COSMIC Heterogeneous Multiprocessor Benchmark Suite user manual

Version 1.0
Mobile Computing System Lab
Department of Electronic and Computer Engineering
Hong Kong University of Science and Technology
http://www.ece.ust.hk/~eexu/
June 2014

Multiprocessor/multicore systems as promising choices to provide high computing power for modern emerging applications, are attracting increasing amount of attention and research efforts. By integrating different types of processing elements on a chip, the heterogeneous multiprocessor systems can potentially offer better tradeoffs among energy efficiency, performance, flexibility, scalability and cost than traditional homogeneous multiprocessor systems. Benchmarks play an important role in different aspects of studies on multiprocessor system, from hardware architecture design to software parallelization strategy developments. However the design exploration of heterogeneous multiprocessor systems is not well supported by traditional benchmarks which rely on compilers and operating systems developed for homogeneous multiprocessor systems. And a specific compiler or OS are generally not available for a novel heterogeneous system, especially at the early design stage. All these factors limit the use of traditional benchmarks in the design exploration and evaluation of novel heterogeneous multiprocessor architectures.

![COSMIC benchmark suite](image)

Fig. 1: Content of COSMIC benchmark suite

To this end, we developed a heterogeneous multiprocessor benchmark suite called COSMIC (Communication-Observant Schedulable Memory-Inclusive Computation), which can be flexibly applied on various multiprocessor systems and effectively accelerate the design process. The new benchmarks are adaptive to both heterogeneous and homogeneous multiprocessor systems, and can be statically and dynamically scheduled for execution by users. The COSMIC is developed based on 7 realistic multiprocessor applications. Besides implementing the applications in high-level programming languages, COSMIC uses a formal computational model called task communication graph (TCG), to explicitly capture the computation and communication requirements of the applications. And using TCG models in addition to real application programs allows us to bypass operating systems (OS) and compilers to quickly explore novel multiprocessor architectures. As showed in Fig. 1, the COSMIC benchmark suite includes the application TCG models,
the corresponding high-level programs of the applications and sample input datasets for the applications. Besides these, COSMIC provides a tool for application task scheduling and memory allocation. The COSMIC benchmark suite can be easily incorporated into existing multiprocessor simulators for high-level multiprocessor design exploration and evaluation.

The rest of the manual is organized as follows. Section I illustrates the generation methodology of COSMIC benchmark suite. In section II, we provide detailed description about each application included in COSMIC benchmark suite. The last section introduces the directory organization and file format of the COSMIC benchmark package.

I. APPLICATION MODELING AND COSMIC GENERATION

An overview of the generation method is shown in Fig. 2. The process starts with the real multiprocessor application algorithms. We analyze the algorithms of the real applications, implement them in high-level languages, such as C and C++, or find their existing implementations, and develop different sample inputs for each application. We further verifying the functionality and accuracy of the provided application programs. Based on the analysis and implementation, we perform computation and communication behavior modeling for each real application. The modeling process mainly include three steps: application task partitioning, task profiling and task communication analysis. After these steps, the computation and communication behaviors of applications are modeled as TCGs. To execute the modeled applications on multiprocessor platforms, two necessary steps are required, one is the allocation of proper memory resources for each application, and the other is the scheduling of application tasks onto different processors for execution. For this end, we provide a sample tool for memory allocation and task scheduling and specifically optimized these two steps to improve the performance. More details about the methodology can be found in our previous publication [1].

![Diagram](image)

Fig. 2: An overview of the COSMIC benchmark generation methodology.

A. High-level program implementation and input datasets

In the suite, benchmarks for 7 real applications are included, which are FFT, Reed-Solomon code, TURBO decoder, LDPC decoder, Molecular Dynamics, Ultrasound, and Ray Tracing. Detailed information
about these applications are presented in section II. We analyze the application algorithms and explore for a suitable high-level program implementation to facilitate the subsequent application modeling process. By collaborating with specific domain experts on these applications respectively, we either develop the high-level programs or use suitable existing ones in our benchmark suite.

Along with the high-level programs, associated sample input datasets are provided in the benchmark suite. On one hand, these input datasets help to accurately profile application tasks without losing generality, and on the other hand, they are used to validate the functionality of the application programs.

B. Application modeling

The applications are modeled by a weighted directed acyclic graph model, called task communication graph. The TCG is defined as a tuple $G_t = (V, E)$, where $V$ is the set of weighted vertices denoting the computational tasks, and $E$ is the set of weighted directed edges denoting the communication channels among tasks. Each task $v_i \in V$ has a worst-case execution time $t_i$, which is measured in the number of clock cycles. Each edge $e_i = (v_{i,s}, v_{i,d}, \omega_i) \in E$ has a source task $v_{i,s}$, a destination task $v_{i,d}$, and the amount of data $\omega_i$ sent from $v_{i,s}$ to $v_{i,d}$, which is measured in number of words (32 bits in this work). The execution time and communication message size are measured on a baseline processor while they can still be applied on heterogeneous systems with different types of processing units. We will elaborate this in section I-B3. A TCG example is showed in Fig. 3, in which the number next to each task refers to the worst-case execution time, and the number on each edge refers to the communication volume. There are 11 tasks and 15 edges in this TCG.

![Task Communication Graph](image)

Fig. 3: An example task communication graph.

1) Application partitioning: Applications are partitioned into various tasks for execution. Generally, the applications are partitioned in different manners according to specific needs of designers. We systematically analyze the algorithm of each application to help decide how to properly partition the application into multiple tasks.

We partition the applications into multiple tasks in a fine-grain manner in order to better explore the potential execution parallelism. Besides, fine-grain models could be transformed by exploiting the task clustering or grouping techniques, to merge multiple elementary tasks into larger ones based on specific needs or design requirements. For the application programs, not all the instructions are packaged into some tasks, instructions performing structural operations, such as those updating loop iterators and interacting with memories, will be translated as inter-task relations and reflected in the structure of the TCG. The instructions inside a task must be executed sequentially in the same processor without preemption. Besides this, for conditional bodies (e.g. "if-else" in C), all instructions contained inside are treated as one task to comply with the definition of a directed acyclic graph.
2) **Dependency graph generation:** Task dependencies are mainly reflected by data communication between different tasks. So the dependency analysis are conducted in pairwise. At code level, tasks with the same operands or using pointers referring to the same memory address have potential dependencies. And for the loop entity, as TCG is acyclic, each iteration needs to be unfolded and the interactions among different iterations are extracted as the task dependencies. And for tasks wrapped as functions, the function calls and their inner sub-calls are completely analyzed to decide all possible dependencies.

3) **Task profiling:** The partitioned task programs are compiled and executed on a processor simulator called SimpleScalar [2] to obtain the worst-case execution time and output data volume of each task. The SimpleScalar processor emulates PISA architecture, which is similar to MIPS-IV ISA [3]. Performances based on SimpleScalar (SS) serve as comparison baselines for new processors, accelerators, and other processing units. When COSMIC is used to explore heterogeneous systems with different types of processing elements or architectures, an attribute, called acceleration factor, can be defined for each processing unit executing different tasks to reflect the relative speedup of the task execution on it compared to the baseline SS processor. The values of acceleration factors are user-defined, and can be derived empirically or by analyzing the differences between the targeted processing unit and the baseline SS processor architecture. So the actual execution time of the tasks on a processing unit are determined by both the baseline execution time we provided and the user-defined acceleration factor of the processing elements.

The worst-case task execution time is measured in number of clock cycles instead of absolute timing for the ease of cycle-level simulation. Since the theoretical worst-case execution time is difficult to determine, we resort to empirical approaches to approximate the worst case. Firstly the application is executed for multiple times with different input datasets and every time the execution time for each task is recorded. We choose the largest execution time for each task and add it with a 10% margin to be the final worst-case time approximation.

The communication volumes between tasks are calculated mainly by recording the variables used for transferring data between task functions, including function parameters and global variables. There are two special cases here, one is dealing with user-defined data types, and the other is the use of pointers. In both of these situations, we statically calculate the sizes of the specific memory spaces defined in the source program.

**C. Memory allocation and task scheduling tool**

The memory allocation and task scheduling tool performs as a bridge to link the application TCG models to the multiprocessor hardware systems. What we provided is a sample such tool which is specifically optimized for different multiprocessor architectures to improve performance, and it have been tested to be functional in many practical works. Still, users are free to develop their own tools to apply the COSMIC benchmarks according to their specific design requirements. We also provide the source codes of this tool for reference.

![Fig. 4: 16-core systems with three different regular-topology NoCs](image-url)
We assume that a virtual private memory space is assigned to each edge to store the data generated by the source task. Many applications, such as the RS and TURBO, run iteratively rather than for once in real systems, thusly insufficient memory allocation can limit the application throughput and affect the overall performance. Hence it is very beneficial to properly determine the memory requirements on each communication edge such that the performance can be maximized with minimum memory resources allocated in total. Basically, we apply genetic algorithms to find the minimum memory space that will make no negative impact to the application performance. There are two objectives to be optimized: maximizing application throughput in higher priority and minimizing total memory size in lower priority. We apply genetic algorithms to explore possible memory size allocations, evaluate them by calculating the theoretical throughput of the TCG under the memory constraint, and conduct these two steps iteratively until a satisfactory result is obtained.

Task mapping and scheduling is optimized specifically for different multiprocessor systems. This tool targets NoC (Network-on-Chip) based homogeneous multiprocessor systems in different sizes. Systems with three regular NoC topologies are considered, which are mesh, torus and fat tree. Detailed system configurations are listed in Table I. The multiprocessor systems with these three regular-topology NoCs are illustrated in Fig. 4. Note each processing block (PB) is identified by the rule shown in Fig. 4, and it will be used to present the mapping result which will be further described in subsequent sections. It is worth noting that optimizations for heterogeneous multiprocessor systems can be supported as well by modifying the source codes accordingly.

<table>
<thead>
<tr>
<th>NoC topology</th>
<th>Size (number of processors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mesh</td>
<td>2x2, 2x4, 3x3, 4x4, 5x5, 4x8, 6x6, 7x7, 8x8, 9x9, 10x10, 11x11, 8x16, 12x12, 13x13, 14x14, 15x15, 16x16</td>
</tr>
<tr>
<td>Torus</td>
<td>2x2, 2x4, 3x3, 4x4, 5x5, 4x8, 6x6, 7x7, 8x8, 9x9, 10x10, 11x11, 8x16, 12x12, 13x13, 14x14, 15x15, 16x16</td>
</tr>
<tr>
<td>Fat tree</td>
<td>4, 8, 16, 32, 64, 128, 256</td>
</tr>
</tbody>
</table>

We assume a static and centralized scheduling strategy to manage the entire chip resources and coordinate PBs. In this way, the scheduling and control decisions made are globally optimized for the whole system. We develop a load balanced mapping and static order scheduling approach. The basic idea is to distribute processing and network transmission workloads evenly to achieve high utilization of the hardware resources. The objective is to minimize the application’s end-to-end execution time with network communication overhead taken into consideration. More details can be found in [1].

II. APPLICATIONS IN COSMIC BENCHMARK SUITE

This section introduces the 7 real applications in detail, which are FFT, Reed-Solomon code, TURBO decoder, LDPC decoder, Molecular Dynamics, Ultrasound, and Ray Tracing. For some of the applications, multiple versions of TCG models are generated with different execution configurations or specifications, such as input data sizes, simulation initial states and so on. A overview of them is tabulated in Table II.

A. FFT

FFT, known as Fast Fourier Transform, have been widely used in many applications in engineering, such as digital signal processing and control system. Typical FFT has $2^N$ inputs and can converts time/space to frequency and vice versa. In the benchmark suite, time to frequency transforming is made in radix-2 with the 1024 inputs and each input is a complex number. We implement the application in C++, and model the application by the method described in last section.
TABLE II: Information of the realistic applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Benchmark name</th>
<th>Description</th>
<th>No. tasks</th>
<th>No. comm. links</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>FFT-1024_complex</td>
<td>Fast Fourier Transform with 1024 complex number inputs</td>
<td>16384</td>
<td>25600</td>
</tr>
<tr>
<td>TURBO decoder</td>
<td>TURBO-dec</td>
<td>Turbo code decoder with 40-bit data block and 1/3 coding rate. Developed with Prof. Wai Ho MOW</td>
<td>33257</td>
<td>86208</td>
</tr>
<tr>
<td>LDPC encoder</td>
<td>LDPC-enc</td>
<td>Low-Density Parity-Check code encoder with 64-bit codeword and 1/2 coding rate. Developed with Prof. Wai Ho MOW</td>
<td>1283</td>
<td>3331</td>
</tr>
<tr>
<td>Reed-Solomon code</td>
<td>RS-32_28_8_enc</td>
<td>Reed-Solomon code decoder with codeword format RS(32,28,8)</td>
<td>262</td>
<td>348</td>
</tr>
<tr>
<td></td>
<td>RS-32_28_8_dec</td>
<td>Reed-Solomon code decoder with codeword format RS(32,28,8)</td>
<td>182</td>
<td>392</td>
</tr>
<tr>
<td>Molecular Dynamics</td>
<td>MD-144</td>
<td>Computer simulation of physical movements of an Argon atom scattered in a Platinum atom box. The box contains 144 Platinum atoms. Developed with Prof. Wenjing Ye</td>
<td>11320</td>
<td>53803</td>
</tr>
<tr>
<td></td>
<td>MD-512</td>
<td>Computer simulation of physical movements of an Argon atom scattered in a Platinum atom box. The box contains 512 Platinum atoms. Developed with Prof. Wenjing Ye</td>
<td>134416</td>
<td>662291</td>
</tr>
<tr>
<td>Ultrasound</td>
<td>US-2D_39x23</td>
<td>Medical diagnostic algorithm using 2D ultrasound imaging. The resolution of input ultrasound images is 39x23. Developed with Prof. Weichuan Yu</td>
<td>81</td>
<td>204</td>
</tr>
<tr>
<td></td>
<td>US-2D_201x200</td>
<td>Medical diagnostic algorithm using 2D ultrasound imaging. The resolution of input ultrasound images is 201x200. Developed with Prof. Weichuan Yu</td>
<td>31265</td>
<td>83780</td>
</tr>
<tr>
<td></td>
<td>US-3D_9x5x10</td>
<td>Medical diagnostic algorithm using 3D ultrasound imaging. The resolution of input ultrasound images is 9x5x10. Developed with Prof. Weichuan Yu</td>
<td>405</td>
<td>550</td>
</tr>
<tr>
<td></td>
<td>US-3D_30x50x20</td>
<td>Medical diagnostic algorithm using 3D ultrasound imaging. The resolution of input ultrasound images is 30x50x20. Developed with Prof. Weichuan Yu</td>
<td>24157</td>
<td>33578</td>
</tr>
<tr>
<td>Ray Tracing</td>
<td>RT-10x10</td>
<td>3D scene rendering algorithm. The resolution of input images is 10x10. Developed with Prof. Pedro V. Sander</td>
<td>7983</td>
<td>20838</td>
</tr>
<tr>
<td></td>
<td>RT-50x50</td>
<td>3D scene rendering algorithm. The resolution of input images is 50x50. Developed with Prof. Pedro V. Sander</td>
<td>206836</td>
<td>540407</td>
</tr>
</tbody>
</table>

Fig. 5 gives a overview of the topology of the generated task communication graph of FFT. There are 4 types of tasks in the TCG, where tasks marked as A are complex adding, S are complex subtracting, M are complex multiplication and I are data input.

Fig. 5: Task communication graph of FFT-1024_complex.
B. Reed Solomon encoder and decoder

Reed-Solomon codes [4] (RS), are non-binary cyclic error-correcting codes that could detect and correct multiple random symbol errors. They have important applications from deep-space communication to consumer electronics, especially in storage systems such as CDs, DVDs, Blu-ray Discs, in data transmission technologies such as DSL and WiMAX, as well as in computer applications such as RAID 6 systems [5]. In the benchmark suite, the RS codeword format is RS(32, 28, 8), denoting that the codeword length is 32 symbols with 28 symbols and each symbol is composed of 8 bits. This type of RS code is able to detect at most 3 symbol errors and correct at most 2 symbol errors. The decoder program uses Euclidean decoding algorithm and Chien Search method [6]. The corresponding encoding algorithm resembles the shift register scheme in hardware implementation. We implemented these two application in C language based on the original program of [7].

![Diagram](image1.png)

(a) Task communication graph for RS-32_28_8_dec.  
![Diagram](image2.png)

(b) Task communication graph for RS-32_28_8_dec.

Fig. 6: Task communication graph of RS encoder and decoder

Fig. 6(a) and Fig. 6(b) respectively show the basic structure of the task communication graphs for RS-32_28_8_dec and RS-32_28_8_enc. The figure shows the encoding/decoding process of only one codeword (32 bits). Due to the complex structure of the graph, we do not mark out all the task types, but only the task ID numbers which are in line with the TCG documents (.app files in the package). Most of the tasks contained in these two application are adding and multiplication in GF(8). And these basic operations in GF(8) can be efficiently completed by looking up a pre-calculated table.

C. TURBO decoder

Turbo coding is also a high-performance forward error correction algorithm widely used in today’s communication field. Since it can nearly reach the theoretical maximum value of the Shannon code

---

1 the Galios Field, or finite field, where 8 elements compose a symbol unit.
rate, it is becoming increasingly popular in modern communication systems such as 3G, 4G, satellite communication as well as data transmission technologies such as IEEE WiMAX. The turbo decoding mechanism consists of two elementary decoders, which are serially interconnected to each other with an inter-leaver installed between them as an coordinator and arbitrator. Ones output will be generated as the others input. Through several iterations, the decoder ends when some criteria have been met. In the benchmark suite, we derive the TCG model based on the turbo decoding application in 3GPP standard [8], the block size of which is 40 bits and it adopts double sets of parity checking bits and the Viterbi algorithm. We implemented the application in C language based on the original program of [9].

Fig. 7 gives an overview of the the task communication graph of one decoder. The application is divided into 7 types of tasks: data input (I), floating point multiplication (M), three-input addition (A), forward state metrics calculation (F), backward state metrics calculation (B), extrinsic information calculation (E), interleaver/deinterleaver (I/D).

![Task Communication Graph of Turbo-dec.](image)

**Fig. 7: Task communication graph of Turbo-dec.**

**D. LDPC encoder**

LDPC codes, standing for Low-Density Parity-Check code, is another important linear error correcting code, which is constructed using a sparse bipartite graph. LDPC codes are widely adopted in modern information transmission systems, such as the new DVB-S2 standard for the satellite transmission of digital television, the Wi-Fi 802.11 standard and so on. And in the last few years, the LDPC codes have shown advances than other coding methods in terms of performance in higher code rate range. In the benchmark suite, we implemented the LDPC fast encoding using sparse matrix methods to encode 1 kb original data which are divided into 256 blocks with 1/2 coding rate. We implement the application in C language based on the original program of [10].

Fig. 8 shows part of the task communication graph of the application *LDPC-enc*. There are 8 types of tasks in it, which are Reading Parity-check matrix (RP), Allocating Memory (AM), LU-Decomposition (DC), data Input (I), Calculation of x (Cx), Calculation of y (Cy), Calculation of checking bits (Cc) and Merging results (MG). The major tasks solve matrix equations, and this is the core part of the sparse matrix encoding algorithm.
Fig. 8: Task communication graph of LDPC-enc.

E. Molecular Dynamics

The name of Molecular Dynamics (MD) stands for a class of processes simulating the interaction between atoms and molecules, and MD is always a popular application for high-performance computing systems. MD includes a variety of specific molecular processes. In the benchmark suite, we studies the simulation that capture the dynamic characteristics of an Argon gas atom scattering on the Platinum wall atoms composing a small box. We modeled the processes under two different Platinum box sizes, one small sized box composed of 6x6x4 atoms and a large one composed of 8x8x8 atoms. We provided the TCG models of the application with these two different simulation configurations. The movement of atoms are determined by solving Newton’s equations of a molecular system where forces between the particles can be found by looking at the potential energy [11]. "search all" method is used to compute inter-atom forces, and we use the Leapfrog method to integrate the equations of motion. The detail of the C code of the application can be found in [12].

Fig. 9 shows part of the task communication graph of the application MolecularDynamics. The application has 9 types of tasks: data input (I), LeapFrog algorithm (LF), Boundary Condition checking (BC), Force Computing algorithm (FC), Energy Calculation (EC), Output Molecule behavior (OM), Energy Accumulation (EA), Temperature Adjustment (TA), Temperature Offset computing(TO).

F. Ultrasound

Ultrasound (US) is an advanced approach used in medical diagnosis. Ultrasound imaging system is becoming a popular medical modality for examining the human body. It has the advantage of being non-invasive compared to other examination methods such as X-ray and surgery, and it is more portable which makes it handy in a variety of situations. The application we included in the suite is the ultrasound image
analyzing algorithm, which is usually applied on 2 ultrasound images (either 2D or 3D) to scan, detect and track tissue motions based on specific image patterns called speckles. We developed the application based on the work in [13], and details about the algorithm can be found in this work as well. The algorithm tries to find the pixel motions which starts with some independent pixels and iteratively spread over the image. Each iteration is called a processing scale, and in this work at most 7 scales can be processed. We respectively modeled the algorithms for 2D and 3D ultrasound images in different sizes which are described in II. And multiple TCG models are generated for this application.

Fig. 9: Task communication graph of MolecularDynamics.

Fig. 10: Task communication graph of 2D Ultrasound.

Fig. 10 shows a sample structure of the task communication graph for 2D images. Each task spacially corresponds to the processing for a certain pixel. There are 4 types of tasks in the TCG: calculation for
optimal motion for Independent Point (IP), Horizontal Point (HP), Vertical Point (VP) and Middle Point (MP).

G. Ray Tracing

Ray Tracing is a 3D computer graphics rendering technique. The main idea of the algorithm is to determine the color of each pixel by tracing a ray from an imaginary eye through the pixel to the light sources and examining its interactions on the 3D objects in the scene. In this benchmark, the application traces a ray from the camera through a certain pixel, and then find the intersection of the ray and the object. Combined with information of estimated light sources and object material properties, we can finally determine the color of the pixel. The primary ray which are directly generated from the camera, can produce secondary rays by refraction or reflection on the surface of the objects. In this work, we constrain the refraction and reflection of the primary ray will at most happen for 5 times. Compared with other rendering techniques such as Scanline Rendering and Ray Casting, Ray Tracing can produced more photorealistic images, but as a trade-off, its performance in term of runtime is considerably degraded and thus not suitable for real-time application like video games. Ray Tracing is commonly used for high-quality still images and special effects of films. The heavy computational cost and algorithm complexity make Ray Tracing a suitable choice for multiprocessor system benchmarking. In the suite, we respectively modeled the Ray Tracing processes with two input scenes in different resolutions, one is in low resolution of 10x10 pixels, and the other is in high resolution of 50x50 pixels. And we provide TCG models for Ray Tracing application with these two different configurations.

![Diagram](image)

Fig. 11: Task communication graph of RayTracing.

Since the rendering process of each pixel is independent, we only show the task communication graph structure for rendering one pixel in Fig. 11. The application has 8 types of tasks: Application Initialization
(AI), Pixel Initialization (PI), Intersection Test (IT), Phong Shading (PS), Reflection Generation (RlG), Refraction Generation (RrG), Color Summing (CS) and Post Processing (PP).

III. DIRECTORY ORGANIZATION, FILE FORMATS OF COSMIC BENCHMARK SUITE

In this section, we introduce the directory organization, the format of each file included in the COSMIC benchmark suite. For instructions on usage, users may refer to [1] which presented more examples and a case study.

A. Directory organization

![Diagram of the directory structure](image)

* More details about the tool directory structure can be found in the README file inside this folder

Fig. 12: File directory structure of COSMIC benchmark suite.

Fig. 12 shows the file directory structure of the COSMIC suite package downloaded online. The files are sorted by different applications. For the high level program of each application, we provide the README file to illustrate how to compile and execute. Since some of the application programs are developed based on existing programs, we also have acknowledged the original programs in the README files and source code files. The sample inputs are basically encoded as ASCII format, for some applications we also provide the configuration files. (users can refer to the README file of each application for more details) For the memory allocation and task scheduling tool, the usage instruction as well as copyright information can be found in the README file in the same folder. In the ‘arc’ folder of the tool, we provide the architecture models of the systems with three NoC topologies and different sizes as aforementioned. Executing the tool could output two types of results: the memory allocation results (with `.buf` extension) and task mapping and scheduling results (with `.map` extension), and they will be generated in the ‘res’ folder.

B. File formats

The format of the application TCG model files (with `.app` extension) is shown in Table ???. We call a data block generated by one time execution of the source task on the edge a “message”, and the message size is given in words (32 bits). The list of starting tasks are the tasks with no preceding tasks or incoming edges, and the list of finishing tasks are the tasks with no succeeding tasks or outgoing edges.
TABLE III: The format of the application model file.

<table>
<thead>
<tr>
<th>Header block (4 line)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>application name</td>
<td></td>
</tr>
<tr>
<td>number of tasks</td>
<td>number of edges</td>
</tr>
<tr>
<td>number of starting tasks</td>
<td>list of starting tasks</td>
</tr>
<tr>
<td>number of finishing tasks</td>
<td>list of finishing tasks</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task execution block (number of tasks lines, each of which is as follows)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>task id</td>
<td>worst-case execution time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task communication block (number of edges lines, each of which is as follows)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>edge id</td>
<td>source task id</td>
</tr>
</tbody>
</table>

1 Unit of task execution time: clock cycle
2 Unit of message size: word or 32 bits

The file format of the architecture model files (with ‘.arc’ extension) are simple, so we do not provide detailed description for it. In an architecture file, each line describes a parameter of system configuration, and the corresponding attribute is self-explained by its name.

TABLE IV: The format of the memory allocation result file.

<table>
<thead>
<tr>
<th>Header block (1 line)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>number of edges</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result block (number of edges lines, each of which is as follows)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>edge id</td>
<td>allocated buffer size</td>
</tr>
</tbody>
</table>

1 Unit of memory size: word or 32 bits

The format of the tool’s output file for memory allocation result (with ‘.buf’ extension) is described in Table IV. The memory size is given in number of words. And we assume a 64-bit address space in the tool.

The format of the tool’s output file for the mapping and scheduling result (with ‘.map’ extension) is described in Table V. Specifically, the mapping result will be given by a two-dimensional coordinate in the format of \((x, y)\) if the topology is mesh and torus, and a single PB ID number if the topology is fat tree.

TABLE V: The format of the statistical traffic trace file.

<table>
<thead>
<tr>
<th>Header block (2 line)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>number of PBs</td>
<td></td>
</tr>
<tr>
<td>number of tasks</td>
<td>number of edges</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result block (number of tasks lines, each of which is as follows)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>task id</td>
<td>mapped PB id / coordinate</td>
</tr>
<tr>
<td>(\mu_t)</td>
<td>(\sigma_t)</td>
</tr>
</tbody>
</table>

1 The mapping result is given by a two-dimensional coordinate \((x, y)\) in mesh and torus, and a single PB ID number in fat tree.
C. Usage instruction

The COSMIC benchmark suite can be easily used in various research aspects of multiprocessor computing. A sample case study of using the COSMIC benchmarks to evaluate NoC-based MPSoCs is presented in [1].

REFERENCES

[9] 3GPP turbo decoder, developed by Yufei Wu, MPRGlab, Virginia Tech.