Network-on-Chip Traffic Patterns Based on Real MPSoC Applications

Jiang Xu

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY
Multiprocessor System-on-Chip

- Put all or most part of a complex system on a single chip
  - High performance, high reliability, low power consumption, compact
  - Cost-effective when volume is high
- Attractive platform for computing systems
  - The de facto choice for low-power, high-performance, high-volume, and mobile computing systems
Subsystems in MPSoC

- **Computation**: process data, implement major functions
- **Control**: coordinate all the subsystems
- **Memory**: temporally store data, instructions, and system status
- **Communication**: transfer data, instructions, and other information inside, into, and out of an MPSoC
- **Support**: maintain appropriate operating conditions, such as power supply, clock, temperature, *etc.*

- Can physically overlap
  - E.g. computation and control
- There are grey areas
  - E.g. communication interfaces

10/1/2013
Jiang Xu (HKUST)
Communication Subsystem

- **Performance**
  - Cooperation among functional units and chips
  - Communication latency

- **Power consumption**
  - Significant dynamic power used by communications
  - Considerable leakage of interconnect drivers and buffers

- **Cost and yield**
  - Metal layers for interconnects
  - Device layer for drivers and buffers

- **Challenges**
  - Longer communication delay
  - More on-chip communications
  - Higher power consumption
Communication Architecture

- **Ad hoc interconnects**
  - Dedicated point-to-point interconnections
  - Intuitive, but not cost-effective for complex systems

- **Bus**
  - Shared media communication architectures
  - Mature, but limited throughput and high power consumption

- **Network-on-Chip (NoC)**
  - Based on switching and routing techniques
  - High-throughput, scalable, and energy-efficient
  - But complex to design

- **Hybrid networks**
  - Very flexible
  - Complex to design and analyze

- **Future NoC directions**
  - Optical, nano-interconnects, wireless, etc.
NoC Traffic Patterns

- NoC traffic patterns are essential for
  - NoC architecture exploration
  - NoC performance evaluation

- Random/synthetic traffic patterns
  - Pro: simple to implement, pinpoint specific aspects of NoCs
  - Con: often cut off from computation and memory requirements, cannot show overall application performance and power, and application-specific issues

- Realistic traffic patterns are based on real applications
  - Pro: consider computation and memory requirements, show overall application performance and power, and application-specific issues
  - Con: difficult to get
MCSL Traffic Patterns

- MCSL stands for Multi-Constraint System-Level
  - Capture communication behaviors as well as their temporal and spatial dependencies
  - Include computation and memory requirements
- Available at [www.ece.ust.hk/~eexu/index_files/traffic.htm](http://www.ece.ust.hk/~eexu/index_files/traffic.htm)

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Tasks</th>
<th>Communication Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT-1024_complex</td>
<td>Fast Fourier transform with 1024 inputs of complex numbers</td>
<td>16,384</td>
<td>25,600</td>
</tr>
<tr>
<td>H264-1080p_dec</td>
<td>H.264 video decoder with a resolution of 1080p</td>
<td>5,191</td>
<td>7,781</td>
</tr>
<tr>
<td>H264-720p_dec</td>
<td>H.264 video decoder with a resolution of 720p</td>
<td>2,311</td>
<td>3,461</td>
</tr>
<tr>
<td>FPPPP</td>
<td>SPEC95 Fpppp is a chemical program performing multi-electron integral derivatives</td>
<td>334</td>
<td>1145</td>
</tr>
<tr>
<td>RS-32_28_8_dec</td>
<td>Reed-Solomon code decoder with codeword format RS(32,28,8)</td>
<td>278</td>
<td>390</td>
</tr>
<tr>
<td>RS-32_28_8_enc</td>
<td>Reed-Solomon code encoder with codeword format RS(32,28,8)</td>
<td>248</td>
<td>328</td>
</tr>
<tr>
<td>SPARSE</td>
<td>Random sparse matrix solver for electronic circuit simulations</td>
<td>96</td>
<td>67</td>
</tr>
<tr>
<td>ROBOT</td>
<td>Newton-Euler dynamic control calculation for the 6-degrees-of-freedom Stanford manipulator</td>
<td>88</td>
<td>131</td>
</tr>
</tbody>
</table>

10/1/2013
Jiang Xu (HKUST)
MCSL Traffic Patterns

- Support different topologies and network sizes
  - Currently cover mesh, torus, and fat tree
- Two types of traffic patterns
  - Recorded traffic pattern (RTP) for detailed analysis
  - Statistical traffic pattern (STP) for average, worst, and best case studies
- RTP includes three stages
  - Ramp-up, stable, and ramp-down
- More applications will be released soon
Generation Flow

- Based on HW/SW codesign methodology
- Optimized to maximize overall application performance
  - Memory space allocation
  - Application mapping and scheduling
- Modular design for future expansions and customizations
The computation and communication requirements are captured by the application model.

Application model is a task communication graph (TCG)

A directed graph $G_t = (V, E)$, where $V$ is the set of computation tasks, and $E$ is the set of communication links between tasks. A task $v$ has an execution time $t$, which is a function of task executions. 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 $v_d$. $w$ is a function of communication transactions.
Genetic algorithm is used to reduce memory size while maintain maximum application performance

**Require:** application model $G_t(V, E)$

1. define $pop\_size$, $gen\_num$, $best\_sol$
2. $pop\_parents = \text{Initialization}(pop\_size)$
3. while $gen\_num$ is not reached do
4. $pop\_offspring = pop\_parents$
5. Crossover($pop\_offspring$)
6. Mutation($pop\_offspring$)
7. $pop\_parents = \text{Selection}(pop\_offspring, pop\_size)$
8. $best\_sol = \text{UpdateBestSolution}(pop\_parents)$
9. end while
10. return the best memory space allocation $best\_sol$

An application memory space is created as the result

Different physical memory space configurations can be used
Architecture Model

- Capture hardware resources
- \( G_p = (P, N) \), where \( P \) is a set of heterogeneous processing blocks (PB), and \( N \) is the topology of an on-chip communication architecture
  - PB can be processors, accelerators, memories, and/or memory interfaces
- Heterogeneous processors/accelerators
  - Using acceleration factor
- NoC topologies and sizes
  - Mesh, torus, fat-tree are covered from 4-core to 256-core in current version
  - Topology coverage can be expanded

PB can be processors, accelerators, memories, and/or memory interfaces.
Application Mapping and Scheduling

Require: application model $G_t(V, E)$, architecture model $G_p(P, N)$

1: $time = 0$
2: while there is unscheduled task in $G_t$ do
3:     readyQueue = UpdateReadyQueue($G_t$)
4:     for each task $v$ in readyQueue do
5:         $minWeight = \infty$
6:         for each PB $p \in P$ do
7:             if $w(v, p) < minWeight$ then
8:                 $selectedProc = p$
9:                 $minWeight = w(v, p)$
10:            end if
11:        end for
12:        $m(v) = selectedProc$
13:        $s(v, selectedProc) = GetOrder(selectedProc)$
14:        $procAvailTimes = GetEarliestAvailableTimes(P)$
15:    end for
16:    $time = TimeAdvance(procAvailTimes)$
17: end while
18: return mapping $M$, schedules $S$

Cost function:

$$w(v, p) = c_1 t(v, p) + q c_2 n(v, p)$$

in which

$$t(v, p) = f(p) + t(v) \times a(p),$$

$$n(v, p) = \sum_{(v, u) \in E} k(v, u) \times l(p, m(u))$$

| $w(v, p)$ | The weight of mapping task $v$ to PB $p$ |
| $t(v, p)$ | The required time for task $v$ to finish on PB $p$ |
| $f(p)$ | The time for executing previously assigned tasks on PB $p$ |
| $n(v, p)$ | The total amount of network transmission generated by task $v$ when it is assigned to PB $p$ |
| $k(v, u)$ | The number of packets that task $v$ generates to edge $(v, u)$ |
| $m(v)$ | The mapping of task $v$ |
| $l(p, q)$ | The distance between PB $p$ and $q$ (predicted according to routing policies on different NoC topologies) |
Recorded Traffic Pattern

- Generated through cycle-accurate simulations
  - All task execution and task communication events are recorded
  - Temporal and spatial communication dependencies (not their exact timing) are kept
  - Distributed shared memory is used

\[ T_r = \{ V_r(p) \mid p \in P \} \]

The traffic is given by the set of tasks allocated on the PBs

\[ V_r(p) = \{ (s(v), t(v), IS(v), OS(v)) \mid v \in V \} \]

The task schedule with dependencies and exact execution times

\[ IS(v) = \{ (v_i(e), n_i(e), m_i(e)) \mid e \in E_i(v) \} \]
\[ OS(v) = \{ (v_o(e), p_o(e), m_o(e), d_o(e)) \mid e \in E_o(v) \} \]

The input and output sets with memory space allocation
RTP Data Structure

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
};
Use statistical distributions to capture the uncertainty in task execution time, data size, and packet interval

\[ T_s = \{ V_s(p) \mid p \in P \} \]

The traffic is given by the set of tasks allocated on the PBs

\[ V_s(p) = \{ (s(v), D_t(v), IS(v), OS(v)) \mid v \in V \} \]

The tasks are scheduled, with dependencies and execution times

\[ IS(v) = \{ (v_i(e), n_i(e), m_i(e)) \mid e \in E_i(v) \} \]
\[ OS(v) = \{ (v_o(e), p_o(e), m_o(e), D_d(e), D_i(e)) \mid e \in E_o(v) \} \]

The input and output sets, with memory space allocation

\[ D_t(v) = (\mu_t(v), \sigma_t(v)) \]
Gaussian distribution of task execution time

\[ D_d(e) = (\mu_d(e), \sigma_d(e)) \]
Gaussian distribution of data size

\[ D_i(e) = \lambda_i(e) \]
Negative exponential distribution of packet generation interval
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
};
RTP vs. STP

• Tradeoffs between accuracy and coverage
• RTP is accurate but low coverage
  • Exact record of every task instance and communication transaction
  • Include ramp-up, stable, and ramp-down stages
  • Useful for detail analysis, such as debugging
• STP is less accurate but cover more cases
  • Using statistical models to cover wide possibilities
  • Useful for average and worst/best case studies which require extensive simulation runs
Communications Among Tasks

FFT-1024_complex

Fpppp

Robot

RS-32_28_8_dec

RS-32_28_8_enc

Sparse
Communications Among PBs on 8x8 Mesh-based NoC

FFT-1024_complex

Fpppp

Robot

RS-32_28_8_dec

RS-32_28_8_enc

Sparse
RTP vs. STP

- Performance results showed by RTP and STP are consistent

<table>
<thead>
<tr>
<th></th>
<th>Average difference of RTP vs. STP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network throughput</td>
<td>1.3%</td>
</tr>
<tr>
<td>Packet delay</td>
<td>6.0%</td>
</tr>
<tr>
<td>Application throughput</td>
<td>2.7%</td>
</tr>
<tr>
<td>Application execution time</td>
<td>0.9%</td>
</tr>
</tbody>
</table>
Comparing to Random Traffics

- Significantly different NoC performance results reported by random traffic patterns
  - E.g. uniform traffic patterns report $34X$ smaller packet delay on average
Acknowledgement

- Zhe Wang, Weichen Liu, Xiaowen Wu, Bin Li, Ravi Iyer, Ramesh Illikkal, Wei Zhang, Yaoyao Ye, Zhehui Wang, Mahdi Nikdast

intel

University Grants Committee
Reference


Thanks!