COSMIC Heterogeneous Multiprocessor Benchmark Suite user manual

Version 4.0

OPTICS Lab

Big Data System Lab

Department of Electronic and Computer Engineering

Hong Kong University of Science and Technology

https://eexu.home.ece.ust.hk

March 2020

CONTENTS

I. Introduction

Recent advances in the computing industry towards multiprocessor technologies shifted the dominant method of performance increase from frequency scaling to parallelism. Due to the huge design space of multiprocessor systems, evaluating candidate architectures in early design stages, when the number of variables is at its maximum, is challenging. Simulation plays an important role not only in validating the circuit correctness but also in estimating candidate architectures performance. Commonly used simulation methods requires the use of representative benchmarks that should be compiled for specific instruction set, and executed on top of an abstraction layer that provides a set of services to access the hardware.

Estimating the average performance of a candidate architecture requires evaluation of the system with a representative number of input datasets and benchmarks that cover most of the architecture purpose domain. However, enumerating all possible inputs for a given application is very inconvenient and in most of the cases impossible. For instance, for a hardware designed to perform *H.264* video coding/decoding, enumerating input datasets means obtaining numerous image frames with representative resolution, to depict temporal and spatial characteristics of realistic videos. Clearly, such approach does not scale well, and is aggravated by the fact that simulation is naturally slow and performing simulation runs for individual input dataset becomes prohibitively time and resource consuming. Additionally, heterogeneous multiprocessor systems has a lack of operating systems support or it is usually not decided in early design stages, diminishing the usefulness of traditional benchmarks in evaluating novel architectures.

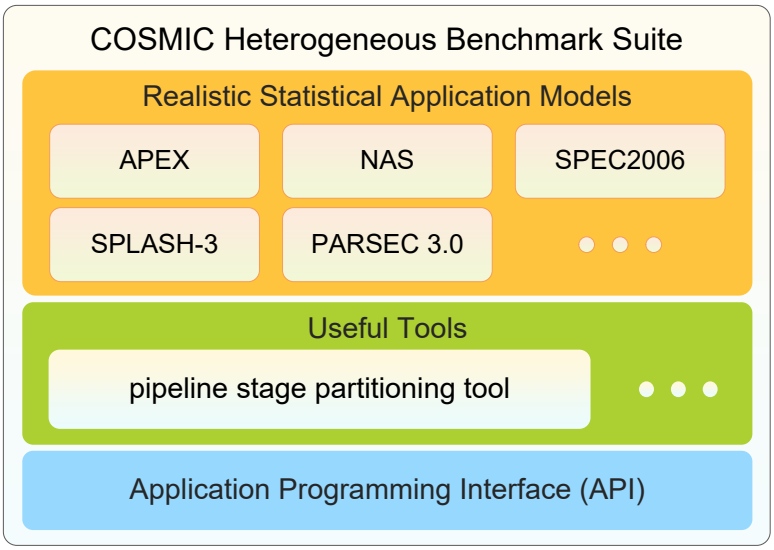


Fig. 1: Content of COSMIC benchmark suite.

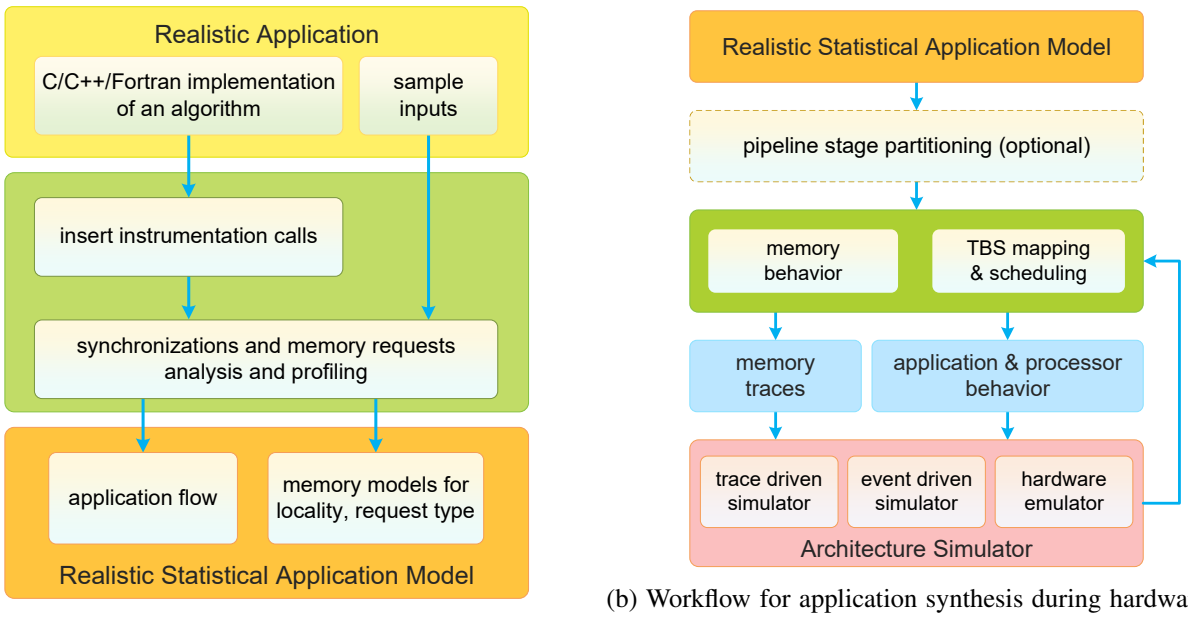
In order to fulfill this gap, we developed a heterogeneous multiprocessor benchmark suite called COSMIC (Communication-Observant Schedulable Memory-Inclusive Computation) that replaces key Operating System (OS) and compiler functionalities, and can be flexibly applied on a variety of architecture exploration platforms. Differently from conventional benchmarks that rely on individual input specification, and require enumeration of a large number of inputs, COSMIC models the application behavior using statistical distributions obtained from profiling the original algorithm implementation, at thread block section level, using a representative number of input datasets. The statistical distributions are used to construct Realistic Statistical Application Models (RSAM). RSAM approximates the overall joint behavior of application and processor by randomly generating task computation time and memory accesses. Due to its randomness, RSAM is useful to replicate many possible application behavior outcomes that could eventually happen and is a more convenient and systematic method to iteratively exercise the system and evaluate the overall average system performance, as well as to identify corner case conditions.

Fig. 1 shows an overview of COSMIC benchmark suite. COSMIC includes statistical application models that are obtained from profiling applications from multiple widely used benchmark suites, including APEX [1], NAS [2], SPEC2006 [3], SPLASH-3 [4] and PARSEC 3.0 [5], for a target processor. Additionally, it provides a useful tool for partitioning an application into multiple stages for pipelined execution during simulation. The COSMIC benchmark can be easily incorporated into existing multiprocessor simulators for high-level multiprocessor design exploration and evaluation by using an easy to use application programming interface (API) provided in C++. For instance, COSMIC has been successfully incorporated in the JADE heterogeneous system simulation platform [6], and also used in various other studies [7].

The rest of the manual is organized as follows. An overview can be seen in Section II. More details about the statistical application model and the profiling is described in Section III, while Section IV illustrates the behavior generation for the realistic models. In Section V, we provide brief description about each application included, and the user's guide on the tools and the API are presented in sections VI and VII, respectively. Section VIII introduces the directory organization and file format of the COSMIC benchmark package.

II. Overview

The overview of how the COSMIC application models are obtained is shown in Fig. 2a. Starting with a real multiprocessor application implemented using high-level languages (e.g, C, C++, and Fortran), we partition the application into multiple thread block sections (TBSs) by inserting instrumentation function calls in the code whenever threads are created, destroyed or synchronized (e.g. barriers, locks, etc). Then, we profile the application to obtain the application flow, which can be represented with a TBS graph, and extract the main behavior statistics with micro-architecture independent metrics for each TBS. Some examples of metrics we profile are the memory locality for data and instruction and the distribution of the request type. The profiled information is used to build what we refer to as Realistic Statistical Application Models (RSAM). These models are later used to reproduce the behavior of the application which can be used to feed, for example, a timing simulator to simulate a target system.



(a) Workflow for application profiling.

(b) Workflow for application synthesis during hardware/software co-simulation.

Fig. 2: Application behavior profiling and synthesis flow.

The COSMIC benchmarks can be easily applied to study heterogeneous systems platforms. During the situation, we generate the application behavior based on RSAM and use it as the stimulus for architecture being studied. This procedure is referred to as synthesis and it is the reverse process of profiling. Fig. 2b shows an overview of how to use the COSMIC application models for synthesis. The application flow is reproduced by traversing the TBS graph. We map each TBS to one core at runtime and synchronizations are issued at the end of the TBS execution. For each TBS, we generate memory requests for data and instructions based on the statistical model generated during profiling. Additionally, we provide an algorithm to partition the application into multiple stages with relatively balanced workload. The user can define their own algorithms to map TBSs belonging to different stages to different chip regions or machines for pipelined execution during the simulation. The behavior generated can be used in several forms. It could be used as stimulus for a even-driven or even trace-driven simulation. For simulations that involves timing, the hardware model should provide some feedback to tune the generation timing information. Figure 3 shows an example of using RSAM to build a simple system simulator. COSMIC has also been integrated in the JADE simulator [6] and is also available online at https://eexu.home.ece.ust.hk/JADE.html.

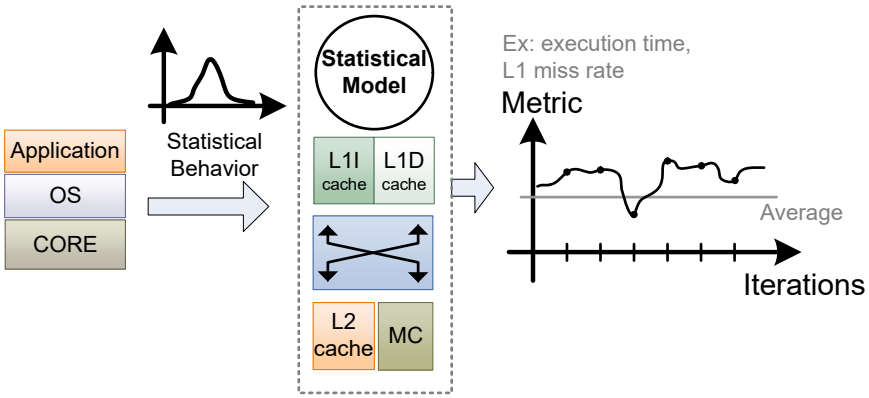


Fig. 3: Example of statistical model usage in a system simulator.

III. Statistical Application Modeling

This section details how the statistical application models are built.

A. Application Modeling

The applications are modeled by the fork/barrier/join model. In this model, the execution flow of the application is divided into parallel sections (PSs). There are three cases where transitions between PSs happen:

- fork: creation of threads;
- join: merging of threads; and
- barrier: synchronization of threads.

We represent the fork/barrier/join model using what we refer as thread block section (TBS) graph, as shown in Fig. 4. This example shows four PSs, each of which is composed of one or more TBSs. We define TBS as a sequential execution section that is executed by one physical thread. The edges in the TBS graph represent the flow dependency between each of the TBSs. Single-threaded application will be represented as a single TBS.

This model helps us to separate the data-sharing into concurrent and sequential data-sharings. In simple words, sequential data-sharing happens when data in one PS is used in another PS, while concurrent sharing is the sharing between TBSs in the same PS.

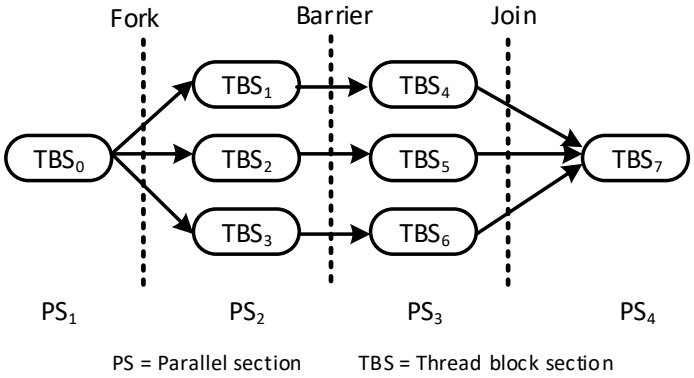


Fig. 4: Application model based on fork/barrier/join.

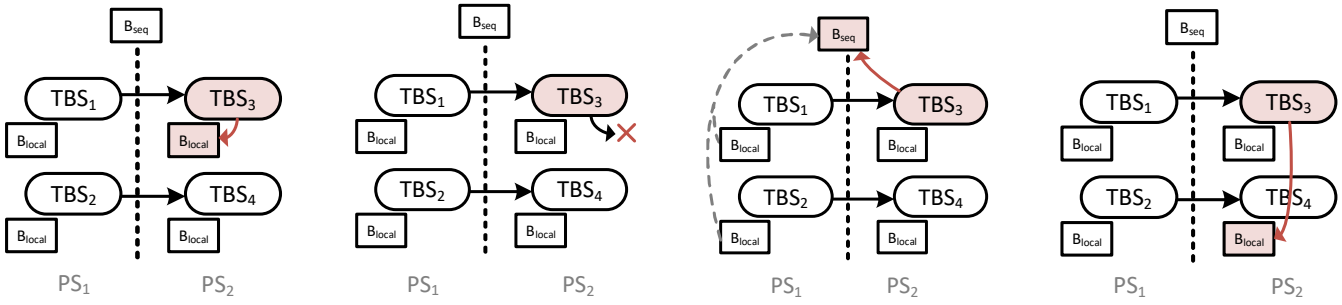
B. Application Profiling

We profile the application during runtime with the goal of obtaining a TBS graph similar to Fig. 4, as well as the profiling information for each of the TBSs. There are two types of information we need to observe during profiling. One type is function calls to synchronizations (i.e. fork, join, and barrier), and we refer to this type as synchronization trace. The second type are the actual memory traces (instructions and data), which will be discussed in subsection III-C.

To obtain the graph, we start by creating the root TBS to represent the master thread in the first PS. Then for each synchronization trace received, we use the physical thread ID (TID) to find the TBS it belongs to. The corresponding TBS is terminated, and the type of the synchronization event and related information are recorded. In the case of a barrier, we keep a unique identifier for it and record how many threads the barrier awaits. In the case of a fork, we record how many threads are spawned and TIDs of the spawned threads. Then, we create new TBSs and assign flow dependency between current and the newly created TBSs. And last, we remap TIDs to the new TBSs.

During profiling, we also inspect critical sections, such as calls to locks and semaphores. For each TBS, we record how many critical sections there are, and how many times the TBS enter the critical section. We record a unique identifier for each lock being acquired/released and how many times that particular lock is used. For semaphores, we also record their initial value and whether the TBS is calling wait or signal.

C. Memory Request Profiling



(a) Local requests. (b) True cold miss. (c) Sequential sharing. (d) Concurrent sharing.

Fig. 5: Different types of memory requests.

COSMIC uses a statistical model to represent the memory locality of each benchmark, called sharing-locality model (Shalom). Shalom is applied to obtain the memory locality for individual TBS to minimize

interference from the others. The output of profiling is the distribution of reuse distance (RD). We profile data and instruction accesses separately. The detailed format of the application models are explained in section VIII. In this section, we give an overview of what kind of information is collected.

As shown in Fig. 5, there are four types of requests. If a TBS accesses a location that has been touched before, it is treated as local, given in Fig. 5a. If a TBS access a memory location for the first time, it is treated as a non-local request. Non-local accesses are further divided into three types. The first type is true cold miss of the application [Fig. 5b]. The second type is sequential sharing, which happens when the address of a non-local access has been used in previous PS [Fig. 5c]. The third type is concurrent sharing, meaning that the non-local memory access is present in another TBS in the same PS [Fig. 5d]. The RD for cold misses is always infinity. For the other three types of requests, we calculate their RD and build the locality distribution model separately.

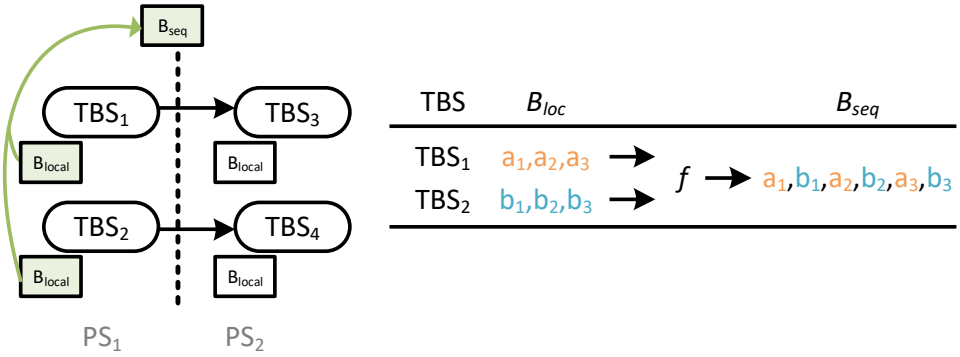


Fig. 6: Merging local buffers into the sequential buffer based on uniform interleaving.

For each TBS, we keep a history buffer to store the local accesses, referred to as B_{loc}. For each PS, we keep a history buffer holding the locations accessed by all previous PSs, represented as B_{seq}. When merging the local addresses touched by multiple TBSs in the previous PS, we assume uniform interleaving. An example of merging local buffers is given in Fig. 6. We keep three histograms for each TBS, which count the RD of local accesses, sequential sharing requests, and concurrent sharing requests respectively. For each memory request, we check whether the address is present in any history buffers and then update the corresponding histogram accordingly. Even though the fraction of non-local accesses is small, they highly affect the accuracy of the simulation. Two major issues arise when ignoring the sharing. First, it will lead to an increased number of cold misses. Second, there will be a similar increase in the application memory footprint. Therefore, capacity misses and conflict misses are going to be equally accentuated. For parallel applications, these source of errors are amplified proportionally to the number of threads.

D. Profiling Setup Details

We profile all applications for two different instruction sets: ARM-v8 and x86_64. The QEMU emulator is used for instrumentation [8], and we develop a profiler called PROFMEM to analyse memory locality and generate the statistical model. During profiling, QEMU and PROFMEM will be launched simultaneously. QEMU sends the instrumented information to PROFMEM through shared memory.

All the profiling have been realized using QEMU user-emulation mode. In this mode, OS and the Linux system calls are not emulated. To enable instrumentation using QEMU we had to do two major changes. The first modification is to annotate the benchmark source code with instrumentation function calls so that QEMU can identify when threads start/end and when they are synchronizing (e.g., barriers and locks). The second modification is in the translation from the target instructions to the host instructions, which enables inspecting every instruction and extracting details such as its type, request size, and memory address. Based on synchronization and memory traces received from QEMU, PROFMEM outputs the

statistical model by applying the Shalom approach. All the applications were compiled using GCC 5.2 with optimizations -O2 whenever possible.

TABLE I: Instruction set details

Instruction Set	Named used	Details
ARM-v8	aarch64	The 64-bit ARM architecture
x86_64	x64	A generic 64-bit version of the x86 ISA.

IV. Statistical Application Behavior Generation

This section details the workflow for application synthesis. The objective of synthesis is to reproduce the application behavior, including the application flow and the memory trace for each of the TBSs.

A. Application Flow

The synthesis should reproduce the flow in the TBS graph. This requires traversing the graph and treating each TBS as a task. During the simulation, we map each TBS to one core at runtime. The CPU dictates when the trace should be generated and issued, and it makes calls to our synthesis algorithm to generate memory requests until we reach the end of the TBS. As in the profiling, the trace is of two types: memory trace or synchronization trace (fork/barrier/join). The memory trace generation is explained in subsection IV-B. Synchronizations are always issued at the end of the TBS execution using the information recorded during profiling. When they occur the simulation should also simulate the primitive. For example, for a fork, we spawn the next TBSs and schedule them in the cores available. During synthesis we also keep the sequential history buffer for each PS, B_{seq}, which is maintained in the same way as in the profiling algorithm. The buffer will be used to generate addresses for memory requests.

B. Memory Trace Synthesis

For each TBS we generate a memory trace length of N that equals the total number of requests of the original realistic application. For each access, we first determine whether it is an instruction or data access based on their proportions obtained during profiling. The next step is to decide the type of access (local, sequential sharing, or concurrent sharing) and sample an address based on the corresponding RD distribution and history buffers. Also, we need to determine the size of the fetch and whether it is a read or write. These choices are obtained following the frequency in which they happen for each type of access (local/sequential/concurrent) during profiling.

C. Critical Section Synthesis

During the generation of each TBS, we enter the critical section proportionally to the number of times they entered in profiling. And we generate the average number of memory requests for that critical section. We assume that critical sections are reached with the same average rate as in the original application for each TBS. The locality inside the critical regions follows the same procedure as explained in subsection IV-B for memory trace synthesis. However, their counters belong to the critical region as the locality inside critical regions tend to differ from the locality in non-critical regions.

TABLE II: List of applications provided

Suite	Application	Description
APEX	hpcg	Conjugate gradient algorithm.
	dgemm	Dense matrix multiplication.
	pennant	Lagrangian staggered-grid hydrodynamics algorithm on 2-D unstructured finite-volume mesh.
	stream	Synthetic benchmark measuring the memory bandwidth and a corresponding computation rate for four simple vector kernels.
NAS	bt	Solving a synthetic system of partial differential equation using block Tri-diagonal solver.
	cg	Estimating the smallest eigenvalue of a large sparse symmetric positive-definite matrix with the conjugate gradient method.
	dc	Data cube operator.
	ep	Generating independent Gaussian random variates using the Marsaglia polar method.
	ft	Solving a 3D partial differential equation using the fast Fourier transform (FFT).
	is	Sorting small integers using the bucket sort.
	lu	Solving a synthetic system of partial differential equation using lower-upper Gauss-Seidel solver.
	mg	Approximating the solution to a 3D discrete Poisson equation using the V-cycle multigrid method.
	sp	Solving a synthetic system of partial differential equation using scalar Penta-diagonal solver.
	ua	Solving heat equation with convection and diffusion from moving ball.
PARSEC 3.0	blackscholes	Option pricing with Black-Scholes partial differential equation.
	canneal	Cache-aware simulated annealing to optimize routing cost of a chip design.
	dedup	Compression with data de-duplication.
	facesim	Simulating the motions of a human face.
	ferret	Content-based similarity search.
	fluidanimate	Simulating fluid dynamics for animation purposes with Smoothed Particle Hydrodynamics (SPH) method.
	freqmine	Frequent itemset mining.
	streamcluster	Online clustering of an input stream.
	swaptions	Pricing of a portfolio of swaptions using the Heath-Jarrow-Morton (HJM) framework.
	x264	H.264 video encoder.

Continued on next page

TABLE II – continued from previous page

Suite	Application	Description
SPEC2006	bzip2	File compression and decompression based on bzip2.
	cactusADM	Solving Einstein evolution equations with staggered-leapfrog method.
	calculix	Finite element method for 3D structural analysis applications.
	dealII	Adaptive finite element method for solving partial differential equations.
	gobmk	Analysing and playing Go game.
	hmmer	Profile hidden Markov models for protein sequence analysis.
	lbm	Lattic Boltzmann method (LBM) for simulating 3D fluids dynamics.
	libquantum	Quantum computer simulation.
	mcf	Single-depot vehicle scheduling.
	namd	Simulating large biomolecular systems.
	omnetpp	Discrete event simulation of a large Ethernet network.
	povray	Ray tracing, rendering algorithm.
	sjeng	Playing chess.
	soplex	Simplex linear program solver.
	specrand	Generating pseudorandom numbers.
	specrand_i	Generating pseudorandom integer numbers.
	sphinx	Speech recognition.
	xalancbmk	Transforming XML documents to other document types.
SPLASH-3	barnes	Simulating N-body problem with Barnes-Hut method.
	cholesky	Blocked Cholesky factorization on a sparse matrix.
	fft	Fast Fourier Transform.
	fmm	Simulating N-body problem with Fast Multipole Method.
	lu	Factorizing a dense matrix into the product of a lower triangular and an upper triangular matrix.
	ocean	Simulating large-scale ocean movements.
	radix	Integer radix sort.
	raytracing	Rendering a 3D scene onto a 2D image plane with ray tracing.
	volrend	Rendering a 3D volume onto a 2D image plane using an optimized ray casting technique developed by Marc Levoy.
	water-nsquared	Simulating the molecular dynamics N-body problem.
	water-spatial	Simulating the molecular dynamics N-body problem.

V. Applications in COSMIC Benchmark Suite

COSMIC contains applications from five commonly used benchmark suites, which are APEX [1], NAS [2], SPEC2006 [3], SPLASH-3 [4], and PARSEC 3.0[5] benchmark suites. The complete list of

benchmarks with their descriptions is summarized in table II.

VI. Useful Tools

A. Pipeline Stage Partitioning

As mentioned in section II, during simulation, we can partition the application into several stages for pipelined execution. As the system scale increases, the performance improvement brought by thread-level parallelism is vanishing due to high synchronization overhead. The introduction of pipelining provides an additional level of parallelism and can potentially lead to more efficient use of hardware resources. When evaluating the performance of large-scale systems, pipelined execution could be the preferred simulation mode. The related python scripts are located in `/home/COSMIC-v4.0/tools/pipeline`.

After partitioning, each pipeline stage consists of one or multiple PSs of the application. During simulation, the application starts at stage 0 and execute each stage sequentially (0, 1, 2, 3 ...). As soon as one stage finishes we re-start that stage. The execution of one stage can be constrained to a specific machine, a specific chip, or a specific region of the chip.

The algorithm for the application partitioning takes into consideration the computation (i.e., number of instructions) and the communication traffic among stages. It is a simple greedy algorithm that tries to minimize the standard deviation of the load for each of the stage. In other words, the algorithm tries to search for a partitioning that attempts to distribute the load evenly across chips or chip regions while considering the communication traffic.

An example on how to use the mapping tool is shown in VI.1.

Example VI.1. Partition the FFT application profiled with four threads into at most 8 stages.

```
./pipelinePartitioning.py splash/fft/4.txt fft_partition.txt 8
```

In the command above, the statistical application model `splash/fft/4.txt` is the input, `fft_partitions.txt` is the output file, and `8` is the maximum number of stages users would like to have. In this case, we will generate partitions with 2, 4, and 8 stages. If the application has fewer PSs than the given number, we will ignore numbers that are larger than the PS count. A sample output file is given as VI.2.

Example VI.2. An example of the output file when the max number of partitions is set to 8.

```
2 4
4 3 4 5
8 1 2 3 4 5 6 7
```

The first column is always the number of stages. The following numbers define the partitions of the parallel sections. The sequence of numbers represent the index of where we put a delimiter for the partition. For example, the first line has "2 4" meaning that there are two stages. Parallel sections with index below 4, which are 0, 1, 2, and 3, are mapped to stage 0 while parallel sections with index equalling or above 4 are mapped to stage 1.

B. Tools installation

Building the COSMIC resources requires a few tools and libraries. Most of current Linux distributions would attend these requirements. Table III lists the recommended set of tools, but users can use your preferred set. Compiling the API is straightforward. We have provided and `CMakeList.txt` that lists the source code. Users can include this file in there makefile.

TABLE III: External dependencies

	Software	Version	Purpose	Quick reference
Required	GCC	5.3.1	Compilation	https://gcc.gnu.org/
	Python	3.6.0	Scripts	https://www.python.org/
	Boost	1.6	Helper library.	https://www.boost.org/

VII. Application Programming Interface

The application models we provide can be used for a variety of studies. Apart from heterogeneous multiprocessor system simulators, COSMIC can be used to evaluate virtual memory assignment, prefetching, cache configurations, and many others. In order to provide a transparent use of the application models, we provide an Application Programming Interface (API) written in C++ to help users read and use the models in the format we provide.

A. Basic data structures

The main data structure that users have to deal with is `TBtbSyn`, which defines the behavior of the TBS. Some basic data members of the `TBtbSyn` class have been listed in Table IV. Users can customize the `TBtbSyn` class by using inheritance as needed.

The `TBtbBarrierInfo` class defines the address of the barrier and the number of threads that should reach it. If current TBS does not reach a barrier, `barrier` points to `NULL`. The `TLocality` class contains the number and locality models of instruction and data accesses, and its data members are listed in Table V. The `TCombinedLocality` contains information about local, sequential sharing, and concurrent sharing accesses.

TABLE IV: Data members of `TBtbSyn` class

Variable Name	Data Type	Description
`btbid`	uint32_t	TBS ID.
`tid`	uint32_t	Thread ID.
`memcount`	vector⟨uint32_t⟩	Number of memory requests in each region.
`cscount`	vector⟨uint32_t⟩	Number of critical sections in each region.
`nextbtb`	vector⟨uint32_t⟩	IDs of TBSs that depend on current TBS.
`dependson`	vector⟨uint32_t⟩	IDs of TBSs that the current TBS depends on.
`btbvec`	vector⟨TBtbIF*⟩	Pointers to all TBSs.
`barrier`	TBtbBarrierInfo*	Pointer to the barrier.
`locality`	vector⟨TLocality⟩	Locality model of the TBS.

B. Load applications

Loading the application using the COSMIC API is straightforward. An code reference is shown in Fig. 7. By calling the function `cosmic::initializeCosmicShalom`, the information of all TBSs will be loaded. Users are required to give as argument containers for instruction and data concurrent access locality models and TBSs.

TABLE V: Data members of `TLocality` class

Variable Name	Data Type	Description
`dcount`	int32_t	Number of data accesses.
`icount`	int32_t	Number of instruction accesses.
`datacloc`	TCombinedLocality	Sharing-locality model for data accesses.
`instcloc`	TCombinedLocality	Sharing-locality model for instruction accesses.

```
TConcLocality icshare;
TConcLocality dcshare;
std::vector<TBtbIF*> btbvec;

cosmic::initializeCosmicShalom(
        "x64/splash/fft/4.txt",
         &icshare, &dcshare, btbvec);
```

Fig. 7: Refernece C++ code for initialization.

VIII. Directory and Files

In this section, we introduce the directory organization and the format of statistical models in the COSMIC benchmark suite.

A. Directory organization

Table table VI shows organization of the root directory. For each application, we provide its statistical models profiled for ARM-v8 and x86_64. As these models are developed based on existing programs, we do not include the source code files in our folder. Under the folder `tools`, we provide the scripts for application partitioning and API for COSMIC.

B. File formats

The statistical models are saved as files with *.txt* extension. The file contains a header and profiling information for each TBS. We discourage users from manually reading these files since they can get a bit large and it uses many sparse vector formats that difficult the reading. Instead, users can refer to the initialization functions in the API to check the order in which each data structure is initialized.

The header contains the file name and developers information. After that, profiling information for each TBS is stored in one line. For each TBS, we start with its TBS ID followed by the relevant information

TABLE VI: COSMIC root directory organization

Class	Directory Name	Description
User space	applications	Contains all application models profiled for different ISAs with various thread counts.
	documents	Documentation and user instructions.
Tools	tools	COSMIC API and tools for designers.

for that TBS. The sequence of the information is:

- TBS ID
- thread ID
- number of regions
- memory trace length of each region
- instruction and data locality information of each region
- number of critical sections in each region
- information of next TBSs

The instruction and data locality information follow the same sequence. They are:

- number of total requests, including local and non-local ones
- average request size
- mean of address accessed
- the standard deviation of address access
- locality information of local requests
- locality information of sequential sharing requests
- locality information of concurrent sharing requests

The locality information of local requests follows this sequence:

- number of local requests
- markovian RW information (see the HRD [9] for details)
- flag of whether we are using sparse (1) or dense (0) vector
- output of the hierarchical reuse distance histogram (two layers)

The locality information of sequential sharing requests follows this sequence:

- number of sequential sharing requests
- number of write requests for sequential sharing
- flag of whether we are using sparse (1) or dense (0) vector
- output of the reuse distance histogram

The locality information of concurrent sharing requests follows a similar format as that of sequential sharing requests. The only difference is that sparse vector is always used, so the flag for indicating the vector format is not required.

In the last section, which is information for following TBSs, we first decide whether it is a barrier or not. In the case of a barrier, the sequence is:

- -1 (indicating it is a barrier)
- address of the barrier
- number of threads that should hit this barrier
- ID of the next TBS

In other cases, we list the number of next TBSs followed by IDs of next TBSs.

IX. Agreement and License

COSMIC copyrights can be found in its root directory. If you use COSMIC in your research, please cite the following paper (https://doi.org/10.1109/HPCA.2017.11):

Rafael Kioji Vivas Maeda, Qiong Cai, Jiang Xu, Zhe Wang, Zhongyuan Tian, *"Fast and Accurate Exploration of Multi-Level Caches Using Hierarchical Reuse Distance,"* in Proceedings of IEEE Symposium on High Performance Computer Architecture (HPCA), Austin, USA, February 2017.

X. Revision History

TABLE VII: Revision history

Version	Changes	Date
1	Initial public release.	1-Jun-13
1.1	Application and mapping and scheduling tools enhancements.	1-Jun-14
2	Intra-task statistical memory behavior introduced.	1-Feb-16
3	Adoption of hierarchical reuse distance. Inclusion of larger benchmarks.	1-Aug-18
3.1	Introduction of the application tuner tool.	1-Sept-18
4.0	Adoption of sharing-locality model. Inclusion of applications in SPEC2006, SPLASH-3, PARSEC 3.0, and NAS benchmark suites.	5-Mar-20

XI. Copyright

References

[1] "APEX benchmarks," https://www.lanl.gov/projects/apex, accessed 28-Feb-2020.
[2] "NAS parallel benchmarks," https://www.nas.nasa.gov/publications/npb.html, accessed 28-Feb-2020.
[3] "SPEC2006 benchmarks," https://www.spec.org/cpu2006, accessed 28-Feb-2020.
[4] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 101–111.
[5] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.
[6] R. K. V. Maeda, P. Yang, X. Wu, Z. Wang, J. Xu, Z. Wang, H. Li, L. H. K. Duong, and Z. Wang, "JADE: A Heterogeneous Multiprocessor System Simulation Platform Using Recorded and Statistical Application Models," in *Proceedings of the 1st International Workshop on Advanced Interconnect Solutions and Technologies for Emerging Computing Systems*, ser. AISTECS '16. New York, NY, USA: ACM, 2016, pp. 8:1–8:6. [Online]. Available: http://doi.acm.org/10.1145/2857058.2857066
[7] Z. Wang, W. Liu, J. Xu, B. Li, R. Iyer, R. Illikkal, X. Wu, W. H. Mow, and W. Ye, "A Case Study on the Communication and Computation Behaviors of Real Applications in NoC-based MPSoCs," in *IEEE Computer Society Annual Symp. VLSI*, 2014.
[8] F. Bellard, "QEMU, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, 2005, p. 46.
[9] R. K. Maeda, Q. Cai, J. Xu, Z. Wang, and Z. Tian, "Fast and accurate exploration of multi-level caches using hierarchical reuse distance," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 145–156.