Dynamic Communication Models in Embedded System Co-Simulation

- A technique allowing communication at multiple levels of detail
- The designer is able to dynamically choose appropriate level at different parts of the system

Co-simulation techniques
(Rowson's)

- Good accuracy, bad performance
  - The nano-second accurate processor model
  - The cycle accurate processor model
  - The instruction set accurate processor model
- Good performance, less accurate
  - The model-free synchronizing handshake
  - The virtual operating system
  - The bus functional processor model

(Continues...)

Co-simulation techniques
(continued)

- None of the previous are entirely satisfactory
  - Cycle and nanosecond accurate not fast enough
  - Virtual operating system and the bus functional models only look half of the system:
    - Software, or
    - Hardware
- Synchronizing handshake
  - Gives a way to simulate the complete system without a processor model

(Continues)

Co-simulation techniques
(continued)

- Validating embedded systems
  - Not interested in internal details of processor's operations
  - Interested in
    - Timing
    - Hardware & software:
      - Interfaces
      - Functionality
- Short: Any processor model that executes code properly, providing right interface events at the right times.

(Continues)

Co-simulation techniques
(continued)

- Selected cycle simulation (Wilson)
  - Combines synchronizing handshake and bus functional models
  - System software compiled for the host computer
  - Communication
    - Sends a message to a bus functional unit in the hardware simulator
    - The bus functional unit performs the requested functions and returns any results
  - Useful, <u>but</u> makes synchronization expensive
  - Well suited when synchronization, hardware-software communication are minimal

Co-simulation techniques
(continued)

- Many groups have found
  - It is worthwhile to compile the system software for the simulator host. Then run it as a process on its own, utilizing OS based primitives for comm. w/HW sim.
    - Good performance simulating computational intensive systems, not requiring much fine grained communication.
    - Communication between HW sim. And SW sim. is expensive => limits speedup possible
    - Especially simulating comm. or synchronizing intensive systems.

The approach

- Many embedded systems have great deal of communication between hardware & software
  - Maybe true even when trivial communication is abstracted away
  - One often only need data transferred, not how it is transferred
- Debugging hardware using logic analyzer
  - Will not collect details of all the transactions
  - Methodically check the interfaces in a reasonable order, narrowing suspected bugs
  - Allows: programming of filters, triggers and trigger windows

The approach

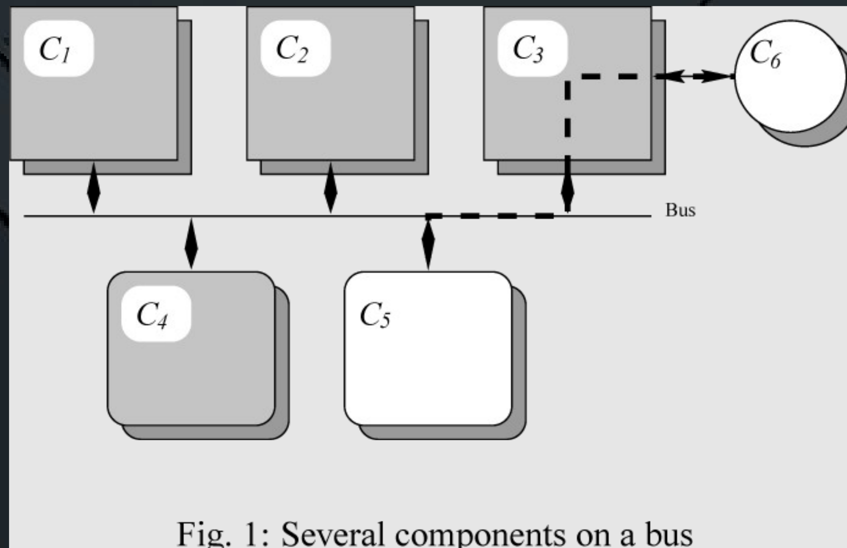


Fig. 1: Several components on a bus

- Example scenario
- 5 Components connected directly to a bus
- C6 Connected through C3
- Data from C5 to C6 is sporadically corrupted
- Would like to avoid details that cost time when simulating
- Tell simulator detail level for data transfers

Pia co-simulation tools

- Mechanism for
  - specifying multiple communication models
    - For each interface
    - Dynamical switching at simulation time
- Processor synchronization in scheduler
  - No need for synchronization through OS primitives
- Mechanism for
  - Processors and processor block to
    - Be described at different levels of detail
    - simulators,re-compiled object code and source code compiled for simulator host

Pia co-simulation tools

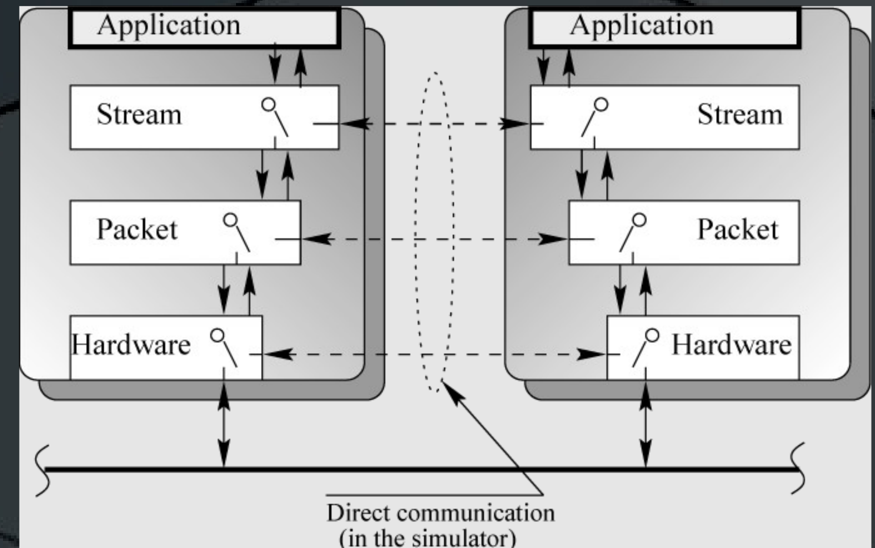
- Implemented as a Ptolemy domain
- Ptolemy
  - Is a simulation environment
  - Has a graphical user interface
  - Provides primitives and tools for writing new simulation domains
  - Primitives for matching communications semantics between domains
- Language for component and interface specifications
  - Specify communications at multiple levels of detail

Driver abstraction and fast communication

- Software communicates with SW/HW components through drivers
- Drivers are often hierarchically layered
  - eg., network app. Calls a stream level driver that calls a packet level driver that calls other drivers, which actually manipulates hardware states
- *Unit of transfer* amount of information transferred on a single transaction
- *Transaction overhead* cost to perform a transaction
- Unit of transfer usually increases for each layer higher up in the hierarchy
  - Use this for fast communication models

Pia tools

- Allows designer to define multiple communications models for *each* layer
- Eg., stream layers can communicate directly with each other
  - or with packet layer
  - lower layers



Synchronization

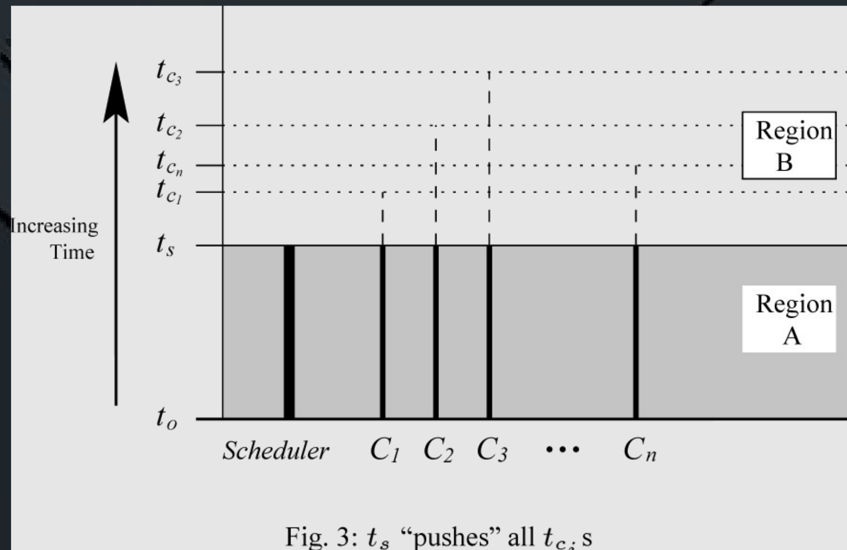


Fig. 3: t_s "pushes" all t_{c_i}s

- Each component has a local time
- System scheduler manages system time
- System time
  - always <= to all local times
  - monotonically increasing
- Components periodically synchronizes its local time with system time

Synchronization

- Active components
  - informs scheduler when to run next time
- Reactive components
  - requires stimuli to run
  - set to system time if (local time < system time)
    - can be done lazily
- Region B are allowed to perform *sends*, not *gets*
- *get*-operations has to wait for system time to catch up
- Components executes *get* when
  (local time == system time)

Pia objects

- Four types of objects
- Components
  - Interfaces and behavior, typically physical
- Interfaces
  - Ports, driver routines, simple asynchronous event handlers
- Ports
  - Directly connect an interface to the outside world
- Wires
  - Used to connect ports

Inheritance

- Prototype-based object-oriented
  - <u>Not</u> class-based
- Object definition creates an instance of an object rather than a class of objects
- Possible to define interfaces which never get imported or used directly
  - Called *abstract interfaces* or *interface templates*
- Interface inheritance suited for libraries
  - eg., an abstract memory interface
- **Interface** DRAM **inherits** memory {…}

Pia Language syntax

- Similar to C++
  - But Pia is not general-purpose OO language
- Has specific types of object to be used in certain ways
- Implicitly active while C++ are passive
- Aware of two different types of methods
  - Driver routines (seqs)
    - Functions that communicate with hardware
  - Handlers
    - Asynchronous event or interrupt handlers

Example 3.1 A simple interface description in Pia

```
interface memory {
  merge Bus {
    inout [32] DATA;
    inout [32] ADDRESS;
    out WR;
    out RD;
  };
  times { write_pulse_duration:6ns,
    address_setup_time:3ns,
    data_setup_time:3ns,
    data_hold_time:3ns };
  init {
    // Make sure we aren't driving the bus
    Bus.DATA.set_tri();
    Bus.ADDRESS.set_tri(); Bus.send();
  };
  seq Write (int A, int D) {
    Bus.ADDRESS = A; Bus.DATA = D;
    advance(address_setup_time);
    Bus.WR = 0; advance(write_pulse_duration);
    Bus.WR = 1; advance(data_hold_time);
    Bus.ADDRESS.set_tri(); Bus.DATA.set_tri();
  };
  seq Read (int A) {
    Bus.ADDRESS = A;
    advance(address_setup_time);
    Bus.RD = 0; advance(data_hold_time); sync();
    Bus.RD = 1;
    return(DATA);
  };
}
```

Dynamic “seqs” and driver abstraction

- Mechanism for switching between different implementations of driver routines
- A *seq* name can refer many multiple driver routines
  - Choice of actual routine deferred until runtime
- An integer describes the *runlevel* or level of communications detail at which the interface is running

Example 3.2 Dynamically dispatched *Write* seqs

```
defval hardwareLevel 0;
defval sharedMemory 4;
  ...
  seq Write (int A, int D);

  seq [hardwareLevel] Write (int A, int D) {
    Bus.ADDRESS = A; Bus.DATA = D;
    advance(address_setup_time);
    Bus.WR = 0; advance(write_pulse_duration);
    Bus.WR = 1; advance(data_hold_time);
    Bus.ADDRESS.set_tri(); Bus.DATA.set_tri();
  };

  seq [sharedMemory] Write (int A, int D) {
    MemArray[A] = D;
    advance(address_setup_time +
                write_pulse_duration +
                data_hold_time);
  };
```

Implementation

- Current implementation
  - Pia language compiler
  - Ptolemy simulation domain
  - Several simulated peripheral and test devices

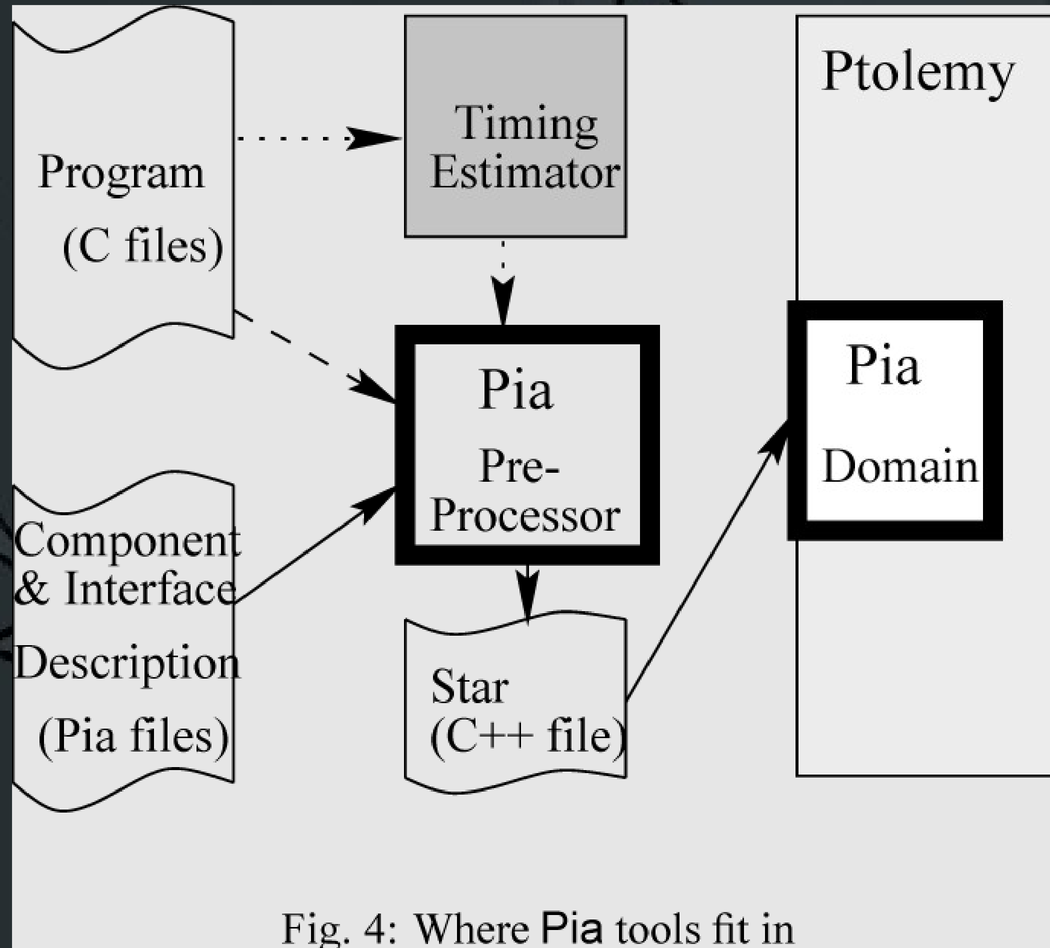


Fig. 4: Where Pia tools fit in

Pia pre-processor

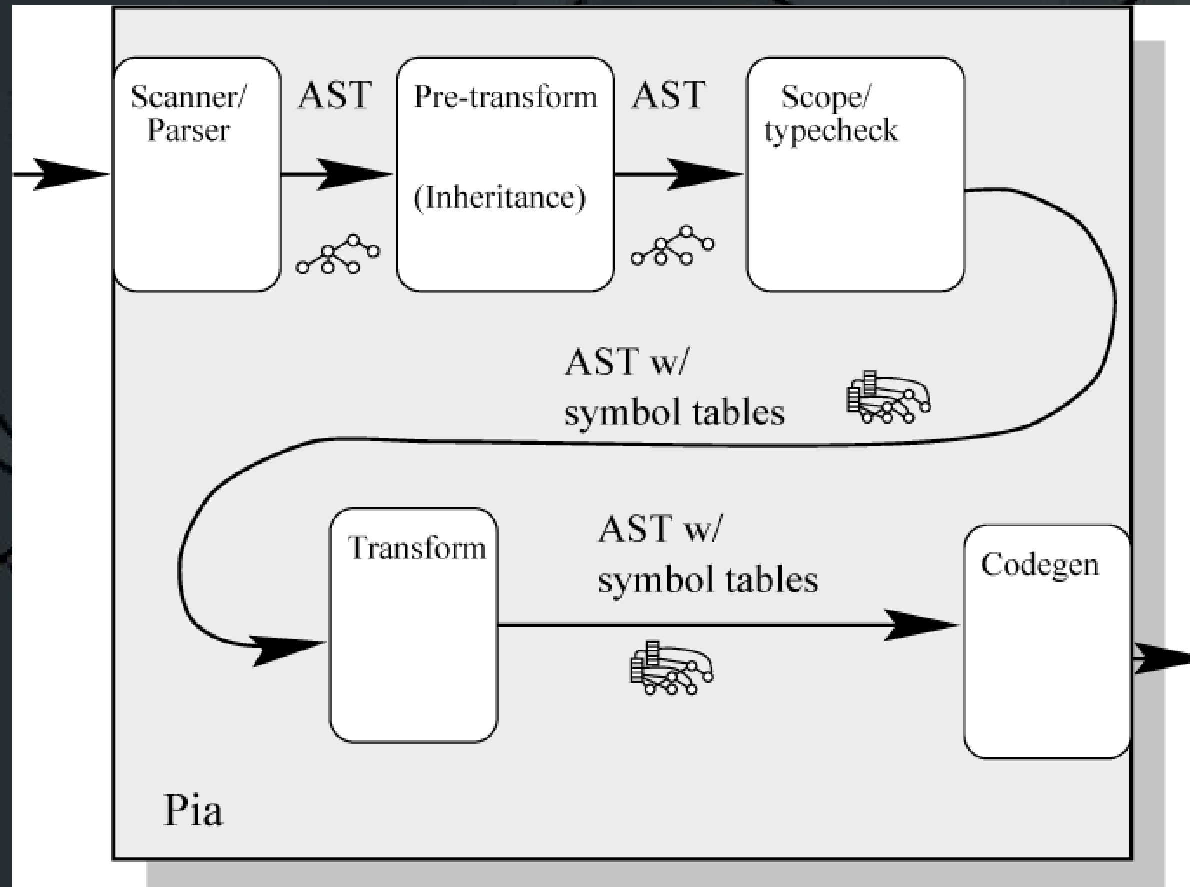


Fig. 5: Pre processor phases

Pia pre-processor

- Scanner/parser generates AST
  - AST is sent through several stages
- Stages
  - *Pre-transform*, resolves inheritance
  - *Scoping*, check definitions into active scope
  - *Transform*, linearizes component description
    - Breaks into atomically executable pieces
  - *Codegen* produces Ptolemy star
- Resulting star can be loaded into Ptolemy in the Pia domain

The Pia simulation domain

- Similar to Ptolemy's standard DE domain
  - Difference: Pia is timed, but also works with untimed components as well
- Pia domain
  - Portholes automatically consume particles
  - Buffer values, makes it possible to check current value of a wire
  - Portholes have a *transition()* method
- Pia scheduler contains *checkpoint* and *restore* methods

Experiments

- Experiment
  - 1: Page transfer
  - 2: Robot vision subsystem

Experiment 1: Page transfer

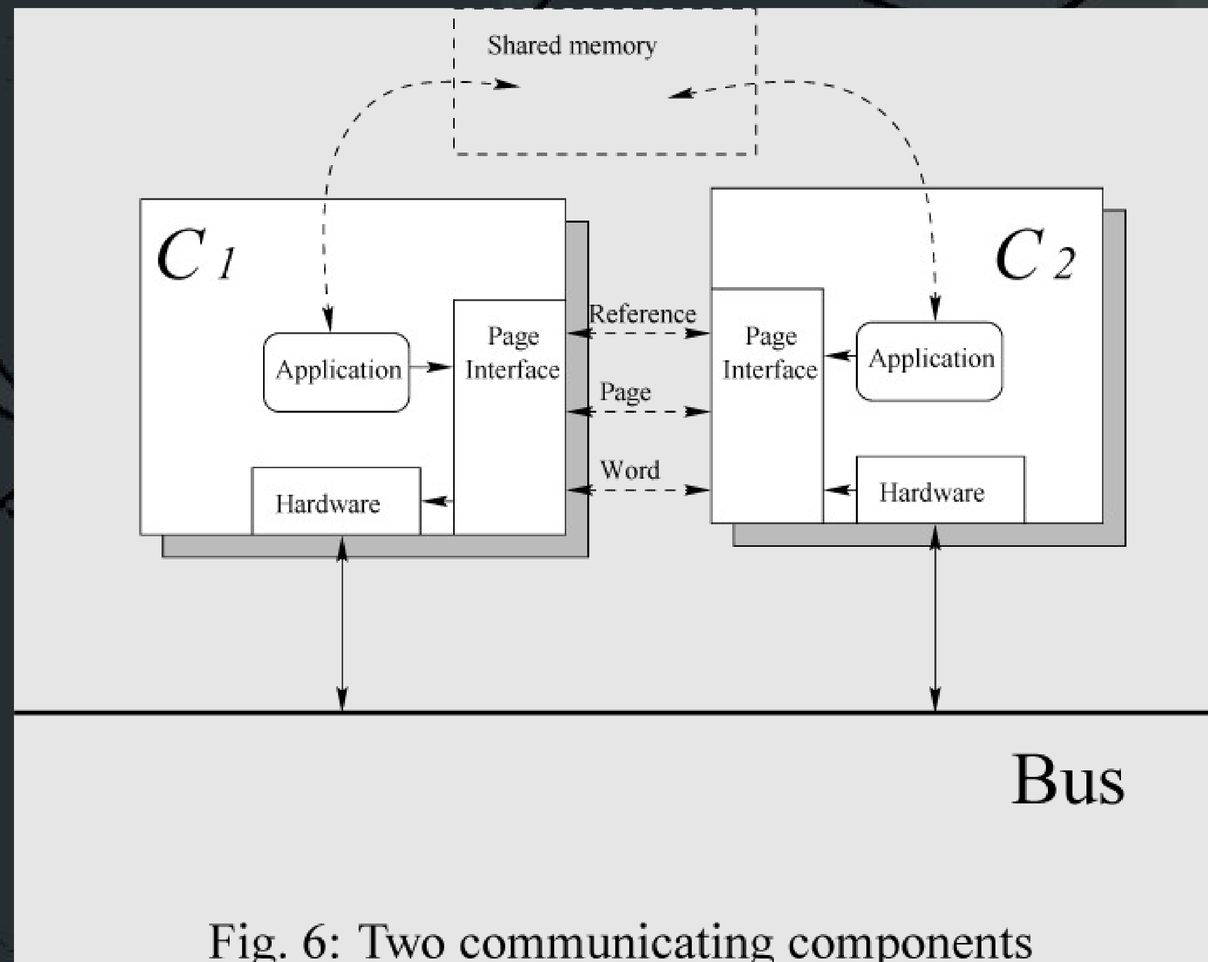


Fig. 6: Two communicating components

Experiment 1: Page transfer

- C1 fills 8KB pages with data, then sends them to C2
- Four separate communication models
  - Hardware – each word (4B) is sent to C2
  - Fast word – each word is directly txed to C2
  - Page transfer – the entire page is copied into a buffer, then transmitted to C2
  - Reference transfer – only a reference to the page is transmitted to C2

Experiment 1: Page transfer

- Table 1 show result for 64 page writes
  - 25 times speedup between mode 1 and 2
  - 96% of time in mode 1 on overhead
    - Function of citizens on the bus

Mode	Transfer time
1	548.91 seconds
2	21.95 seconds
3	1.80 seconds
4	0.24 seconds

Tab. 1: Simulation times for 64 page writes

Experiment 2: Robot vision subsystem

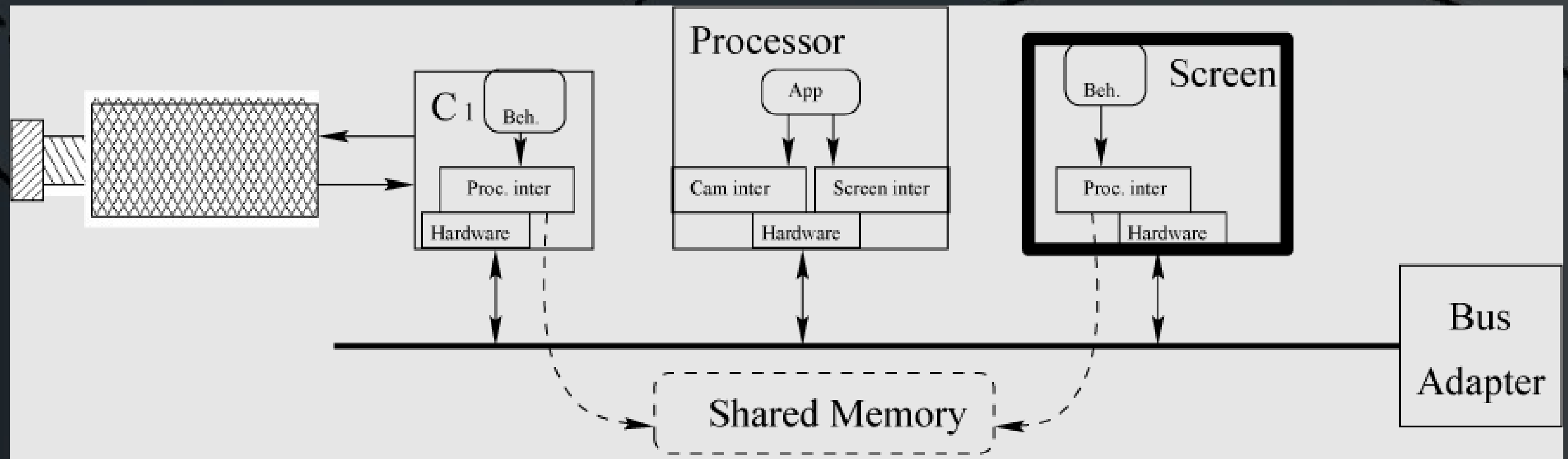


Fig. 7: Robot vision subsystem

Experiment 2:
Robot vision subsystem

- Test the effective speedup in a mixed dense and sparse communication path system with non-trivial computation
  - Image processing system
- Main processors issues commands
- System contains
  - Camera with a controller
  - A processor
  - LCD screen
- Bus adapter generates an interrupt whenever it gets a command from the main processor

Experiment 2: Robot vision subsystem

- On receiving interrupt
    1. Reads the command, and data from bus adapter
    2. Performs the requested action, and
    3. Writes a return value to the bus adapter
- On each “grab the frame” command, copies new frame to the LCD screen
    1. Processor <=> bus adapter (low density),
    2. Processor <=> camera (high density), and
    3. Processor <=> LCD. (high density).

Experiment 2: Robot vision subsystem

- Two communication modes for high density path
  - Hardware accurate mode
  - Shared memory mode
- Specification tested
  1. All communication through the bus
  2. Camera and processor communicate through shared memory, the rest interface accurate
  3. Camera and screen both communicate with processor through shared memory, the rest of the system, interface accurate.

Experiment 2: Robot vision subsystem

- Table 2 shows times to complete the following commands
  1. Get a new frame, and
  2. Find the center of the brightest section of the center scan line

Configuration	Transfer time
1	228.78 seconds
2	115.64 seconds
3	0.92 seconds

Tab. 2: Sequence times

Future work

- Automatically generate fast communication models from high level system descriptions
- Speculative communication through statically determined reasonably large units of transfer
  - Fast transfer that rarely need to be undone
- Debugging support (modified gdb?)

Conclusion

- Hardware-software co-simulation of embedded systems
  - Performs poorly usually when too detailed
- Processor internal details can be abstracted away in most cases
- Communication a bigger problem, since level of detail is not always known a priori
- Focused on techniques giving the ability to
  - focus in on certain communications
  - representing the rest in less detail
- Speedup vary depending on ratio between communication and computation