Synthetic Traffic Generation as a Tool for Dynamic Interconnect Evaluation

Wim Heirman, Joni Dambre, and Jan Van Campenhout
Ghent University, ELIS
Sint-Pietersnieuwstraat 41
9000 Ghent, Belgium

wim.heirman@ugent.be, joni.dambre@ugent.be, jan.vancampenhout@ugent.be

ABSTRACT

Run-time reconfigurable interconnection networks can provide significant performance gains in shared-memory multiprocessor systems. However, designing such networks is hard, requiring detailed but slow execution-driven simulations, since faster methods are currently not suitable for use with dynamic network topologies. In this paper, we extend one of these methods, synthetic traffic generation, to incorporate the dynamic traffic behavior necessary to accurately determine the performance of a reconfigurable network. Our synthetic traffic flow has the same characteristics as the flow resulting from an execution driven simulation, but can be much shorter: we can gain a reduction in simulation time of up to $100\times$ at only a limited expense in accuracy. This way, it is possible to quickly analyze the dynamic interconnect requirements of an application and evaluate various aspects of a proposed reconfigurable interconnect implementation.

Categories and Subject Descriptors:

C.1.4 [Processor Architectures]: Parallel Architectures—*Distributed architectures*

General Terms:

Algorithms, Design, Measurement, Performance

Keywords:

Synthetic traffic generation, reconfigurable interconnect, dynamic interconnect requirements

1. INTRODUCTION

Traffic patterns on an interprocessor communication network are far from uniform. This makes the load over the different network links vary greatly across individual links, as well as over time when the application executed on the multiprocessor machine goes through different phases, or when different applications are executed. Most fixed-topology networks are therefore a suboptimal match for realistic network loads.

One solution to this problem is to employ a reconfigurable network, which has a topology that can be changed at run-

time, so as to most efficiently (with the highest performance, the lowest power consumption, or a useful tradeoff between them) accommodate the traffic pattern at each point in time. We have previously introduced a generalized architecture in which a fixed base network with regular topology is augmented with reconfigurable *extra links* that can be placed between arbitrary node pairs [4]. When the network traffic changes, the extra links are ‘moved’ to positions where contention on the base network is most significant. A possible implementation using optical interconnection technologies and tunable lasers to provide the reconfiguration aspect can be found in [1].

While adding a reconfigurable network to a multiprocessor machine can greatly enhance performance, designing such a network presents the network designer with additional problems. To evaluate different implementation proposals, and especially when doing an exploration of the design space, a large number of simulations are needed. Traditionally, one tries to reduce the number of slow and detailed execution-driven simulations by using synthetic traffic generators [7] that employ a standard traffic pattern (*uniform*, *hotspot*, *perfect shuffle*, ...). However, these existing traffic generators fail to accurately capture the time-varying behavior of the traffic pattern which is exploited by reconfiguration. Additionally, they generate single, independent packets. While this may suffice for the evaluation of message-passing architectures, traffic inside a shared-memory machine is characterized by the fact that messages are highly structured in (sometimes multi-level) request-response structures. This makes for more correlation between individual messages, which can invalidate assumptions based on the independence of packets that are commonly made when working with existing generators.

A synthetic traffic generator that does model dynamic traffic behavior would therefore constitute a useful tool. It can be positioned in the design flow after an initial design-space exploration, for instance using our prediction models presented in [4], and the final tuning and verification of network performance through execution-driven simulation. When implemented correctly, synthetic traffic has the advantage that the relevant traffic properties of a real traffic flow are preserved, but that the flow can be much shorter, equally reducing simulation time. In addition, the processors, caches etc., of which detailed models are needed in an execution-driven simulation, no longer need to be considered when synthetic traffic is used. This greatly reduces the complexity of the simulator and again decreases the simulation time significantly. In contrast to our prediction mod-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SLIP’07, March 17–18, 2007, Austin, Texas, USA.

Copyright 2007 ACM 978-1-59593-622-6/07/0003 ...\$5.00.

Figure 1: Schematic overview of a shared-memory multiprocessor machine. All memory can be accessed by every processor, non-local accesses are serviced by the network interfaces (NI) which generate the necessary network packets.

els, which can only predict some aspects of network performance like average packet or memory access latency, a synthetic traffic flow can be fed to a detailed network simulation model, allowing all network parameters and effects (including routing protocols, congestion, possibility of deadlock, ...) to be considered. The correlation between packets can be maintained by generating what we will call *packet groups*. These are sets of packets that are generated as a unit, they will stay connected throughout the simulation so that proper sequencing of related packets is maintained.

In this paper, we present our synthetic traffic generator and use it to explore the behavior of a number of reconfigurable networks. Section 2 first summarizes the architecture of both the shared-memory machine and the reconfigurable network that were used in this study. In section 3, relevant details about the implementation of our simulator are provided. The traffic generation algorithm is presented in section 4. Section 5 uses this model to explore several properties of some example network implementations. The variability of these results is discussed, and the relation between the length of a trace and the accuracy of the resulting measurements is explored. Section 6 provides some opportunities for future work, we summarize our conclusions in section 7.

2. SYSTEM ARCHITECTURE

2.1 Multiprocessor architecture

We have based our study on a multiprocessor machine that implements a hardware-based shared-memory model (Figure 1). This requires a tightly coupled machine, usually with a proprietary interconnection technology yielding high throughput (tens of Gbps per processor) and very low latency (down to a few hundred nanoseconds). This makes them suitable for solving problems that can only be parallelized into tightly coupled subproblems (i.e., that communicate often). Communication is automatically initiated when a processor tries to access a word in memory that is not on the local node. This happens without programmer's intervention making such machines relatively easy to program. But since network traffic is now largely hidden from the programmer, the performance of such machines is very vulnerable to increased network latencies.

Modern examples of this class of machines range from small, 2- or 4-way SMP server machines (including multi-core processors where several CPU cores are on the same sili-

con chip), over mainframes with tens of processors (Sun Fire, IBM iSeries), up to supercomputers with hundreds of processors (SGI Altix, Cray X1). For several important applications, the performance of the larger types of these machines is already interconnect limited [9]. Also, their interconnection networks have been moving away from topologies with uniform latency such as busses, into highly non-uniform ones where latencies between pairs of nodes can vary by a large degree. To effectively use these machines, data and processes that communicate often should be clustered to neighboring network nodes. However, this clustering problem often cannot be solved adequately, because the communication is of a higher degree than the network. Also communication requirements can change too rapidly for a software approach to work effectively. This makes these types of machines very likely candidates for the application of reconfigurable interconnection networks.

For this study we consider a machine in which coherence is maintained through a directory based coherence protocol. This protocol is, in one of its variants, used in all modern large shared-memory machines. In this computing model, every processor can address all memory in the system. Accesses to words that are allocated on the same node as the processor go directly to local memory, accesses to other words are intercepted by the network interface, which generate the necessary network packets requesting the corresponding word from its home node. Since processors are allowed to keep a copy of remote words in their own caches, the network interfaces also enforce cache coherence which again causes network traffic and may stall the processor for one or more network round-trip times (in the order of hundreds of nanoseconds, even on a highly performant, custom designed interconnection network). This is much more than the time that out-of-order processors can occupy with other, non-dependent instructions, but not enough for the operating system to schedule another thread. This makes it very difficult to effectively hide the communication latency, making system performance very much dependent on network latency.

2.2 Reconfigurable network architecture

Our network architecture starts from a base network with fixed topology. In addition, we provide a second network that can realize a limited number of connections between arbitrary node pairs – these will be referred to as *extra links* or *elinks*. A schematic overview is given in Figure 2. To keep the complexity of the routing and elink selection algorithms acceptable, packets can use a combination of base network links and at most one elink on their path from source to destination.

The elinks are placed such that most of the traffic has a short path (a low number of intermediate nodes) between source and destination. This way a large percentage of packets has a correspondingly low (uncongested) latency. In addition, congestion is lowered because heavy traffic is no longer spread out over a large number of intermediate links. A heuristic is used that tries to minimize the aggregate distance traveled multiplied by the size of each packet sent over the network, under a set of implementation-specific conditions: the maximum number of elinks n , the number of elinks that can terminate at one node (the *fanout*, f), etc. After each interval of length Δt (the reconfiguration interval), a new optimum is computed using the traffic pat-

Figure 2: The network consists of a base network with regular topology, augmented with a limited number of reconfigurable links.

tern measured in the *previous* interval, and the elinks are reconfigured (Figure 3).

The reconfiguration interval must be chosen short enough so that traffic doesn't change too much between intervals, otherwise the elink placement would be suboptimal. This limits the choice of the reconfiguration technology to one that has a switching time much shorter than the reconfiguration interval. In this work, we assume this to be the case and further ignore the switching times.

3. METHODOLOGY

3.1 Simulation platform

We have based our simulation platform on the commercially available Simics simulator [6]. It was configured to simulate a multiprocessor machine resembling the Sun Fire 6800 server, with 16 UltraSPARC III processors clocked at 1 GHz and running the Solaris 9 operating system. Stall times for caches and main memory are set to realistic values (2 cycles access time for L1 caches, 19 cycles for L2 and 100 cycles for SDRAM). The directory-based coherence controllers and the interconnection network are custom extensions to Simics, and model a full bit vector directory-based MSI-protocol and a packet-switched 4×4 torus network with contention and cut-through routing. To model the elinks, a number of extra point-to-point links can be added to the torus topology at any point in the simulation.

Since the simulated caches are not infinitely large, the network traffic will be the result of both coherence misses and cold/capacity/conflict misses. To make sure that private data transfer does not become excessive, a first-touch memory allocation was used that places data pages of 8 KiB on the node of the processor that first references them. Also each thread is pinned down to one processor (using the Solaris `processor_bind()` system call), so the thread stays on the same node as its private data for the duration of the program.

The SPLASH-2 benchmark suite [9] was chosen as the workload. It consists of a number of scientific and technical applications using a multi-threaded, shared-memory programming model. Because the default benchmark sizes are too big to simulate their execution in a reasonable time, smaller problem sizes were used. Since this affects the working set, and thus the cache hit rate, the level 2 cache was resized from an actual 8 MiB on a real UltraSPARC III

Figure 3: The observer measures network traffic, and after each interval of length Δt makes a decision where to place the elinks. This calculation takes an amount of time called the *selection time*. Reconfiguration will then take place, during which time the elinks are unusable (*switching time*).

to 512 KiB. Also the associativity was increased to 4-way (compared to 2-way for the US-III) after we experienced excessive conflict misses in Solaris' internal structures with the 2-way caches. Overall, this resulted in realistic 93–97% hit rates for the L2 caches. 50–60% of L2 misses were cataloged as coherence misses (resulting in communication between different processors), the remaining 40–50% were cold/conflict/capacity misses.

3.2 Network architecture

To avoid pinning our discussion down on the peculiarities of a specific network architecture, we test our model with a hypothetical parameterized architecture that provides the infrastructure to potentially place an elink between any two given nodes. Two constraints are made on the set of elinks that are active at the same time: (1) a maximum of n extra links can be active concurrently, and (2) the fanout of each node is limited to f , not including connections to the base network. The time between reconfigurations, called the reconfiguration interval Δt , is the third parameter. The results in this paper will be based on different sets of values for these three parameters. Additionally, results for a network using the selective optical broadcast element described in [1] will be shown as illustration of the performance of an actual implementation. This network can be modeled using $n = 16$, $f = 1$, and additional constraints on which destinations (only 9 out of 16) can be reached from each source node.

3.3 Extra link selection

For every reconfiguration interval, a decision has to be made on which elinks to activate, within the constraints imposed by the architecture, and based on the expected traffic during that interval (in our current implementation, the traffic is expected to be equal to the traffic measured during the previous interval). As explained in section 2.2, we want to minimize the number of hops for most of the traffic. We do this by minimizing a cost function that expresses the total number of network hops traversed by all bytes being transferred. This cost function can be written as

$$C = \sum_{i < j} d(i, j) \cdot T(i, j)$$

with $d(i, j)$ the distance between nodes i and j , which is a function of the elinks that are selected to be active, and $T(i, j)$ the number of bytes exchanged between the node pair

in the time interval of interest. Since elinks are bidirectional, $T(i, j)$ is the sum of traffic in both directions.

The time available to perform the extra link selection is of the same order of magnitude as the switching time, because both need to be significantly shorter than the reconfiguration interval. Since the switching time will typically be in the order of milliseconds, we use a greedy heuristic that can quickly find a set of active elinks that satisfies the constraints imposed by the architecture and has an associated cost close to the global optimum.

4. SYNTHETIC TRAFFIC GENERATION

4.1 Statistical traffic model

We will now present our synthetic traffic generator. First we define properties of the network traffic and measure their values during an execution-driven simulation (i.e. where the benchmark application is in control of the processors, and where the traffic on the network is the result of remote memory accesses performed by this application). This results in a statistical profile of the traffic flow, which is specific for each benchmark application. This profile will later be used to synthesize a new traffic flow.

4.1.1 Packet groups

As mentioned before, network traffic is the result of memory accesses performed by the application. Each memory access results in a variable number of packets sent over the network, structured into request-response structures. When analyzing traffic, and later when synthesizing a traffic flow, we would like to keep this structure of memory access operations intact. Therefore we will analyze, and generate, *groups* of packets, rather than individual packets. This keeps the behavior of the synthetic traffic much closer to that of the real traffic.

The coherence protocol, extensively described in for instance [3], is in charge of supplying remote data words to the processors while keeping cached copies of the same words coherent. A remote memory access starts when the requesting node sends a request message to the home node (which is determined by a subset of the memory address bits). This home node will return the requested data word, possibly after communicating with other *3rd party nodes* to enforce cache coherence. Figure 4 summarizes the situations that can occur. (a) denotes a simple transaction in which only one request-response pair (REQ and REPLY packets) is needed. Situations (b) and (c) require involvement of one or more *3rd party nodes*, here a number of extra request-response pairs are exchanged. They are all started in parallel upon reception of the initial REQ packet, the final REPLY is sent once the last WBreply or INVreply arrives at the home node.

4.1.2 Number of involved nodes

As can be seen in Figure 4, the number of nodes involved in a remote memory operation, which will later be synthesized as one packet group, is either 2, 3 or n ($n > 2$). By properly annotating memory accesses in the log files of an execution-driven simulation, this node count can be determined for each memory access, and a distribution is made. Figure 5(a) shows such a distribution, for the FFT benchmark when run on a 16 processor machine. It can be seen that simple memory accesses, involving no *3rd party nodes*, are the most common. Accesses in which the owner or just

Figure 4: Possible sequences of packets, synthesized as *packet groups*. (a) no 3rd party nodes involved, (b) data is in state modified/exclusive at another node and must first be written back, (c) data is in shared state at other nodes and must be invalidated.

one sharer needs to be contacted account for 13%, accesses in which data on more than one node must be invalidated are evenly distributed and together account for about 1%.

4.1.3 Distribution of home nodes

In a shared-memory machine, each node contains a fraction of total system memory. All this memory is accessible transparently to the processors in a single physical address space. Some bits of the physical address, in our implementation the upper most ones, determine at which node this address is located.

Using memory management units and virtual addressing, available on all current microprocessors, one can play with the virtual to physical address mapping. This way the operating system can, at a page granularity (8 KiB), decide which node data should be placed on. In our simulator this is done using a first-touch algorithm: each page is placed on the node that first writes to it. This way, data private to a thread is always on the same node, requiring no network traffic.

The home node is thus determined by the address that is referenced. Since address streams, even when measured between the L2 cache and main memory, exhibit spatial and temporal locality, we would expect the ‘stream’ of home nodes also to exhibit a high degree of temporal locality (spatial locality in the address stream, when within the same 8 KiB page, translates to the same home node being accessed again; spatial locality in the home nodes would require spatial locality in the address stream beyond page boundaries, which is usually low and is therefore not modeled here).

We measured this temporal locality using the concept of reuse distance [2], this is the number of distinct nodes that are accessed between two subsequent accesses of the same node, by a given requesting node. If the contacted nodes are put on a stack (when contacting a node already on the stack it is moved to the top), the reuse distance is given by the distance of this node to the top of the stack (at the time before the new memory access is performed, so before moving the node to the top). Figure 5(b) shows an example distribution of this reuse distance. We can see that distance 0 occurs very frequently, this means the same node is contacted twice or more in succession. Beyond that the reuse distance drops off sharply, meaning that there is indeed a high degree of temporal locality in which nodes are contacted. This is expected, a longer period during which the same home node remains at the top of the stack would re-

Figure 5: Distributions of (a) the number of involved nodes per memory operation, (b) the reuse distance of home nodes per requesting node, (c) computation time (time between requests from the same node), all for the FFT benchmark executed on 16 processors.

sult in a communication *burst* between requesting and home nodes, which have been previously described in [5].

4.1.4 Distribution of owner and sharing nodes

Most traffic is exchanged between the requesting node and the home node (in **REQ** and **REPLY** packets). While memory accesses requiring invalidations can potentially lead to much more packets, they are also relatively infrequent. Therefore, in order to limit the complexity of our packet generator, we have decided not to model the destination of these write-back and invalidate packets. Only their number (discussed in 4.1.2) is modeled, the destinations will be generated using a uniform distribution in our synthetic traffic. This way the global network load will still be relatively accurate (the correct number of packets will be generated), only the (source, destination) distribution will be distorted slightly.

4.1.5 Time between requests

In a real application execution, a large fraction of time will (hopefully) be spent by the processors doing calculations. At certain instants, these calculations need data in external memory and a remote memory access is performed. An important parameter in this respect is the computation to communication ratio, which tells us whether the execution of a certain application is dominated by useful computation, versus waiting for remote memory accesses. If in our synthetic traffic we would only generate requests and not model the computation time, much more requests would be generated per unit of time, overloading the network and causing much more congestion than there would be in reality. Therefore, we measure the time between subsequent requests from the same node. In the context of communication networks this time is also referred to as the *think time*, during which the processor or user ‘thinks’ about what request he will make next. The distribution of this time as measured during a simulation of the FFT benchmark is shown in Figure 5(c). When generating requests, each node will insert delays between subsequent requests to model this computation time, this way a realistic network load is generated.

4.2 Generating synthetic traffic patterns

The measurements from section 4.1 provide us with a statistical profile about the memory accesses performed, containing the following information:

- the distribution of the number of involved nodes,

- the distribution of the reuse distance of the home node from a certain requesting node,
- the distribution of delays between launching new requests.

Our synthetic traffic generator takes these distributions and generates a ‘script’, one for each node, which is executed by an entity that generates network traffic in subsequent simulations. This script will contain the type of packet group to be generated ((a), (b) or (c) in Figure 4), the identities of the home node and possible 3rd party nodes, and the delay that should be taken into account before launching the next request in the script.

The number of involved nodes and the delay are generated randomly, using a random number generator that matches the distribution given in the profile. For home node identifiers, a reuse distance is generated according to the distribution provided. This reuse distance is used to look up the home node on a stack that contains the last accessed home nodes. After generating each access, this home node is moved to the top of the stack. Identifiers for 3rd party nodes are generated uniformly.

To validate certain assumptions, we also included the possibility of writing a script that closely follows the memory accesses performed in an execution-driven simulation. To this end, a packet group is generated for each memory operation, with the same number of involved nodes and followed by the same delay. The locations of the 3rd party nodes are not maintained but are randomized with a uniform distribution. This enables testing the effects of this simplification.

4.3 Simulating the synthetic traffic flow

For simulations with synthetic packet traces, the same simulation platform is used as for doing execution-driven simulations. This guarantees that the network model used on both cases is identical, and reduces implementation work. The processors, caches and directory controllers are now disconnected, and a special packet generator is connected to each of the network nodes instead. This packet generator creates request packets and injects them into the network, according to the script that was generated previously. Each packet contains a reference to the description of the packet group it belongs to, so when the packet arrives at its destination, the packet generator object at that node knows what actions are required to continue generation of the complete packet group. These actions can be to send a reply packet

Figure 6: Selection of measured network performance indicators for the execution of the FFT benchmark on 3 different networks (rows), relative to the performance of the base network (dotted line with radius 1). From left to right, the columns represent (a) results from an execution-driven simulation, (b) a trace-driven simulation with the trace provided by the simulation from (a), (c) a trace-driven simulation with the trace provided by a simulation with only the base network, and (d) a simulation with a synthetic trace based on data from a base network only simulation.

after a certain amount of time (modeling the time required to look up a data word in main memory), or send further packets (**WBreq** or **INVreq** requests), await arrival of their corresponding replies (**WBreply** or **INVreply**) and only then send the **REPLY** back. **WBreq** and **INVreq** packets contain the same reference so the 3rd party nodes know to send their **WBreply** or **INVreply** to the home node. The packet generators thus perform two actions simultaneously and independently: generate new requests according to the script provided, and take part in the completion of requests of other nodes by receiving network packets and sending replies to them. Each packet generator also measures the time that transpires between sending the **REQ** packet for a request and the arrival of the corresponding **REPLY**, so the expected remote memory access latency can be measured.

The CPU time required for the generation and simulation of the synthetic trace was less than 10 minutes, compared to 92 minutes for an execution-driven simulation, both for a simulated time of 90 million clock cycles. This is because, in an execution-driven simulation, most of the computational work is in the instruction set simulation of the 16 UltraSPARC processors which is no longer needed using our technique. In section 5.2 we will show that the simulation time can be further reduced by using shorter synthetic traces, at only a slight expense of accuracy. Note that for each benchmark a single execution-driven simulation will still be necessary to compute the statistical traffic profile, but its cost can be amortized over a large number of trace-based simulations.

5. RESULTS AND DISCUSSION

5.1 Results

Figure 6 shows a number network performance indicators for the FFT benchmark. Seven indicators are shown in a radar plot, allowing a visual comparison between the behaviors of different reconfigurable networks (rows) and simulation methods (columns). Since there is a wide variety in parameters and their scales, each indicator is scaled to its value in a *baseline* simulation, this is an execution driven simulation where only the base network (the 4×4 torus) is active, without elinks.

The seven indicators reported are, from the top and in clockwise direction:

- $E[\text{packet latency}]$: average packet latency,
- $E[\text{memop latency}]$: average memory access latency,
- $E[\text{distance}(\text{packets} \times \text{size})]$: average distance packets need to travel, weighted by their size,
- $P[\text{distance}(\text{packets} \times \text{size}) > 2]$: probability a packet has to travel over more than 2 network hops, again weighted by its size,
- $P[\text{link-congestion} > 0]$: the probability a link causes congestion during a certain time interval,
- $P[\text{packet-congestion} > 0]$: the probability that a packet undergoes congestion on its way from source to destination,
- **EA.cost**: the minimal cost (as defined in 3.3) the extra link selection could achieve, averaged over all reconfiguration intervals, indicating how closely the network can be adapted to the traffic pattern.

When reading the different graphs from left to right, 4 measurements were done of all performance indicators for each network. The leftmost one, (a) *execution-driven*, is a normal execution-driven simulation. This is the most accurate simulation we can do, but also the most time-consuming

Figure 7: Detail for the average packet latency for the FFT benchmarks, in 4 simulation scenarios corresponding to Figure 6(a)-(d), and 5 different networks.

one. It will be used to determine the accuracy of the other steps. Results on the relative run-times of different methods is provided in the following section.

In the next situation, (b) *trace (this network)*, the memory operations from the first simulation are translated to packet groups, as described at the end of section 4.2. The difference between (a) and (b) is due to our approximations when generating the packet traces. Possible reasons are ignoring NAKs and retries, ignoring cache eviction of modified blocks, and the randomization of 3rd party nodes. Also, the generation of requests by different nodes is no longer synchronized: each node executes its script at a pace dependent on the time the individual requests take. If initially there is a high correlation between the behavior of the different nodes, but during the course of the simulation some nodes are sped up or slowed down more than others, this correlation will slowly disappear.

Situation (c) *trace (base network)* resembles (b), but here a trace is used that was extracted from a baseline simulation (i.e., a simulation without extra links). Here the change (or lack thereof) in network traffic can be seen when the same application is executed on different networks. The difference is minor, which means we only need to do one execution-driven simulation (the baseline), and can use its traffic profile to evaluate a large number of other networks. Finally, in situation (d) *synthetic* our synthetic traffic is used.

Figure 7 repeats the results for the average packet latency of the FFT benchmark with 5 different network configurations. In the first one only the base network is active. The next three are instances of our parametric reconfigurable network model with $f = 2$ and varying n and Δt . The final one, *prism*, refers to a possible network implementation described in [1]. The 4 colored bars represent the same situations as Figure 6(a)-(d). Again we see a significant error by moving from execution-driven (a) to trace-driven simulation (b), although the error is still smaller than 10%. Moving from real traces in (b) and (c) to a synthetic trace (d) results in less than 3% additional error. Moreover, the relative change in performance when moving to different networks is maintained, making our method a useful tool during network design.

Figure 8: Accuracy of shorter synthetic packet traces. The indicator ‘average packet latency’ is measured in multiple traces. Average, standard deviation, minimum and maximum (dashed lines) are plotted grouped by trace length.

5.2 Required trace length

The main reason to use synthetic traffic traces was the promise that they could be shorter than a complete execution trace, but still retain all relevant information. The question now is, how much shorter can they be while still providing enough accuracy? This question is addressed in Figure 8.

For this figure a large number of short traces are generated using the profile of the FFT benchmark and executed on a reconfigurable network with parameters $n = 4$, $\Delta t = 100\mu\text{s}$ and $f = 2$. In each trace the network performance indicator ‘average packet latency’ is computed, these measurements are then aggregated for all traces of the same length. Figure 8 shows average (centerline), standard deviation (whiskers), minimum and maximum (dashes) statistics for each of the trace lengths considered.

One can now read from the graph the expected accuracy that shorter synthetic traces would be able to attain: the measurement of a single short trace executing in for instance 1.2M clock cycles could be anywhere between 252 and 264 (minimum and maximum values), and will be 258 on average with a standard deviation of 3.3 cycles. For longer runs this deviation diminishes, at the expense of execution time.

Table 1 compares these results with the required simulation time. From left to right, the table shows the length of the trace (in simulated clock cycles), the standard deviation of different measurements with the same trace length (in cycles), the difference between minimum and maximum measurements (cycles), and the CPU time required for one simulation of a trace of this length (seconds). The last column shows the total CPU time required, including the initial execution-driven simulation to measure the traffic profile, assuming this profile is re-used 100 times.

By comparison, the complete execution of the FFT benchmark takes 89M cycles, and results in an average packet latency of 241 cycles. Since the errors in our trace-driven simulation are systematic (as evident from Figure 7(a) and (d)), this does not mean one should expect a high variability from the synthetic trace-based results. Comparison should therefore be made with a trace-driven simulation using the

Trace length	σ	diff	wall time	+profiling
160 k	6.80	50	2.45	64
640 k	4.37	29	9.79	71
3 M	2.44	14	39.2	100
10 M	1.40	6	157	218
41 M	0.73	2	627	688
execution-driven (89M)	2.48	7	7514	

Table 1: Comparison of variability and run-time between trace- and execution-driven simulations.

complete trace (situation (c) in Figure 7), which yielded a measurement of 258 cycles. This falls within the expected accuracy range for synthetic trace-based results. Moreover, execution-driven simulation induces its own variability. The last line of Table 1 shows this: the stability is only as good as that of a synthetic trace of about a factor of 10 shorter (between 3M and 10M cycles), although the required simulation time is 100 times longer.

6. FUTURE WORK

Some systematic inaccuracies are present in our synthetic trace-based simulations. These should be investigated further, and reduced where possible. One problem may be that the network traffic behavior changes throughout the execution of the program, and that just one traffic profile does not suffice. Using known *program phase behavior* techniques [8] we will investigate this, and explore the benefits of using multiple traffic profiles for each benchmark application, one per program phase, and average the results according to the relative occurrences of each phase.

Also, we would like to use our synthetic traffic techniques to further explore the benefits of reconfigurable networks, especially when moving to larger networks. This is very slow when having to rely on execution-driven simulations alone, but should be much faster when shorter, synthetic packet traces can be used. To this end we will parameterize the traffic profile (the distributions shown in Figure 5). This way it should be possible to generate traces for different benchmarks or execution on larger machines by tuning the profile parameters, rather than measuring the distributions again in execution-driven simulations.

7. CONCLUSIONS

We introduced a synthetic traffic generation algorithm that can be used in the context of shared memory machines and run-time reconfigurable networks, both of which are not sufficiently considered in existing traffic generators. We analyzed the accuracy of our technique to measure a number of network performance indicators, and found that acceptable relative accuracies can be achieved. The required simulation time however is only one tenth of the time required for an execution-driven simulation. It can be reduced further by another factor of 10 by using shorter traces, with almost no additional expense in accuracy.

8. ACKNOWLEDGMENTS

This paper presents research results of the Inter-university Attraction Poles Programs PHOTON (IAP-Phase V) and photonics@be (IAP-Phase VI), initiated by the Belgian State, Prime Minister’s Service, Science Policy Office.

9. REFERENCES

- [1] I. Artundo, L. Desmet, W. Heirman, C. Debaes, J. Dambre, J. Van Campenhout, and H. Thienpont. Selective optical broadcast component for reconfigurable multiprocessor interconnects. *IEEE Journal on Selected Topics in Quantum Electronics: Special Issue on Optical Communication*, 12(4):828–837, July 2006.
- [2] K. Beyls and E. D’Hollander. Reuse distance as a metric for cache behavior. In T. Gonzalez, editor, *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 617–622, Anaheim, California, USA, Aug. 2001.
- [3] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1999.
- [4] W. Heirman, J. Dambre, I. Artundo, C. Debaes, H. Thienpont, D. Stroobandt, and J. Van Campenhout. Predicting reconfigurable interconnect performance in distributed shared-memory systems. *Integration, the VLSI Journal: Special Issue on SLIP’05*, 2007. To appear.
- [5] W. Heirman, J. Dambre, J. Van Campenhout, C. Debaes, and H. Thienpont. Traffic temporal analysis for reconfigurable interconnects in shared-memory systems. In *Proceedings of the 19th IEEE International Parallel & Distributed Processing Symposium*, page 150, Denver, Colorado, Apr. 2005. IEEE Computer Society.
- [6] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.
- [7] F. Ridruejo, A. Gonzalez, and J. Miguel-Alonso. TrGen: a traffic generation system for interconnection network simulators. In *1st. Int. Workshop on Performance Evaluation of Networks for Parallel, Cluster and Grid Computing Systems (PEN-PCGCS’05)*, Oslo, Norway, June 2005.
- [8] F. Vandeputte, L. Eeckhout, and K. De Bosschere. An analysis of program phase behavior and its predictability. In D. Dubois, editor, *Proceedings of the 7th International Conference on Computing Anticipatory Systems (CASYS)*, volume 839, pages 361–370, Melville, New York, June 2006. American Institute of Physics.
- [9] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, 1995.