Composable Resource Sharing Based on Latency-Rate Servers

Benny Åkesson
Eindhoven University of Technology
The Netherlands

Andreas Hansson
Eindhoven University of Technology
The Netherlands

Kees Goossens
NXP Semiconductors Research &
Delft University of Technology
The Netherlands
Trends in MPSoC Design

► Embedded system design gets increasingly complex
  – Moore’s law allows increased component integration
  – Digital convergence creates a market for highly integrated devices

► Systems are implemented as MPSoC platforms with
  – a large number of heterogeneous intellectual property (IP) components
  – many concurrently executing applications with real-time requirements

► Pressure to quickly design systems in a cost-effective manner
Verification Problem

Resource sharing
- is required to reduce cost.
- introduces interference between applications.
- makes it difficult to satisfy real-time requirements.

Verification is a design bottle-neck
- Verification by simulation of use case executing on platform
- Number of use cases grows exponentially with the number of applications
- System-level simulation is slow, resulting in poor coverage
- Reverification required if an application is added or changes behavior

Verification is costly and effort is expected to increase in future!
Predictability

- **Predictable systems** are proposed to reduce verification complexity
  - Isolates applications by providing *lower bounds* on service
  - Example, lower bound on memory bandwidth or CPU cycles

- Verification requires **performance monotonic** execution of application
  - Means that more service cannot reduce performance

- **Performance monotonicity** restricts both applications and hardware
  - Hardware must be free from *timing anomalies*
    • May occur in caches and out-of-order processors, such as PowerPC
  - Applications cannot have *timing dependent behavior*
    • For example changing QoS depending on time
    • Applications can furthermore not be safely distributed
Composability

Applications in a **composable system** are **completely independent**
- Isolated in both value and time domains
- Cannot affect each other’s functional or temporal behavior

Composability simplifies verification for the following five reasons:
1. **Linear verification complexity**
   - Applications can be verified in isolation
2. **Increases simulation speed**
   - Only need to simulate application and its required resources
3. **Incremental verification process**
   - Verification process can start when first IP is available
4. **Increased IP protection**
   - Verification process no longer requires IP of ISVs
5. **Functional verification is simplified**
   - Bugs caused by, e.g. race conditions, are independent of other applications
Existing Approaches

There are currently three approaches to composable systems:

1. **Not sharing** any resources
   - Trivially composable, but prohibitively expensive

2. **Statically schedule** all resource accesses at design time
   - Requires a global notion of time
   - Limited to applications that can be statically scheduled

3. **Share resources at run-time using** time-division multiplexing (TDM)
   - Couples latency and rate
   - Cannot efficiently satisfy tight latency requirements
Contributions

The two main contributions of this paper are:

1. A novel approach to composability that allows resources to be shared using any arbiter in the class of LR servers
   - Provides better service differentiation than if just using TDM

2. An architecture of a resource front end that contains the arbiter
   - Provides composable service for any resource with bounded service time
   - Not limited to inherently composable resources, such as SRAM.
Presentation Outline

Conceptual Overview

Front-end Architecture

Experimental Results

Conclusions
Considered System

- Resource used by requestors
  - Ports on processing elements to which applications are mapped

- Requestors and resource communicate via requests and responses buffered in Request and Response Buffers at the resource.
  - Communicate with DTL/AXI/AHB type of protocols

- Buffer overflow prevented with flow control signals
  - Robust in case an application malfunctions

- Processing elements and interconnect shared in composable way
Our approach to composability is based on predictability.

Service specification based on latency-rate server framework:
- General framework for analyzing scheduling algorithms.
- Allocated bandwidth, $\rho'$, guaranteed after initial service latency, $\Theta$.
- Latency-rate servers are an interface for predictable service.
Benefits of Latency-Rate Servers

► Many well-known arbiters belong to the class
  – Example: TDM, Weighted Round-Robin, Fair Queuing, etc.

► Supports combining simulation-based verification with formal performance analysis with a variety of known frameworks
  – Latency-rate analysis, network calculus and data-flow analysis
Composable Service

- Composable service is provided by emulating maximum interference
  - Responses stored in Response Buffer until a worst-case finishing time
  - Flow control based on worst-case scheduling time
  - Creates a temporally independent interface per requestor

- Works for any predictable resource
  - Means that time to serve a request is bounded

- Composable service with any latency-rate server
  - Not restricted to TDM or static scheduling
Presentation Outline

Conceptual Overview

Front-end Architecture

Experimental Results

Conclusions
 ► Concepts embodied as a **resource front end** with three main blocks and a Configuration Bus
  – Atomizer
  – Delay Block
  – Data Bus

 ► We will look at the different blocks in more detail
The Atomizer splits requests into **atomic service units** (atoms)
- Smaller requests of fixed programmable size and known service time
- Atom size chosen to be minimum size efficiently served by the resource
  - Example: 1 word for SRAM, 4-8 words, or even larger for SDRAM

Atomizer also merges responses
- Size of request stored in a FIFO

**Benefits of the Atomizer**
- Makes design **robust** against malfunctioning requestors
  - We do not rely on characterizations of request sizes
- Homogeneous requests **simplifies** the rest of the architecture
Delay Block Overview

- The purpose is to **absorb jitter** caused by other requestors

- **Emulates worst-case interference** on all signals going to the Atomizer
  - Responses to read requests
  - Flow control signals

- We look into how this is done in more detail
Composable Responses

- Requests arrive in a Transaction Buffer

- A Request Validator computes worst-case starting and finishing times
  - Computed when request has fully arrived and there is space for response
  - These are derived from the latency-rate service bound
  - Bounds stored in FIFO buffers
  - Request is then stored in Request Buffer until scheduled by arbiter

- Responses are stored in Response Buffer until worst-case finishing time
Composable Flow Control

▶ Problem with flow control
  – Requests that are scheduled before worst-case scheduling time release space in Request Buffer prematurely
  – Next request may be admitted earlier into Delay Block resulting in different worst-case finishing time
  – Results in non-composable behavior!

▶ Proposed solution
  – A Flow Controller emulates worst-case Request Buffer filling
  – Space released at worst-case starting time
A Delay Block is programmed with two values
- The service latency of the requestor, \( \Theta \)
- The completion latency of a request, \( 1 / \rho' \)
  - Only a single completion latency is required, since all requests are atoms

The Delay Block can be dynamically disabled
- Done by programming service latency and completion latencies to zero
- Enables providing composable service only to some requestors
- Allows non-real-time requestors to use slack to improve performance
Problem if completion latency is not an integer
- Rounding up reduces throughput
  - Significant for requestors with high allocated rate
- Rounding down may result in non-composable behavior
  - Worst-case finishing times are optimistic

We introduce a discrete approximation mechanism
- Alternates between rounding up and down in a weighted fashion
- Approximation is conservative
- Worst-case finishing time never deviates with more than one clock cycle
Data Bus

- Data Bus is a regular DTL bus

- Requests are scheduled periodically according to latency-rate arbiter
  - Counts down from completion latency

- The id of scheduled requestor is stored in FIFOs
  - Used to demultiplex responses to the appropriate Delay Block
  - Separate FIFOs for reads and writes
    - DTL does not enforce order between these
Conceptual Overview
Front-end Architecture
Experimental Results
Conclusions
Experimental Setup

- We use a SystemC model of a predictable and composable MPSoC

- Traffic Generators mimic processing elements
  - Assumed not to be shared or shared in a composable fashion

- IP components connected using Æthereal network-on-chip
  - Predictable and composable guaranteed throughput connections

- Example resource is a 32-bit SRAM interface running at 200 MHz
  - Atoms are 1 word, which is served in 1 clock cycle by the memory

- Front end uses a Credit-Controlled Static-Priority (CCSP) arbiter
  - Combination of rate regulator and static-priority scheduler
  - Belongs to the class of LR servers
Use Case

Simple use case with 4 requestors
- Mix of reads and writes
- Different request sizes
- CCSP arbiter uses descending priorities
- 100% bandwidth allocated (800 MB/s)

<table>
<thead>
<tr>
<th>Requestor</th>
<th>Type</th>
<th>Size [B]</th>
<th>BW [MB/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>r₀</td>
<td>Read</td>
<td>32</td>
<td>20</td>
</tr>
<tr>
<td>r₁</td>
<td>Read</td>
<td>64</td>
<td>260</td>
</tr>
<tr>
<td>r₂</td>
<td>Read</td>
<td>4</td>
<td>260</td>
</tr>
<tr>
<td>r₃</td>
<td>Write</td>
<td>16</td>
<td>260</td>
</tr>
</tbody>
</table>

800

Example use case
We compare the front-end configuration with CCSP and TDM
- Assume equidistant (best-case) slot allocation for TDM
- TDM is unable to provide low latency to $r_0$ without over allocating
  - Latency and rate is coupled
  - CCSP solves this by assigning high priority

Completion time of $r_1$, $r_2$, and $r_3$ is non-integer ($1 / 0.325 = 3.08$)
- Rounding up reduces throughput to 200 MB/s
- Solved by discrete approximation mechanism

<table>
<thead>
<tr>
<th>Requestor</th>
<th>Type</th>
<th>Size [B]</th>
<th>BW [MB/s]</th>
<th>$\rho'$</th>
<th>$\Theta^{tdm}$ [cc]</th>
<th>$\Theta^{ccsp}$ [cc]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_0$</td>
<td>Read</td>
<td>32</td>
<td>20</td>
<td>0.025</td>
<td>43</td>
<td>4</td>
</tr>
<tr>
<td>$r_1$</td>
<td>Read</td>
<td>64</td>
<td>260</td>
<td>0.325</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>$r_2$</td>
<td>Read</td>
<td>4</td>
<td>260</td>
<td>0.325</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>$r_3$</td>
<td>Write</td>
<td>16</td>
<td>260</td>
<td>0.325</td>
<td>7</td>
<td>13</td>
</tr>
</tbody>
</table>
Front End Behavior

- Use case simulated during 1 ms – displaying first 200 requests from $r_2$
  - Worst-case time between arrival and finishing is not constant
    - Maximum difference in simulation is 115 cycles.
    - Enforcing a constant delay would be very inefficient!
  - Minimum time between actual and worst-case finishing time is 3 cycles
    - Bound is rather tight and can be improved further
  - Worst-case finishing time 32.4% later than actual finishing time on average
    - More difficult to satisfy requirements on average latency
Composable Service

Next, we show that the front end provides composable service

– We simulate the use case twice, but change the request generation for $r_0$
– As a result, the actual finishing times of $r_2$ changes
– However, the worst-case finishing times of $r_2$ are unaffected
  • Responses of $r_2$ are hence released independently of behavior of $r_0$
– Similar experiments have been done with flow control signals

Requestor in composable system is unaffected when higher priority requestor changes behavior.

WARNING: Lower latency might result in missed deadlines in non-composable systems due to timing anomalies!
Presentation Outline

Conceptual Overview
Front-end Architecture
Experimental Results

Conclusions
Conclusions

► We addressed the **increasing verification problem** in SoCs
  – Slow simulation-based verification with **poor coverage**
  – Number of use cases **increase exponentially**

► We presented a **novel approach to composable resource sharing**
  – Idea is to **emulate worst-case interference**
  – Concepts implemented in a **resource front end**

► **Benefits of our approach:**
  – Very flexible
    • Works with **any combination** of predictable resource and latency-rate arbiter
    • Arbiter can be chosen to fit with requestor requirements
  – Does **not have any assumptions** on the applications
  – Composability can be **dynamically enabled / disabled** per requestor
    • Allows slack to be used to improve performance or reduce power