation results, the full simulation, scaledown simulation and analytical model results match very well. Within a tile (8 cores), scaling is perfect as indicated by the dotted line. The network connecting cores on a tile has the lowest latency and highest bandwidth, explaining the good scaling. For RW and BFS, scaling is slightly worse than perfect when going to two tiles (16 cores). RW and BFS are latency sensitive, the extra latency introduced by going off-tile reduces its performance. This behavior is captured by both the downscale simulation and analytical model, which shows that the downscale simulation is able to capture the network effects, even though the simulated thread count is 16 times lower than that of the dense simulation.

For most applications, we see a larger deviation from perfect scaling once we go beyond 1 node (128 cores). The inter-node network links have the longest latency, which is the main limiting factor for RW and BFS. Furthermore, the uniform access pattern means that in a multinode system, most accesses will leave the node, saturating the inter-node links. This is the the case for Application Classification. Graph Search is memory bandwidth bound, and because aggregate memory bandwidth scales linearly with core count, it shows perfect scaling.

At large core counts (multiple nodes), we notice some deviation between the analytical model and the scaledown method, although they follow the same trend compared to perfect scaling. For RW, the analytical model predicts a slightly larger execution time. The analytical model assumes the worst latency for going off-node: first going to another core on the same tile, then going to another tile, before jumping to the other node, and the same path on the destination node (to other tile and other core). Some operations do not need to take this path, because their local inter-node link already connects to the destination node. The scaledown method models this routing more accurately, resulting in a lower average latency and better performance.

To further validate the accuracy of the scaledown method, Figure 3 compares the average IPC of a pipeline (max 1), average memory bandwidth usage (fraction of peak bandwidth) and average network bandwidth usage for a 2-tile simulation using a full simulation (2 tiles of 8 cores of 4 pipelines) and the downscale simulation (2 tiles of 2 cores of 1 pipeline). The figure shows that for all of these metrics, downscale simulation method matches full simulation, and it tracks the diversity of these metrics across the 4 applications.

The results show that scaledown simulation method is able to accurately model a large scale system while limiting the simulation size and overhead. It models the effect of intra-node and inter-node network latency and bandwidth on performance, and finds scaling bottlenecks. While we show that an analytical model is also able to track these effects, downscale simulation is easier to deploy, because it boils down to a simulation with a slightly changed configuration, while analytical modeling requires more insight and could be impossible for more complex applications. Full simulation is the most accurate projection technique, but also requires the most time and resources. As an example, the full 2-node simulation of RW took almost 3 days using 33 high-end Xeon machines, while the 2-node scaledown simulation took 4.5 hours on one machine.

VII. Conclusions

We propose a method to perform performance projections for large scale systems, while keeping the simulation overhead limited. The system is downscaled at each level, such that we still model all components (cores, network, memory) and their effects on performance. By extrapolating the results of a downscaled simulation, we can project the performance of that system, either through weak or strong scaling. We show that the downscale method corresponds well with full simulation for small scale systems, and that it follows the same trends as an analytical model, while being more straightforward to use and potentially more accurate, because it simulates the (downscaled) application in detail, while an analytical model is more high-level.

Acknowledgement

This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

[1] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber, "HyperX: topology, routing, and packaging of efficient large-scale networks," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 41.

[2] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov 2012, pp. 1–10.

[3] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sampled simulation of multi-threaded applications," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013, pp. 2–12.

[4] ——, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011.

[5] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 3, pp. 28:1–28:25, Aug. 2014.

[6] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*, 2004, pp. 442–446.

[7] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. D. Bosschere, "Statistical simulation: Adding efficiency to the computer designer's toolbox," *IEEE Micro*, vol. 23, no. 5, pp. 26–38, 2003.

[8] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Trans. Comput. Syst.*, vol. 27, no. 2, pp. 3:1–3:37, May 2009.

[9] S. Eyerman, W. Heirman, I. Hur, and J. B. Fryman, "Programmable unified memory architecture (PUMA)," FOSDEM 2020, Feb 2020.

[10] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *International Symposium on High-Performance Computer Architecture (HPCA-16)*. IEEE, 2010, pp. 1–12.

[11] J. Gonzalez, J. Gimenez, M. Casas, M. Moreto, A. Ramirez, J. Labarta, and M. Valero, "Simulating whole supercomputer applications," *IEEE Micro*, vol. 31, no. 3, pp. 32–45, May 2011.

[12] T. Grass, C. Allande, A. Armejach, A. Rico, E. Ayguadé, J. Labarta, M. Valero, M. Casas, and M. Moreto, "MUSA: A multi-level simulation approach for next-generation HPC machines," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016, pp. 45:1–45:12.

[13] C. L. Janssen, H. Adalsteinsson, S. Cranford, J. P. Kenny, A. Pinar, D. A. Evensky, and J. Mayo, "A simulator for large-scale parallel computer architectures," *Technology Integration Advancements in Distributed Systems and Computing*, vol. 179, 2012.

[14] N. Jiang, J. Balfour, D. U. Becker, B. Towles, W. J. Dally, G. Michelogiannakis, and J. Kim, "A detailed and flexible cycle-accurate network-on-chip simulator," in *IEEE International Symposium on Performance Analysis of Software and Systems (ISPASS)*, 2013, pp. 86–96.

[15] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic, "FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 29–42.

[16] J. Leskovec and C. Faloutsos, "Sampling from large graphs," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006, pp. 631–636.

[17] G. Levchuk, J. Colonna-Romano, and M. Eslami, "Application of graph-based semi-supervised learning for development of cyber COP and network intrusion detection," in *Disruptive Technologies in Sensors and Sensor Systems*, R. D. Hall, M. Blowers, and J. Williams, Eds., vol. 10206, International Society for Optics and Photonics. SPIE, 2017, pp. 67 – 82. [Online]. Available: https://doi.org/10.1117/12.2263543

[18] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *International Symposium on High-Performance Computer Architecture (HPCA-16)*. IEEE, 2010, pp. 1–12.

[19] A. Mohammad, U. Darbaz, G. Dozsa, S. Diestelhorst, D. Kim, and N. S. Kim, "dist-gem5: Distributed simulation of computer clusters," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2017, pp. 153–162.

[20] M. Mubarak, C. D. Carothers, R. B. Ross, and P. Carns, "Enabling parallel simulation of large-scale HPC network systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 87–100, Jan 2017.

[21] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," *IEEE Micro*, vol. 23, no. 6, pp. 84–93, Nov 2003.

[22] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanović, "RAMP gold: an FPGA-based architecture simulator for multiprocessors," in *Proceedings of the 47th Design Automation Conference*, 2010, pp. 463–468.

[23] V. Viswanathan, A. K. Sen, and S. Chakraborty, "Stochastic greedy algorithms," *International Journal on Advances in Software*, vol. 4, no. 1, 2011.

[24] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling," in *International Symposium on Computer Architecture (ISCA)*, June 2003, pp. 84–95.