MCSL Network-on-Chip Benchmark Suite User Manual

Version 1.5
Mobile Computing System Lab
Department of Electronic and Computer Engineering
Hong Kong University of Science and Technology
http://www.ece.ust.hk/~eexu/
January 2013

Network-on-chip (NoC) traffic patterns are essential tools for NoC performance assessment and architecture exploration. The fidelity of NoC traffic patterns has profound influence on NoC studies. Random traffic patterns use probability distributions to randomize on-chip communication traffic characteristics, such as packet destination and transmission interval. Configuring parameters properly for random traffic requires comprehensive knowledge of the corresponding realistic traffic patterns, and it is in general hard to truly reflect realistic traffic characteristics. On the other hand, realistic traffic patterns are based on the behaviors of real applications, and they can provide more accurate performance and power consumption results, and more comprehensive information to improve NoC architectures.

We provide a NoC traffic benchmark suite for efficient NoC-based multiprocessor system-on-chip (MPSoC) evaluations. The publicly released MCSL NoC benchmark suite includes a set of realistic traffic patterns for seven real-world applications and covers popular NoC architectures, and can be downloaded from [1]. It captures both the communication behaviors in NoCs and the temporal dependencies among them. Each traffic pattern in MCSL has two versions, a recorded traffic pattern (RTP) and a statistical traffic pattern (STP). The former provides detailed communication traces for comprehensive NoC studies, while the latter helps to accelerate NoC explorations at the cost of accuracy. The proposed traffic generation methodology uses formal computational models to capture both communication and computation requirements of applications. It optimizes application memory requirements, mapping and scheduling to maximize overall system performance and utilization before extracting traffic patterns through cycle-accurate simulations. The MCSL NoC benchmark suite provides an essential tool for NoC architecture exploration and evaluation. It can be easily incorporated into existing NoC or multicore/many-core simulators and substantially improve NoC simulation accuracy.

I. Traffic modeling and generation methodology

An overview of the traffic generation methodology is shown in Fig. 1. The generation process starts with an application model and an architecture model. Since MPSoC applications are often performance-sensitive, it is necessary to perform optimizations and performance evaluation for traffic generation. The traffic obtained in this way is essentially different from that obtained via a random approach and is more realistic to reflect the actual communication behaviors in a practical MPSoC design. As shown in Fig. 1, the traffic patterns are generated through four steps: memory space allocation, mapping and scheduling, performance evaluation and traffic generation. Two types of traffic patterns are generated, called RTP and STP. The generation steps interact with each other closely. They are essential for the methodology since these decisions substantially affect the final traffic pattern. Optimized decisions can take full advantage of the parallel hardware resources in MPSoC and improve overall system performance and resource

utilization. The methodology is designed with the flexibility, scalability and extensibility on the choice of these algorithms. More details of the methodology can be found in our previous publication [2].

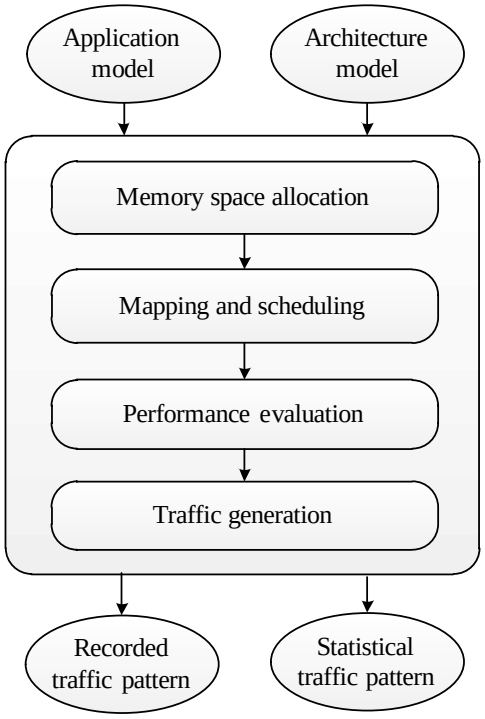

Fig. 1: An overview of the traffic generation methodology.

A. Memory space allocation

Task communication graphs (TCGs) are used to model the applications. In the application model, each edge has a piece of memory space for data exchange between tasks. When the application is executed repeatedly in an iterative fashion, insufficient memory space allocated to the edge can limit the parallelism of the application and impact its performance. Therefore, it is important to determine the memory requirement on the edges to maximize the performance.

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 satisfiable result is obtained.

B. Mapping and scheduling

The traffic generation methodology uses a centralized scheduling strategy to manage the entire chip resources and coordinate processing blocks (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 mapping strategy is to assign tasks to PBs one by one in the topological order defined by the dependency relations in the graph, and the schedule on each PB is determined by the sequence of the tasks on the same PB generated during the mapping.

The objective is to minimize the application's end-to-end execution time with network communication overhead taken into consideration.

C. Generation of traffic patterns

The RTP contains detailed and accurate trace of task executions and communications. It is used for precise and comprehensive NoC studies. A RTP is generated during cycle-accurate simulations for an application model on a NoC simulation platform based on SystemC [3] with the memory space allocation and mapping and scheduling results. It contains more accurate computation and communication traces, where all the task execution and packet generation events are recorded. The RTPs are reusable on NoCs with different configurations but the same topology. Since the exact packet delays among PBs are related with specific NoC configurations, the RTPs keep the packet dependencies instead of exact timings. When the traffic patterns are applied to a different NoC configuration, all the temporal relations can be reconstructed correctly.

The STP gives a concise representation of the information through mathematical modeling. It can be used to support long simulation runs, and is useful for system-level statistical evaluation and analysis. With the results obtained above, the STPs can be synthesized for applications that provide statistical distributions of task executions, packet generations and transmissions. We design the traffic patterns with enhanced reusability compared to previous works.

II. The MCSL NoC benchmark suite

We provide two types of traffic patterns for NoC. RTP is useful for detailed studies as it provides accurate information. STP is useful for long simulation runs to find out average and worst/best characteristics due to its statistical nature. We will describe each type of traffic pattern in detail in the following sections.

A. Application and architecture models

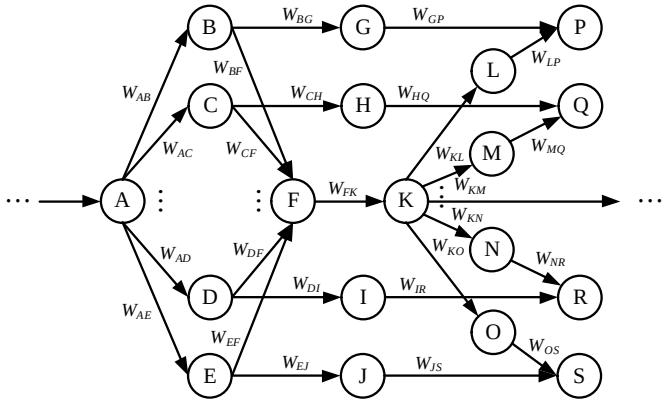

Fig. 2: Part of the H.264 video decoder's task communication graph.

We use the TCG model as the input of the traffic modeling and generation flow to faithfully capture the computation and communication requirements of real applications. A TCG is a directed graph $G_t = (V, E)$, where V is the set of vertices representing computation tasks, and E is the set of edges representing communication links between tasks. A task v has a normalized execution time t. A directed edge $e = (v_s, v_d, w)$ has a source task v_s, a destination task v_d and the amount of data w that sends from v_s to v_d. For example, Figure 2 shows a part of the TCG for H.264 decoder. We include applications from different domains in the MCSL NoC benchmark suite v1.5. Their characteristics are listed in Table I.

TABLE I: The applications included in the MCSL NoC benchmark suite.

Application	Description	Number of Tasks	Number of Communication Links
RS-32_28_8_enc	Reed-Solomon code encoder with codeword format RS(32,28,8)	248	328
RS-32_28_8_dec	Reed-Solomon code decoder with codeword format RS(32,28,8)	278	390
H264-720p_dec	H.264 video decoder with a resolution of 720p	2311	3461
ROBOT	Newton-Euler dynamic control calculation for the 6-degrees-of-freedom Stanford manipulator	88	131
FPPPP	SPEC95 Fpppp is a chemical program performing multi-electron integral derivatives	334	1145
FFT-1024_complex	Fast Fourier transform with 1024 inputs of complex numbers	16384	25600
SPARSE	Random sparse matrix solver for electronic circuit simulations	96	67

An architecture model captures the hardware resources in an MPSoC and includes PBs and NoC. Each PB could be a processor core, a cluster of processor cores, a memory/buffer/cache, or a combination of them. MPSoCs can have homogeneous as well as heterogeneous PBs and use different NoCs. For the benchmark suite, we currently target regular NoC topologies, such as mesh, torus and fat tree. MPSoC architectures with three different regular-topology NoCs are illustrated in Fig. 3. The selection tries to cover the most popular NoC architectures first and will be expanded in the future. The rules of identifying the PBs with their coordinates (for mesh and torus) and serial numbers (for fat tree) are also displayed in Fig. 3. They will be applied in the traffic trace files to present the mapping results.

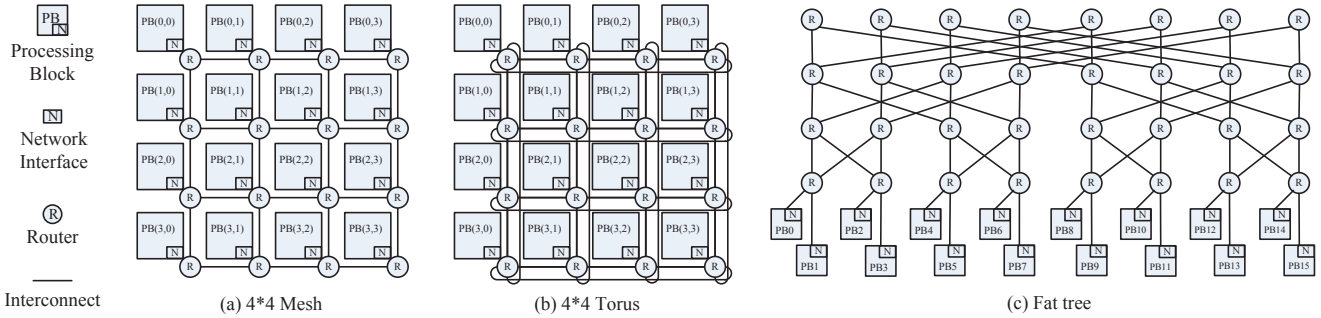

Fig. 3: 16-core MPSoCs with three different regular-topology NoCs.

We define an architecture model as a graph $G_p = (P, N)$, where P is a set of heterogeneous PBs, and N is an on-chip communication architecture organized in a NoC paradigm. A PB p has an attribute, acceleration factor a, for tasks executed on it, and the actual execution time of the task on this PB is determined both by its normalized execution time and the acceleration factor of the PB. Specifically, the running time is the multiplication of the two values. The NoC topologies and sizes included in the current version of the benchmark are listed in Table II. The numbers in the code column are used to indicate the topologies in the traffic files.

In the MCSL NoC benchmark suite v1.5, traffic patterns for RS-32_28_8_enc, RS-32_28_8_dec, ROBOT, FPPPP, FFT-1024_complex and SPARSE on three NoC topologies with all sizes listed in Table II

TABLE II: The NoC configurations included in the MCSL NoC benchmark suite.

Topology	Code	Size (number of processors)
Mesh	0	2x2, 2x4, 3x3, 4x4, 5x5, 4x8, 6x6, 7x7, 8x8, 9x9, 10x10, 11x11, 8x16, 12x12, 13x13, 14x14, 15x15, 16x16
Torus	1	2x2, 2x4, 3x3, 4x4, 5x5, 4x8, 6x6, 7x7, 8x8, 9x9, 10x10, 11x11, 8x16, 12x12, 13x13, 14x14, 15x15, 16x16
Fat tree	2	4, 8, 16, 32, 64, 128, 256

are provided. And for the application H264-720p_dec, traffic patterns on mesh and torus based NoCs with sizes from 2x2 to 8x8 are included in the Suite.

B. The recorded traffic pattern

A *RTP* is given by $T_r = \{V_r(p) \mid p \in P\}$, where $V_r(p)$ represents the recorded behaviors of the set of tasks scheduled and executed on PB p. The tasks $V_r = \{(s(v), t(v), IS(v), OS(v)) \mid v \in V\}$, where task v has execution time t, and the unique sequence number for scheduling it on p is $s(v)$. The execution condition of task v is given by its input set of information $IS(v) = \{(v_i(e), n_i(e), m_i(e)) \mid e \in E_i(v), v_i(e) \in V\}$, where $E_i(v) \subseteq E$ is the set of incoming edges of v, the data on every incoming edge $n_i(e)$ must be ready for v, and the data are obtained from the corresponding predecessor task $v_i(e)$ and read from the memory space started at $m_i(e)$. The result of the task execution is given by the output set of information $OS(v) = \{(v_o(e), p_o(e), m_o(e), d_o(e)) \mid e \in E_o(v), v_o(e) \in V, p_o(e) \in P\}$, where $E_o(v) \subseteq E$ is the set of outgoing edges of v, and that is to generate data of size $d_o(e)$ to edge $e \in E_o(v)$, the destination is the successor task $v_o(e)$ on PB $p_o(e)$, and the data are written to the same memory space $m_o(e)$, respectively. Each data is written to and read from the same virtual memory address, i.e., $m_i(e) = m_o(e)$. The data size $d_o(e)$ determines the number of packets that may be delivered through the network.

Since the absolute timing between task executions and communications is dependent on the simulation platform and can be affected by routing algorithm, router architecture, etc., it could not be used in new simulations with different settings. The RTP keeps data dependencies and temporal relations using relative timing instead of absolute timing. The relative timing will be generated according to the real dependencies in the applications and is guaranteed to be the correct semantics. This enhances the reusability of our traffic suite. NoC designs with the same topology and network size to the given traffic pattern can use it to evaluate the performance of their platform.

1) File format: The format of the trace file is shown in Table III. We call a data block generated by one execution of the source task on the edge "a message", and the message size is given in *words (32 bits)*. The task execution time is given by the number of *clock cycles*. Each edge is allocated a dedicated virtual memory space aligned on *words*. The virtual memory address for every message is given in the trace using *byte addressing*. In addition, 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. They are also provided in the trace files.

Note that there are multiple entries for the items starting with "*list of*" and the number of entries is equal to the number of iterations. In the benchmark, 20 iterations are recorded starting from an MPSoC executing the first tasks to the execution of the last tasks. We divide the total 20 iterations into 3 stages, namely Ramp-Up stage (the first five iterations), Stable stage (the 6th-15th iteration) and Ramp-Down

TABLE III: The format of the recorded traffic trace file.

Header block (5 line) [1]			
trace type			
topology	number of PBs	number of rows	number of columns
number of tasks	number of edges	number of iterations	
number of starting tasks	**list of** starting tasks		
number of finishing tasks	**list of** finishing tasks		
Task execution block (number of tasks lines, each of which is as follows) [2,3]			
task id	mapped PB id / coordinate		
list of schedule sequence numbers	**list of** recorded execution times		
Task communication block (number of edges lines, each of which is as follows) [4]			
edge id	source task id	destination task id	
list of memory addresses	**list of** recorded message sizes		

[1] The number of rows and columns are both 0 in fat tree
[2] 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. The rule for processor identification is illustrated in Fig. 3.
[3] Unit of task execution time: clock cycle
[4] Unit of message size: word or 32 bits

stage (the last five iterations).

2) Data structure: We define the data structure for the RTP. Using the included traffic loader, the traffic can be easily used in various simulators. As shown in Table IV, the trace is recorded according to a vector of PBs, each of which contains recorded information for the task invocations, memory addresses and packet transmissions on the PB. The schedule of the task instances on each PB is explicitly marked in each task and implicitly included by the sequence of the tasks stored in the task list vector.

C. The statistical traffic pattern

A *STP* is given by $T_s = \{V_s(p) \mid p \in P\}$, where $V_s(p)$ represents the statistical behaviors of the set of tasks scheduled and executed on PB p. The task set $V_s = \{(s(v), D_t(v), IS(v), OS(v)) \mid v \in V\}$, where the schedule of task v is given by a unique sequence number $s(v) \geq 0$, and the execution time of the task in different instances follow the Gaussian distribution with mean μ_t and standard deviation σ_t, $D_t(v) = (\mu_t(v), \sigma_t(v)), \mu_t(v) \geq 0, \sigma_t(v) \geq 0$. Suppose task v is executed on p for l times. Let the execution time of the j-th ($j \in [1, l]$) execution be t_j. The mean and standard deviation for the Gaussian distribution of v's execution can be computed as follows:

$$\mu_t(v) = \frac{1}{l}\sum_{j=1}^{l} t_j \tag{1}$$

$$\sigma_t^2(v) = \frac{1}{l}\sum_{j=1}^{l}(t_j - \mu_t(v))^2 \tag{2}$$

The execution condition of task v is given by its input set of information $IS(v) = \{(v_i(e), n_i(e), m_i(e)) \mid e \in E_i(v), v_i(e) \in V\}$, where $E_i(v) \subseteq E$ is the set of incoming edges of v, the data on every incoming

TABLE IV: The data structure of the recorded traffic trace.

```
class RecEdge {
  int                 id;                     // the id of the edge
  RecTask*            src_task;               // the source task
  RecTask*            dst_task;               // the destination task

  vector<int>         mem_addr_list;          // the list of memory addresses
  vector<double>      rec_msg_size_list;      // the list of recorded message sizes
};

class RecTask {
  int                 id;                     // the id of the task
  RecProc*            proc;                   // the PB the task is assigned
  vector<int>         schedule_list;          // the list of task schedule sequence numbers
  vector<int>         rec_time_list;          // the list of recorded execution times

  vector<RecEdge*>    incoming_edge_list;     // each entry is an incoming edge
  vector<RecEdge*>    outgoing_edge_list;     // each entry is an outgoing edge
};

class RecProc {
  int                 id;                     // the id of the PB
  int                 row_index;              // the row index in mesh/torus
  int                 col_index;              // the column index in mesh/torus
  vector<RecTask*>    task_list;              // the list of scheduled tasks
};

class RecNOCTraffic {
  int                 topology;               // the topology code
  int                 num_row;                // the number of rows in mesh/torus
  int                 num_col;                // the number of columns in mesh/torus
  int                 num_iter;               // the number of iterations the graph executes for
  vector<RecProc>     proc_list;              // the list of PBs
  vector<RecTask>     task_list;              // the list of tasks
  vector<RecEdge>     edge_list;              // the list of edges

  vector<RecTask*>    starting_task_list;     // the list of starting tasks
  vector<RecTask*>    finishing_task_list;    // the list of finishing tasks
};
```

edge $n_i(e)$ must be ready for v, and the data are obtained from the corresponding predecessor task $v_i(e)$ and read from the memory space started at $m_i(e)$. The result of the task execution is given by the output set of information $OS(v) = \{(v_o(e), p_o(e), m_o(e), D_d(e), D_i(e)) \mid e \in E_o(v), v_o(e) \in V, p_o(e) \in P\}$, where $E_o(v) \subseteq E$ is the set of outgoing edges of v, and that is to generate some amount of data to each edge $e \in E_o(v)$, the destination is the successor task $v_o(e)$ on PB $p_o(e)$, and the data are written to the memory space started at $m_o(e)$, respectively. We provide virtual memory address in the benchmark suite. Each data is written to and read from the same virtual memory address regardless of the memory architecture, i.e., $m_i(e) = m_o(e)$. The data size generated on an edge can be described by the Gaussian distribution $D_d(e) = (\mu_d(e), \sigma_d(e)), \mu_d(e) \geq 0, \sigma_d(e) \geq 0$. Suppose the data sizes generated on edge e in the l times of v's executions are $d_1 \ldots d_l$, respectively. The data size generated on each outgoing edge of task v can be calculated as follows:

$$\mu_d(e) = \frac{1}{l}\sum_{j=1}^{l} d_j \tag{3}$$

$$\sigma_d^2(e) = \frac{1}{l}\sum_{j=1}^{l}(d_j - \mu_d(e))^2 \tag{4}$$

With a given memory architecture, if the data generated by a task is going to be sent to a different PB via the on-chip communication network, they will be assembled into packets as they are generated

during the execution of the task. The benchmark suite is compatible with different packet definitions, and we assume a fixed packet size is used as an example in the STPs. Therefore, a number of packets are generated to assemble the data. The packet generation interval, which is the relative distance between two consecutive generated packets, follows the negative exponential distribution $D_i(e)$ with rate parameter $\lambda_i(e) \geq 0$ for v's executions. To make it realistic, we restrict the upper bound of the packet generation times to the finish time of the source task, i.e., the task will generate all the remaining output data at the end of the execution. Suppose the fixed packet size is z. The rate parameter for the negative exponential distribution of the packet generation intervals can be computed as follows:

$$\lambda_i(e) \quad = \quad \frac{\mu_d(e)}{z \cdot \mu_t(v)} \tag{5}$$

We use a fixed packet size of 8 flits and 32 bits per flit in the STPs included in the benchmark suite. The defined traffic pattern describes the statistical behaviors of the application running on the platform. We specify the deterministic task dependency relations in the generated traffic by the input and output sets $IS(v)$ and $OS(v)$, and include the task mapping and scheduling results as well as memory space allocated to the edges and distributed among the network. Three key components, the task execution times, the sizes of the data produced by the tasks and the relative time intervals that these data are assembled into packets, are described by statistical formulations. The generated pattern is useful for benchmarking NoCs with the same topology and network size, and other NoC metrics can be flexibly chosen.

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

Header block (5 line) [1]			
trace type			
topology	number of PBs	number of rows	number of columns
number of tasks	number of edges		
number of starting tasks	**list of** starting tasks		
number of finishing tasks	**list of** finishing tasks		
Task execution block (number of tasks lines, each of which is as follows) [2,3]			
task id	mapped PB id / coordinate	schedule sequence number	
μ_t	σ_t		
Task communication block (number of edges lines, each of which is as follows) [4]			
edge id	source task id	destination task id	
memory starting address	memory size		
μ_d	σ_d	λ_i	

[1] The number of rows and columns are both 0 in fat tree
[2] 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. The rule for processor identification is illustrated in Fig. 3.
[3] Unit of task execution time: clock cycle
[4] Unit of message size: word or 32 bits

1) File format: The format of the trace file is shown in Table V. The basic settings are similar with those in the RTP as presented in Section II-B1. A difference is that the virtual address of the memory space is given by a pair: the starting address and the memory size. *Byte addressing* is used, but each message occupies the space for an integer number of words.

TABLE VI: The data structure of the statistical traffic trace.

```
class StatEdge {
  int                 id;                     // the id of the edge
  StatTask*           src_task;               // the source task
  StatTask*           dst_task;               // the destination task

  int                 mem_start_addr;         // the starting address of the memory
  int                 mem_size;               // the size of the memory
  double              mu_msg_size;            // the mean of the message size
  double              sigma_msg_size;         // the sd of the message size
  double              lambda_pkt_interval;    // the rate parameter, the inverse of
                                              // the average packet generation interval
};

class StatTask {
  int                 id;                     // the id of the task
  StatProc*           proc;                   // the PB the task is assigned
  int                 schedule;               // the task schedule sequence number
  double              mu_time;                // the mean of the task execution time
  double              sigma_time;             // the sd of the task execution time

  vector<StatEdge*>   incoming_edge_list;     // each entry is an incoming edge
  vector<StatEdge*>   outgoing_edge_list;     // each entry is an outgoing edge
};

class StatProc {
  int                 id;                     // the id of the PB
  int                 row_index;              // the row index in mesh/torus
  int                 col_index;              // the column index in mesh/torus
  vector<StatTask*>   task_list;              // the list of scheduled tasks
};

class StatNOCTraffic {
  int                 topology;               // the topology code
  int                 num_row;                // the number of rows in mesh/torus
  int                 num_col;                // the number of columns in mesh/torus
  vector<StatProc>    proc_list;              // the list of PBs
  vector<StatTask>    task_list;              // the list of tasks
  vector<StatEdge>    edge_list;              // the list of edges

  vector<StatTask*>   starting_task_list;     // the list of starting tasks
  vector<StatTask*>   finishing_task_list;    // the list of finishing tasks
};
```

2) Data structure: We define the data structure for the statistical traffic pattern. Using the included traffic loader code, traffic can be easily used in various simulators. As shown in Table VI, the trace is recorded according to a vector of PBs, each of which contains statistical information for the task invocations, memory addresses and packet transmissions on the PB. The schedule of the tasks on each PB is explicitly marked in each task and implicitly included by the sequence of the tasks stored in the task list vector.

D. Memory information

The benchmark provides the memory addresses for the storage of the data generated on the edges in the form of virtual memory addresses. The use of virtual memory addresses allows users to experiment different memory organizations and physical memory allocations. The addresses are given in a single continuous virtual memory space, and each edge is assigned with a dedicated section of the space. The source task of the edge will write data to the given address and the destination task will read data from the same address. The sequence of address assignment is given PB by PB that starting from 0x0. Byte addressing is applied, i.e., each memory address identifies one particular byte. Note that the size of a data block is aligned on words (32 bits), so the memory space for each edge is always an integer number of words.

The STPs are intended for long simulation runs. In the STP, the size of the data block on each edge is described by a Gaussian distribution. The allocated memory space should be able to support the worst case in which a data block has the largest possible size. Therefore, the size of the data block in the worst case is used to calculate the memory size for the edge in the STP to guarantee correct executions. In the recorded traffic patterns, the exact size is given for each data block generated by one execution of a task. The memory address is given in this regard and a vector of memory addresses are available for the iterative generation of data blocks on the edge when the task executes for multiple instances.

E. Application performance measurement

The applications in the MCSL benchmark suite are run iteratively and different iterations are overlapped to achieve higher performance. The performance metric of "Application execution time per iteration" is the average time used for one iteration of an application's execution. We have listed the starting task set and finishing task set for each application in the traffic patterns. The starting time of an iteration is determined by the earliest starting time of the tasks in the starting task set, the finishing time of an iteration is determined by the latest finishing time of the tasks in the finishing task set, and the difference of these two metrics for one iteration is the execution time of that iteration of the application. The Application execution time per iteration is the average time over all the iterations of the application's execution, and can be obtained by monitoring the starting and finishing task sets.

REFERENCES

[1] www.ece.ust.hk/~eexu.

[2] W. Liu, J. Xu, X. Wu, Y. Ye, X. Wang, W. Zhang, M. Nikdast, and Z. Wang, "A noc traffic suite based on real applications," in *VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on*, july 2011, pp. 66 –71.

[3] http://www.systemc.org.