Attackboard: A Novel Dependency-Aware Traffic Generator for Exploring NoC Design Space

Yoshi Shih-Chieh Huang, Yu-Chi Chang, Tsung-Chan Tsai,
Yuan-Ying Chang, and Chung-Ta King
Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan
{yoshi, yuchi, tctsai, elmo, king}@cs.nthu.edu.tw

ABSTRACT

Network-on-chip (NoC) is very important for many applications, such as many-core architectures and application-specific usages. For exploring the design space, several approaches have been proposed with different considerations. In this paper, inspired by bloom filters, we propose *Attackboard*, a novel design for exploring the design space of NoC, which satisfies accuracy, space efficiency, and simplicity. To justify the usage of Attackboard, a parallel object detection program is used as the benchmark program to evaluate the performance of a specific NoC. By comparing the results with an execution-based simulator, it shows that Attackboard simultaneously achieves the requirements of fast speed, simplicity, and accuracy.

Categories and Subject Descriptors

B.4 [**Input/output and data communications**]: Processors

General Terms

Performance, Design

Keywords

Network-on-chip, Many-core, Dependency, Traffic generator, Table-driven

1. INTRODUCTION

Network-on-chip (NoC) has become the de facto of the substrate of many-core architecture due to its simplicity and scalability. Exploring the design space of NoC is therefore increasingly important. To study a novel NoC architecture, architects usually exercise it with realistic workloads and measure quantitatively how close the design objective is approached. Consequently, a model of the interested architecture needs to be established and evaluated first, even when the detailed implementations are not available yet.

Early-stage models are essential for architects working on novel architectures, in order to evaluate the proposed architecture even before the detailed RTL or circuit design is available. Existing early stage models for NoC architectures come in different flavors, and cover a wide range of accuracy, simulation speed, and flexibility. One of the most confident solution to early stage models are full-system simulators, which have been success in flexibility as it takes moderate efforts to develop and configure at early design stage. For the mature full-system simulators including NoC, a wide range of topologies, routing algorithms, and router architectures can be modeled by changing the simulation parameters, such as Simics with Garnet, GEMS, M5, SESC, and etc [3, 10, 5, 8]. Full-system simulators have been successfully deployed to study various architectures, due to their high flexibility and accessibility. However, these simulators have relatively lower simulation performance, and does not scale with the progress of modern many-core architectures.

In contrast to simulate the full system, one another way to address the simulation complexity is through trace-driven simulator, which takes the trace log generated in execution as input. A second simulation run, usually with cycle-accuracy, is driven by the previously generated trace. Based on similar methodology, a number of trace-driven simulators for NoCs have been proposed, such as BookSim [1] and [9]. Comparing to full-system simulation, trace-driven simulation is simple and fast. However, the trace-driven simulation is open-loop, which does not consider the backpressure from NoC to the processing elements. Therefore its accuracy is low and may not be tolerable for all studies.

As a result, improving the accuracy of trace-driven simulation has been addressed in state-of-the-art research. In [11], dependencies among injected packets are inferred and embedded into the raw trace logs. Embedding dependencies into traces absolutely brings more confident results of evaluation. However, once dependencies are embedded into original trace logs, two following problems arise. First, it makes the trace logs much complicated and require more storage space, ranging from megabytes to gigabytes according to the granularity. Second, each entry in trace-log becomes correlated and therefore need to be processed while evaluating systems with dependency-aware traces.

In this paper, we propose *Attackboard*, a new methodology to help explore the design space of network-on-chip. Attackboard strikes a balance between speed and accuracy, while keeping the simplicity as trace-driven simulation. The essence of Attackboard is multiple bloom filters. By taking advantages of the property of denial for sure, permitting for high confidence, Attackboard[1] can rebuild the application behavior with a simple pattern-oriented table.

Since each attackboard is a dependency-driven table rather than a time-driven table, redundant dependency patterns only take one line in an attackboard. This property helps reduce the size of table since a program is usually intrinsically repeating program because of loops. By keeping the causality-domain instead of time-domain, the execution can be logged with the pattern-oriented method.

Our evaluations show the accuracy and space overhead of Attackboard by comparing with an execution-based simulator. To identify the contribution of this paper, we listed them as follows:

- A novel design with multiple bloom filters for NoC evaluation is proposed.
- Experiments for comparing Attackboard with an execution-based simulator are evaluated.
- Analysis for the practical implementation is discussed.

The rest of paper is organized as follows. Related works are given in Section 5. Next, Section 2 provides a formal definition of Attackboard. In Section 3, we firstly start with the overview of Attackboard, and followed by the detailed operations and design options. In Section 4, we evaluate Attackboard by comparing with execution-based simulation. Finally, conclusion is drawn in Section 6.

2. PROBLEM FORMULATION

Assume that there are N routers in an NoC. To each router k, it contains its own *attackboard*, denoted as ab_k, which is a two-dimension table, and in which each row stands for a unique dependency pattern, and each column represents the necessity of *must having data from this router*, i.e., a necessary predecessor. For example, if there is a **1** in the cell of row i and column j, it means that for satisfying pattern i, router j must have already injected data to router k. In contrast, if the number in the cell is **0**, it means that this pattern is not necessary to be after the injection of router j to k.

Each router k contains a *Current Status* (CS_k) with length N in terms of bit. By using CS_k as the index to match an entry in ab_k, once a pattern is satisfied, the corresponding injection is generated. This problem can be modeled as a *Multiple Bloom Filters* problem. Each row represents a bloom filter, and CS_k is an entry which is trying to pass one or more bloom filters.

3. SYSTEM DESIGN

3.1 Overview

There are mainly two stages in Attackboard. In the first stage, as Figure 1(a) shows, the dependencies of the injections from each source node are firstly discovered, and then represented with a table structure. The basic idea of this stage is to *periodically capture the relationships among injections.* Once the stage one is done, the characteristics of the benchmarks are now represented by the attackboards.

[1]In the following context, *Attackboard* with initial capital stands for the whole design, and *attackboard* with the lower case stands for the table structure.

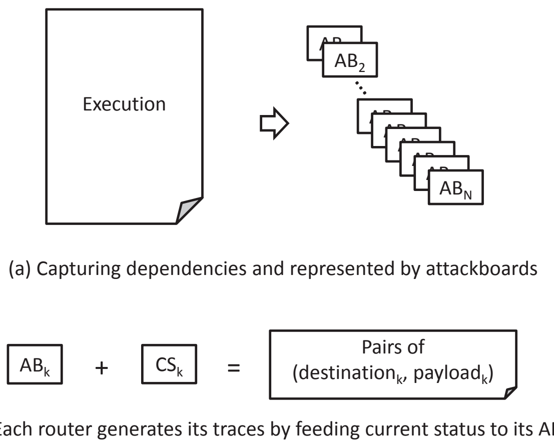

Execution
AB$_2$
AB$_N$
(a) Capturing dependencies and represented by attackboards
AB$_k$
+
CS$_k$
=
Pairs of (destination$_k$, payload$_k$)
(b) Each router generates its traces by feeding current status to its AB

Figure 1: The system flow of Attackboard. (a) Execution logs are broken to attackboards and distributed to each router. (b) A router generates traffic by feeding its current status to its attackboard.

In the second stage, as shown in Figure 1(b), while evaluating an NoC configuration, each router only needs to look up its own attackboard to generate traffic. Similarly, the basic idea of this stage is to *periodically generate the traffic according to the indications of attackboards.*

3.2 Stage 1 - Creation of Attackboard

For each send event, we observe the received traffic for an interval. Suppose that a send event n is observed at cycle x, and an interval size I is selected as our observing window size, i.e., any traffic occurs between cycle $I - x$ to x would be granted as a dependent receive event to the send event n. As the representation between these dependent events and send event n, a corresponding row is inserted into the attackboard. For each dependency event, the relationship would be marked as **1** in the corresponding cell in the table, indicating if the source core of a dependent receive event had communication with the core of the observed send event. For example, in the scenario shown in Figure 2, the receive event $n - 1$ and $n - 2$ would be granted as the dependent events to send event n. And then a dependency pattern could be built as $< 1, 0, 1 >$, which implies that core 0 and core 2 have communicated with core 3 before the send event n occurs.

The attackboard is able to semantically reveal the program behavior, thus serve as a good evaluation tool to exploit the performance of NoCs. As previous works [11, 7], it is important to collect *correct* dependency information for attackboard so as to discover the semantic meaning of programs. The selection of dependency-extraction interval size I is thus critical. The size of I decides how many dependent (or independent) events of a send event could be seen, and therefore strongly correlates to the accuracy of the recorded dependency information.

Take the function `MPI_Allreduce` as an example. `MPI_Allreduce` performs a reduce operation after core 0 collects data from all other cores and then broadcast the results back to all cores. Since the broadcast can only happen after core 0 receives all data from other cores, the corresponding attackboard of this operation should be like $< 0, 1, 1, 1 >$ with send events $P_1(4), P_2(4), P_3(4)$. This attackboard indicates that only after receiving from core 1, 2, 3, can core 0 sends

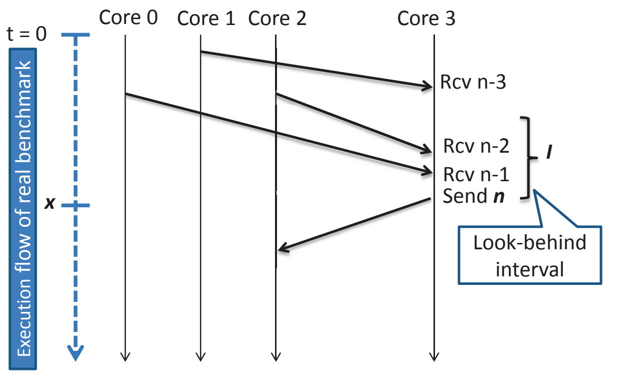


Figure 2: Recognizing the dependent receives of send n.

data to core 1, 2, 3, each with data quantity 4, which is exactly how `MPI_Allreduce` works. By using this small and scalable attackboard, the semantic meaning of program behaviors can be well presented. In Section 4.1, we have an experiment to discuss the choices of dependency extracting interval.

3.3 Stage 2 - Usage of Attackboard

3.3.1 Initialization

The dependency status of each router k is initially set as all **0**s. Note that there is always an attackboard with dependency pattern as all **0**s, since the very first send event of the program depends on no other send events but the startup of the execution. Therefore, the attackboard need nothing to be propelled but the startup of the simulation. After the first send event generated by the attackboard, the simulation would keep generating dependency-aware traffic to cause the following injections as a chain reaction.

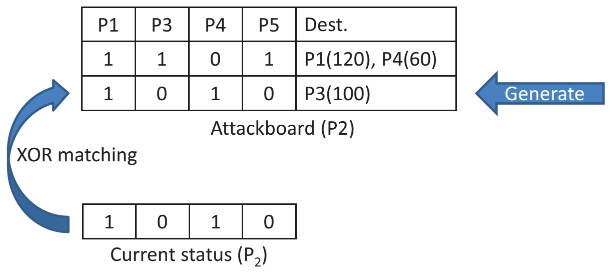
3.3.2 Entry Matching

As Figure 3 shows, each router k keeps its *Current Status* (CS_k) which is caused by others. CS_k is a bitwise vector with length $|N|-1$, where N is the number of processors. For a bit in CS_k at column y, **0** means that currently processor k has not received any packet from processor y. In contrast, **1** represents that processor k has received data from processor y.

In the end of each interval I', CS_k is used as an index to match its own $attackboard_k$ (ab_k) to find the matches. As the matches are found, the recorded send events of this dependency pattern entry will be generated. Note that the generated traffic of an entry will be evenly distributed to the incoming interval for avoiding all the send events injecting at the starting point of the next interval. This is for simplifying the simulation without recording the computation time before an injection.

3.4 Table Minimization

Since the design of Attackboard are based on bitwise bloom filters, the number of entries are proportional to the number of processors, i.e., $2^{(N-1)}$, where N is the number of processors. For future many-core architecture, the number of processors is expected to be thousands of cores in a chip. Considering such a situation, we propose two methods to reduce the size of attackboards.



P1	P3	P4	P5	Dest.
1	1	0	1	P1(120), P4(60)
1	0	1	0	P3(100)

1	0	1	0

Figure 3: A sample of an attackboard and the procedure of entry matching and traffic generating

3.4.1 Merging duplicated entries

During our gathering the dependency information, those send events with the same dependency pattern would be compressed into the same dependency pattern category. Upcoming send events would be appended to the end of the dependency pattern category, except for those send events with same quantity and destination. The send events of same quantity and destination would be granted as the repetition of one send events, and thus only one send event would be recorded. However, the payloads of two or more send events may be different. Denote a set S as set of the send events which have the same predecessors and injecting destination, and denote the corresponding payloads as $payload_i, i \in S$. We proposed three solutions: First, by averaging the payloads and only record the averaged value as the constant payload. That is, $\frac{\sum_{i \in S} payload_i}{|S|}$. Second, record the payloads as a sequence, i.e., in the cell of destination, keep the sequence as $P_i(payload_1, payload_2, ..., payload_{|S|})$. While generating traffic, the sequence acts as a circular queue. Third, based on the second solution, instead of using the round-robin strategy to select, different payloads are tagged with probability values according to the number of occurrences, and the selection is based on the probability.

3.4.2 Merging similar entries

For further minimizing each attackboard, similar entries can be merged into one. For identifying those entries which have similar patterns, *eXclusive OR* (XOR) operation can calculate the Hamming distance of two entries. For those entries which have closest Hamming distance are considered to be merged first. Once two entries are considered as high similarity, denoted as E_i and E_j, the two entries are removed and replaced by a new entry $E_k = E_i \cdot E_j$. For example, if $E_i =< 1,0,1,1 >$ and $E_j =< 1,1,0,1 >$, then it can be replaced with $E_k =< 1,0,0,1 >$. This operation relaxes the condition of the necessity of predecessors, but the new entry still keeps the coexisting predecessors.

4. EVALUATION

The evaluation of NoC architectures usually involves performance of different NoCs during the executions of real programs. If the average network delay of NoC A is more than that of NoC B, we assert that the NoC B is more suitable than NoC A for target programs. In the context of this evaluation standard, we would evaluate how close the average network delay is by comparing the results of attackboard with real benchmark executions on an execution-based simulator, since it is a representative of the semantic meaning

Table 1: Default simulation setup

Simulation Platform	
Native processor element	Tilera TILE64 [4]
Native processor frequency	700 MHZ
Simulated topology	4×4 mesh network
Routing algorithm	Dimension-order
Bandwidth	1 flit/cycle per port

Core ID	Attackboard dependency pattern entry	send event
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	$P_1(4)$
	0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0	$P_1(4)$
1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	$P_2(4)$
	0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0	$P_0(4)$
15	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	$P_{14}(4)$

(a) Attackboards of odd-even sort (16 cores)

Core ID	Attackboard dependency pattern entry	send event
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	$P_1(2, 4, 30)$
1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	$P_2(2)$
	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	$P_2(4, 30)$, $P_0(30)$, $P_3(30)$
15	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	$P_7(30)$

(b) Attackboards of object detection (16 cores)

Figure 4: Attackboards of odd-even sort and object detection.

of real programs. The accuracy is considered higher if the simulation results of attackboard is closer to the results of real benchmark executions. Besides accuracy, the size of attackboard is also evaluated compared with communication traces to meet the needs of easy distribution.

In the following paragraph, we evaluate Attackboard with execution-based simulation in terms of accuracy, size, and how representative the attackboard is for semantic meaning of real programs. The parameters of the simulated environment are shown in Table 1. The utilization of Attackboard involves two stages. At the first stage, we run real benchmarks to generate Attackboard by retrieving dependency information, and examine if attackboard could successfully portray the semantic meaning of the behaviors of real programs. Afterwards, at the second stage, we would run the simulation to generate dependency-aware traffic with Attackboard to examine the accuracy. Finally, we would compare the size of attackboard with trace.

4.1 Dependencies Extracting

We use an execution-based simulator to execute the instrumented programs and capture the dependencies for a micro-benchmark from Intel MPI Benchmark (IMB) suites [2] and a parallel object detection program. The instrumentations are done by hand and therefore guaranteeing the true-dependencies. Other strategies to automatically capture the dependencies are discussed in recent works [7, 11, 13].

Note that the dependent events could be repetitive among send events. That is, if an upcoming send event $n+1$ occurs right after send event n, the observing window of these two send events might overlap, which results in the same dependency pattern, the reason is that these two send events are just dependent on the same receive events simultaneously (such as the MPI group communication as `MPI_allgather`, `MPI_alltoall`, etc.) in real benchmarks. Furthermore, as a dependency pattern driven traffic generation, Attackboard would generate two send events once the receiving dependency pattern is satisfied, which is compatible with the semantic meaning of programs.

Figure 4 shows the derived attackboard of odd-even sort and object detection. In the odd-even sort part, the attackboards of core number 0, 1, and 15 are shown. The first entry of attackboards shows that core 0 would send data (with quantity 4) to core 1 without depending on any send events, and the core 1 would send data to core 2 in the same situation. Meanwhile, the core 15 would only generate data to core 14 after receiving data from core 14. This shows the odd phase of the odd-even sort, while the even phase is shown in the second dependency entry of core1, and the first entry of core 15. In the even phase, the core 1 would pass data back to core 0, which describes the "swap" operation in the odd-even sort. And the second entry of core 0 conveys that if the even phase ends, there should be another upcoming odd phase. The attackboards of the unmentioned cores also obey this communication behavior as discussed.

In the other hand, the object detection part shows that, initially core 0 would pass data to core 1, while core 1 passes data to core 2. And after the receiving data from core 0, core 1 would send (handled) data back to core 0 and also pass data to core 2 and core 3. Meanwhile, the core 15 would send data to core 7 after receiving data from core 14. This behavior indicates group communication. The object detection algorithm here uses data parallelism to detect if object exists in a frame. To achieve data parallelism, some data must be exchanged between cores to learn acknowledge of other part of data. The attackboard also presents the behavior of data parallelism, which is compatible of the semantic meaning of the programs of object detection.

To conclude, it is feasible to use the derived attackboards to reveal the semantic meaning of the corresponding programs.

4.2 Traffic Generating

As discussed in section 4.1, the retrieval of Attackboard involves the dependency extraction. The dependency extraction methodology has been proposed, with an interval size I introduced. However, it may be questionable that if a fixed dependency-extraction interval size could really embrace all essential dependency information for different send events occurred in execution. Thus, during the generation of dependency-aware traffic, instead of matching the recorded dependency patterns of attackboard, we grant these dependency patterns as bloom filters. By using the characteristic of bloom filter – *denial for sure, permitting for high confidence* – we could generate the traffic only with essential dependent send events, and still provide chances for those unrecorded send events (which is also likely to be a predecessor, but somehow lost during creation of attackboard).

With the Attackboard composed of bloom filters, we could then generate confident dependency-aware traffic. The traffic is generated periodically. Every time the setup simulation interval I' expires, an attackboard would be thoroughly examined. During the examination, we compare the current status and the dependency entries of attackboard. Only the **1**s in the dependency pattern entries matter. That is, if we encounter a **0** in the dependency patterns, we simply treat it as a "don't care". For example, with current status $< 0, 1, 1 >$ comparing a dependency pattern $< 0, 1, 0 >$, the generation of traffic is permitted. Contrarily, if a dependency pattern $< 1, 1, 1 >$ is encountered, the traffic genera-

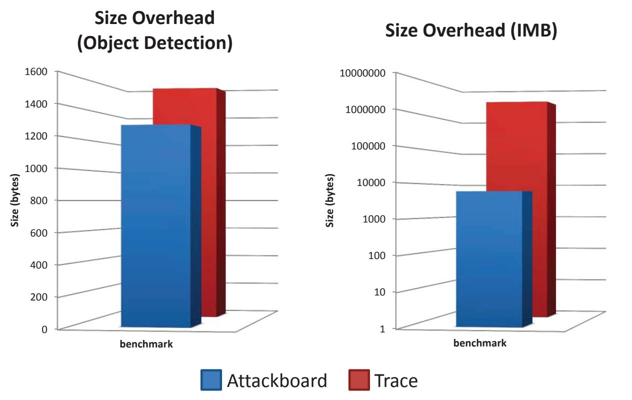


Figure 5: The space overhead of attackboards compared to trace files.

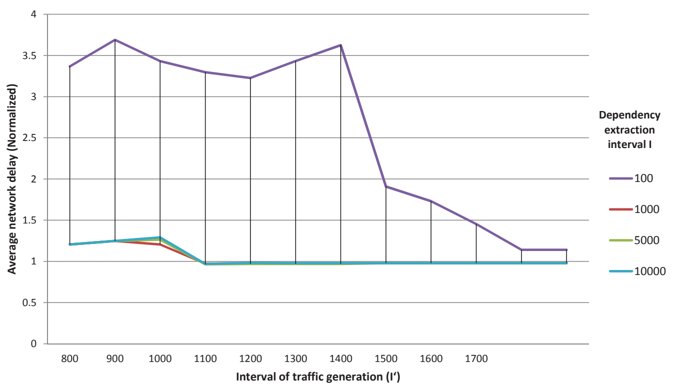


Figure 6: The accuracy of object detection under different I and I'.

tion is not permitted.

Once a suitable dependency pattern is found, corresponding traffic would be generated. If no suitable dependency pattern is found, then no traffic would be generated. Since the generated traffic would obey the form of dependency information revealed from real benchmarks, the generated traffic behaves like the real benchmarks.

As described, the attackboard would periodically generate dependency-aware traffic with simulation interval I'. Since the traffic injection rate is a critical factor to evaluate NoCs, the selection of I' (i.e., how often Attackboard generates traffic) plays an important role of attackbard simulation. An appropriate I' could attackboard behave as real programs. The selection of I' heavily depends on the NoC architectures, so a tuned I' for a simulating NoC architecture is always needed in the attackboard simulation. In the following experiments, we would show that with the easily tuned I', the attackboard could be a representative of real benchmark. After the tuned I' is found, the average network delay could then be generated with attackboard simulations.

4.3 Space Overhead

As shown in Figure 5, the space requirement of Attackboard is quite small. In the size comparison for object detection, the size of attackboard is comparatively small than the size of communication traces. Note that the improvement is not big because of the communication events are relatively lesser and execution time is shorter compared to other real programs. A great improvement can be observed in the size comparison for IMB, it is because the long execution time and intensive communications, thus the size of communication traces is very large, and it would grow even larger if the execution time is set longer. Compared with the large size of IMB traces, attackboard is much smaller. Attackboard is scalable, since Attackboard is a timeless table, so the size of Attackboard would not grow linearly with time, but depends on *how many distinct dependency patterns exist in the programs.* Further, the dependency pattern that Attackboard records are in essence the semantic meaning of program, the repetition of behaviors would be folded naturally with Attackboard, which makes Attackboard more scalable to use.

4.4 Case Study

The accuracy of Attackboard is evaluated by the comparison of average network delay. We compared Attackboard with the average network delay of the execution of parallel object detection and IMB Broadcast. The accuracy is presented as the normalized average network delay with that of execution of real benchmarks. Also, multiple I and I' are tested to derive the best result.

Figure 6 and Figure 7 show the statistics of normalized average network delay Attackboard achieved with different I and I' (in terms of cycle). The x-axis represents simulation interval length I' and the y-axis represents the normalized results. Different curves in the figure represent an attackboard with different simulation interval length I. In the following paragraphs, we discuss different cases, respectively.

4.4.1 *Parallel Object Detection*

As we can see in Figure 6, there are two groups of the curves. The first group is the curve with I set as 100. The curve is initially very high, and falls down slowly. After I' is set larger than 1600, it would arrive a steady value while still remain a comparatively high value. The reason could be found obviously in the content of the attackboards. There is only one dependency entry – all zeros – of the attackboard with I set as 100. This shows that the length of dependency-extraction interval is too small, so that the attackboard could not capture any useful dependency information for send events. Therefore, during the simulation of Attackboard, the send events would inject every interval, which causes severe congestion.

The second group is the curve with I set as 100, 1000, 5000, and 10000. The curves of second group would startup high and then flatten out afterwards. It is because when the length of I' is too small, congestion would occur since the injection rate is too high. In contrast, after I' is fairly set, the injection would then balance with routers' receiving, thus generate a steady average network delay value, which matches the result of the real program. In this group, the results almost fit the ideal case.

4.4.2 *Intel IMB Broadcast*

Different from the simulation behaviors of object detection, the simulation results of attackboard of IMB broadcast is very centralized. As Figure 7 shows, the accuracy (normalized average network delay) of the simulation results ranges from 0.8 to 1.1, showing the high feasibility of the

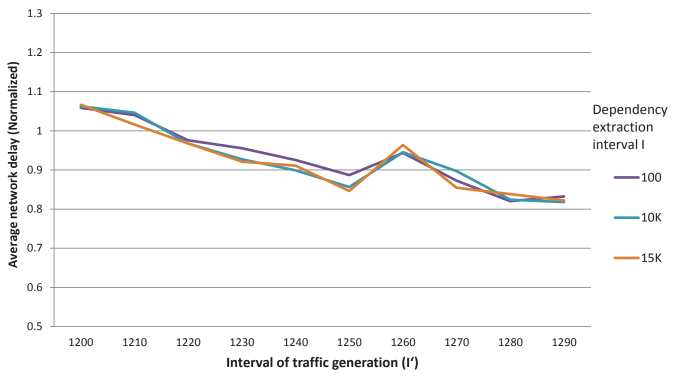


Figure 7: The accuracy of IMB under different I and I'.

dependency-aware traffic generation by Attackboard. However, the selection of I does not really matter in this case. As discussed in section 4.4.1, the interval size has a strong correlation with the dependency information. If the smaller the dependency-extraction interval is chosen, the less dependency information. But in this case study, attackboards of various interval sizes seem to behave in the same way. It is thanks to the power of bloom filters that suffice the distortion of dependency information. Since Attackboard only checks the confident recorded predecessor (the **1**s in the dependency entry of Attackboard), and considers others as "don't cares", the distorted result is automatically corrected by the natural operations of simulator. Because Attackboard will not reject those "high-confident" traffic generation, the behavior of broadcast is appropriately replayed, thus achieve high accuracy.

In summary, since Attackboard could achieve accurate results with easily tuned parameter, it becomes a good substitute for evaluating NoCs. Using Attackboard as the simulation tool thus much alleviates the evaluation of NoC for real programs. The easy implementation, small size, and the capability of exploiting the semantic meaning of real programs even make Attackboard a better tool to evaluate NoCs.

5. RELATED WORKS

Research on the on-chip interconnection network greatly uses a trace-driven simulator [1, 6, 13]. Unfortunately, the accuracy of trace-driven simulation is not convincing, because in most cases, the dependencies of the transmission are broken, i.e., each entry in the trace log are not correlated. As a result, improving the accuracy of trace-driven simulation has been important in state-of-the-art research. In [11], dependencies among injected packets are inferred and embedded into the original trace logs. Similar idea can also be found in [7]. Once dependencies are embedded, it absolutely improves the accuracy of trace-driven simulation; however, it makes the trace logs much complicated and require more storage space, ranging from megabytes to gigabytes.

Many other methodologies to help explore design space of NoC are proposed, including full-system simulation [3, 5], mathematical models, FPGA-based acceleration [12], etc. Each of them has their own pros and cons. Among these existing works, Attackboard strikes the balance between full-system simulation and trace-driven simulation.

6. CONCLUSION AND FUTURE WORKS

In this paper, we propose Attackboard, a new methodology to help explore the design space of network-on-chip. Attackboard takes advantages of repetitive behaviors and dependencies among injections. We use IMB and a parallel object detection program to evaluate the performance of Attackboard. The results show that Attackboard has high accuracy as programs directly run on the execution-based simulator. On the other hand, we also compare Attackboard with trace-driven simulation in terms of space requirements. The results show that the space overhead is much smaller while keeping the accuracy.

The future works are twofold. First, the intervals for extracting and generating rely on empirical rule to find. We are developing an automatic process to find the suitable intervals. Second, Attackboard currently is evaluated with message passing programs. In our ongoing work, the traffic of shared-memory programs will be included.

7. REFERENCES

[1] BookSim 2.0.

[2] Intel MPI Benchmarks.

[3] N. Agarwal, T. Krishna, L.-S. Peh, and N. Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software, 2009. ISPASS 2009.*, pages 33 –42, April 2009.

[4] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - processor: A 64-core soc with mesh interconnect. In *Proceedings of Internation Conference on Solid-State Circuits, 2008. ISSCC 2008.*, pages 88 –598, feb. 2008.

[5] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt. The m5 simulator: Modeling networked systems. In *Proc. of the 39th Int'l Symposium on Microarchitecture*, volume 26, pages 52–60, 2006.

[6] F. Fazzino, M. Palesi, and D. Patti. Noxim: Network-on-chip simulator, 2008.

[7] J. Hestness, B. Grot, and S. W. Keckler. Netrace: Dependency-driven trace-based network-on-chip simulation. In *Proceedings of the Third International Workshop on Network on Chip Architectures*, pages 31–36, New York, NY, USA, 2010. ACM.

[8] C. Hughes, V. Pai, P. Ranganathan, and S. Adve. Rsim: Simulating shared-memory multiprocessors with ilp processors. *Computer*, 35(2):40–49, 2002.

[9] A. B. Kahng, B. Lin, K. Samadi, and R. S. Ramanujam. Trace-driven optimization of networks-on-chip configurations. In *Proceedings of the 47th Design Automation Conference*, pages 437–442, New York, NY, USA, 2010. ACM.

[10] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33:92–99, November 2005.

[11] C. Nitta, M. Farrens, K. Macdonald, and V. Akella. Inferring packet dependencies to improve trace based simulation of on-chip networks. In *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '11, pages 153–160, New York, NY, USA, 2011. ACM.

[12] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanović, and D. Patterson. A case for FAME: FPGA architecture model execution. In *Proc. of the 37th Annual Int'l Symposium on Computer Architecture*, pages 290–301, 2010.

[13] F. Trivino, F. J. Andujar, F. J. Alfaro, J. L. Sanchez, and A. Ros. Self-related traces: An alternative to full-system simulation for nocs. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 819 –824, july 2011.