RAMP-White: An FPGA-Based Coherent Shared Memory Parallel Computer Emulator

Hari Angepat, Dam Sunwoo, Derek Chiou
Electrical and Computer Engineering, UT Austin
{angepat, sunwoo, derek}@ece.utexas.edu

***Abstract*— This paper describes the architecture of RAMP-White, an FPGA-based coherent shared memory machine. RAMP-White aims to provide a configurable coherent shared memory research platform while also demonstrating how large systems can be assembled using RAMP building blocks. The platform is sufficiently flexible to handle different coherency protocols, memory hierarchies and even processor ISAs and implementations.**

I. Introduction

¡¡¡¡¡¡¡ intro.tex ======= RAMP-White is a shared-memory FPGA-based architecture currently under development at UT-Austin. The project is one of the three initial prototypes associated with the RAMP (Research Accelerator for Multiple Processors) [8] initiative. RAMP-White is designed to provide a flexible baseline platform with which to perform research into shared memory architectures. ¿¿¿¿¿¿¿ 1.6

RAMP-White is an FPGA-based coherent shared memory architecture and is one of the three initial prototypes being developed by the RAMP (Research Accelerator for Multiple Processors) group. In the spirit of the RAMP project, RAMP-White has been architected to provide a flexible baseline research platform, rather than an aggressive implementation of coherent shared memory.

To provide platform flexibility, the components from which RAMP-White is constructed are defined and implemented with composability in mind. The use of generalized component interfaces and messages enables the components to be connected in a variety of configurations. Additionally, the RAMP-White architecture separates the coherency protocol and the network topology from components that are common to all shared memory machines, permitting them to be easily modified or exchanged for others. Both design objectives enable extensive experimentation with alternatives while leaving large portions of the platform unchanged.

In keeping with the general goals of the RAMP initiative, the platform is designed to support large-scale investigations in parallel systems. By supporting a variety of interconnection schemes the platform can scale from small dual-core configurations hosted on a single development board to much larger many-core platforms hosted on a cluster of FPGA development boards. The RAMP-White architecture is designed around point-to-point connections, facilitating the use of the RAMP Description Language (RDL) to specify and implement connections.

II. System Overview

The basic components of the RAMP-White design can be grouped into several categories:

- Node Interconnection: (Intersection Unit, Cache Coherency Engine)
- Processor Components: (Processor Cores, Caches)
- Network: (Network Interface, Router)
- Memory: (Memory Controller)
- Platform Support: (IO Bus, Timer, Ethernet, MP Interrupt Controller)

A brief description of the functionality of the various components is presented below. Specifications for the baseline configuration of the platform can be found in Appendix A which details component and message interfaces, as well as board-specific hardware support.

A. Intersection Unit (IU)

The intersection unit (IU) is a switch between the processor, local memory, IO, a coherency engine and the network interface. The initial IU implementation will support exactly one processor each. The IU manages the various queues for the ports and keeps track of outstanding requests for matching replies. It also exposes a set of programmable address map registers to indicate which region of global memory the unit is responsible for. There are four functional interfaces to the IU:

- Processor port: Interfaces with the processor or a coherent cache
- Memory Controller port: Interfaces with the physical memory attached to node

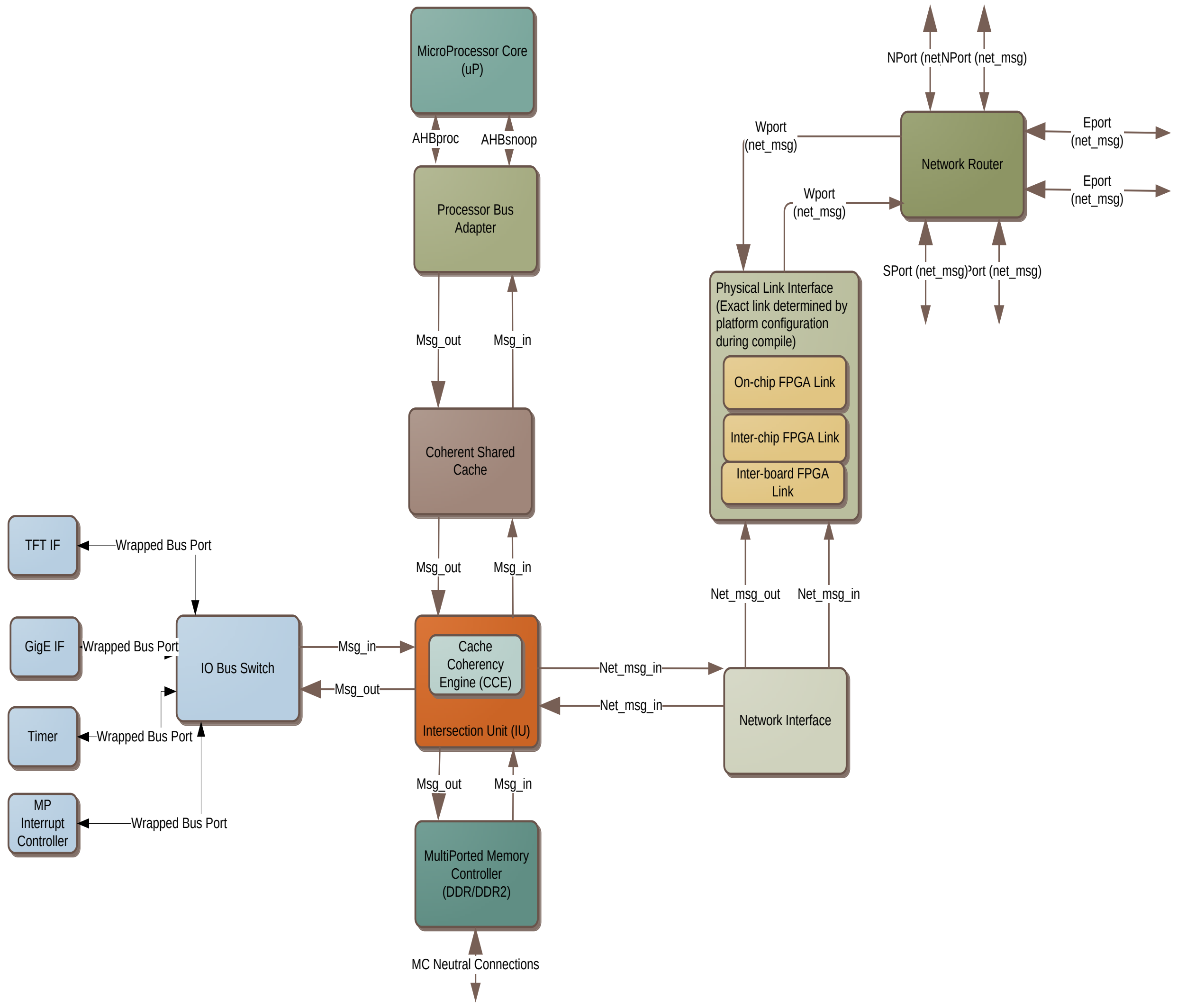

Fig. 1. Baseline RAMP-White Platform Overview

- Network Interface port: Interfaces with the network for sending request/replies to other nodes in the system
- IO Bridge: Link to IO Bus and associated devices

B. *Processors*

RAMP-White processors are full processors appropriately wrapped to interface with the rest of RAMP-White. The processors used in RAMP-White are restricted to those that support full operating systems and thus must include an MMU. These processors can be either hard-core processors embedded in some FPGA parts, or soft-cores provided as Verilog/VHDL sources. We have selected two processors to be supported in the initial version of RAMP-White: the PowerPC 405 as well as the Leon3. The PowerPC 405 is an 32-bit embedded hard-core available in certain Xilinx FPGA parts, while the Leon3 is a 32-bit SparcV8 compatible soft-core from Gaisler Research. RAMP-White will support both cores. We only support the hard-core 405 since the only currently available PowerPC soft core is larger than is practical.

To isolate downstream components from processor-specific bus interface details, each bus is abstracted to a generic bus interface using a bus adapter that provides a PLB or AMBA bus interface to the processor and a RDL connection-based interface to the rest of the RAMP-White system. Bus operations are translated into a standard component message format and vice-versa.

If multiple bus operations are supported by the processor, the bus adapter also keeps track of outstanding message requests so replies can be matched when they return. Furthermore, since the IU only exposes a single bi-directional channel to the processor, at least two virtual channels in each direction will be supported by the adapter. From processor to IU, there is a low priority processor request virtual channel and a high priority snoop reply virtual channel. From IU to processor there is a low priority snoop request virtual channel and a high priority virtual processor reply channel.

C. Coherent Cache (CC)

A coherent cache may be introduced into the system by inserting it between the processor and the IU. The specific implementation of these caches vary depending on their size and resource requirements. In order to emulate larger shared cache structures that may not fit in FPGA Block RAMs alone, the use of host-FPGA DRAM can be used to store cache data and potentially even the tags, depending on the size requirements of the shared cache. The caches must adhere to the same Processor-IU interface.

D. Cache Coherency Engine (CCE)

The cache coherency engine (CCE) implements the logic necessary to support cache coherency protocols through the network. While the IU is responsible for basic request/response buffering from the various input/output ports, the CCE determines how requests are actually serviced. This partitioning allows the CCE to service a single request at a time if desired. It also enables experimentation with different coherency schemes by replacing the CCE.

For the case of a incoherent platform, the CCE degenerates to a set of simple arbiters driven by address mapping. For the case of a common directory-based coherency protocol using MESI, the CCE encloses both the response logic to service requests, as well as the coherency logic to issue and collect coherence messages.

E. Network Interface Unit (NIU)

The network interface block connects with the IU and the network router, using an abstract RDL channel to provide compatibility across the potential physical connections. The module is responsible for message queuing and flow-control. The NIU accepts various memory request/replies from the IU and appends the necessary (src, size) tagging to construct a formatted network message.

F. Network Router

This block is responsible for implementing the actual network topology for interconnecting processors in the system. The block accepts the message from one of the NIUs through a physical channel and routes the formatted message according to routing policies implemented in the block. A credit-based flow control system is used to back-pressure the NIU and prevent buffer overflow for a given destination. The network is assumed to be reliable and loss-less and provide a minimum of two virtual channels for request and reply messages. The platform is designed to support multiple network topologies through the use of a replaceable network router. For smaller development platforms, this may be a single ring.

G. Memory Controller (MC)

The memory controller block is responsible for interfacing with the physical DRAM connected to a given node, providing a abstract logical interface to the IU. To ensure cross platform compatibility, the memory controller presents an asynchronous user interface to upstream blocks. This decouples clock-domains as well as simplifying the memory protocol interface that the memory unit must interact with. In order to support scaling and multiple processors sharing a single set of DIMMs, the memory controller supports multiple ports using an arbitration scheme.

H. IO Bus

In order to support the various IO devices available from either the Gaisler GRLIB IP Library or the Xilinx EDK IP Library, a suitable replacement for their shared bus architectures must be created. With the GRLIB library, the AMBA (AHB/APB/ASB) bus protocols are used for connecting cores, while the EDK library uses the IBM CoreConnect PLB/OPB bus technology. To implement similar functionality, a multi-ported switch can be used to emulate the behavior of a shared bus using only point-to-point connections. If the IO devices need not be emulated directly, they could be grouped together with a single bus-master bridge serving as the point to point connection to the remainder of the system, e.g., an OPB or APB bridge with associated peripherals.

I. Multiprocessor Interrupt Controller (MPIC)

The interrupt controller is responsible for interprocessor interrupts and interrupt vectoring to support SMP operating system support without loading a single processor with all the operating system tasks.

III. System Components

A. Node Connections

To promote interchangeable components and alternative design platforms, a unified message format connecting node components is defined. This message format is used to communicate between the interfaces inside a given node. The format of the message is designed to provide some level of generality in command operation support and provides support for a large variety of cache coherent operations.

The message is composed of a request/reply tag (which is used for fabric priority) followed by several fields. The actual field specification below presents the necessary operations needed to support coherent shared memory memory operations. The primary command field indicates the type of operation being performed (read, write, coherency), while the permission field indicates the coherency permission requested for the operation. This partition allows a large number of cache operations to be implemented, providing support for multiple styles of cache coherency protocols.

The size field is used to allow variable data sizes to be encoded in the message. Currently there are 4 data sizes used: byte, word, double word and full cache-line. The tag field is used to mark an outstanding request so its reply can be matched when it returns to the requester. Finally the address and data fields and encoded in the message. The address specified in the component message format differs from the processor or physical address, which is 32-bits for processors selected for the system. This global address provides the ability to translate or partition nodes, providing larger memory space access.

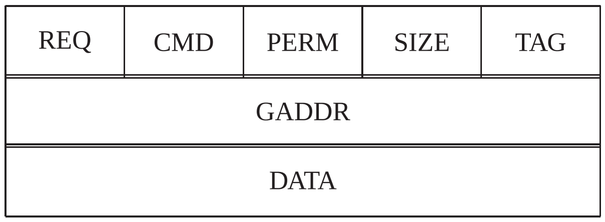

Fig. 2. Component Message Format

With distinct fields for a memory command as well as a coherency permission request, a variety of cache coherency protocols may be supported. Note however not all of these operations may be implemented directly; their support depends on details of the coherency protocol, processor ISA support and specific cache configurations selected for the platform.

Name	Type	Description
REQ	bit	Request/Reply message indicator
CMD	cmd_t	Command to execute
PERM	op_t	Coherence permission requested
SIZE	size_t	Size type for data field
TAG	tag_t	Request/Reply tag for matching
GADDR	gaddr_t	Global address of transaction. The global address provides a means to translate between a processor address into a larger global space for flexibility
DATA	data_t	Data field to accompany transaction. The size of the data is set based on the SIZE field.

TABLE I
Component Message Fields

The node messages transfer data in a common format between modules within a single node.

By decomposing the cmd from the coherency permission request, a large variety of coherency operations may be supported in a concise manner.

B. Network Connections

Similar to the node message format, the network message format provides a common format for communicating between nodes, independent of network topology. A formatted network message is transmitted over a physical channel to the router. The format of the network message adds additional fields to the standard component message, adding source/destination IDs for routing purposes, as well as a message payload size and network tag.

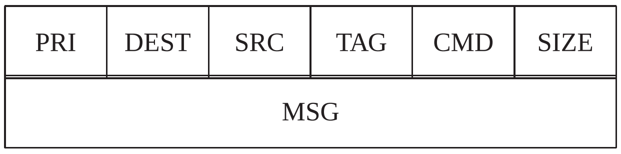

Fig. 3. Network Message Format

The network message introduces ID field used to identify nodes in the platform, which sets an upper limit on the number of procesors supported without changing the network message format. The command and tag fields are specified in the network message format, although in the baseline implementation these fields are not currently used. These fields would allow more complex network messages that could be processed

Field	Operation	State	Description
REQ	Request	0	Indicates message is a request
	Reply	1	Indicates message is a reply
CMD	Read	00	Read transaction
	Write	01	Write transaction
	Coherency	10	Coherency maintenance operation
PERM	Invalidate	00	Requester indicates it does not want a copy of the cache-line
	Shared	01	Requests a shared copy of the cache-line
	Exclusive	10	Requests an exclusive copy of the cache-line (invalidate sharers)
	Flush	11	Requests everyone else should flush the given cacheline
SIZE	Byte	00	Single byte transaction
	Word	01	Single word transaction
	Double-word	10	Double word transaction
	Cache-line	11	Full cache-line transfer

TABLE II
COMPONENT FIELD STATES

CMD/PERM	Action
Read/Shared	memory read
Read/Exclusive	memory read with intent to modify
Write/Shared	write-back line keeping shared copy
Write/Invalidate	write-back line without keeping copy
Write/Exclusive	write-through keeping an exclusive copy
Coherency/Invalidate	I'm dropping the cache line
Coherency/Shared	Clean (force modified copy to be written back but not flushed)
Coherency/Exclusive	upgrade my shared copy to exclusive, forcing other sharers to flush
Coherency/Flush	Flush (flush everyone's copy, including initiator's)

TABLE III
EXAMPLE COHERENT MEMORY REQUEST OPERATIONS

Name	Type	Description
PRI	bit	Priority message indicator
DEST	node_t	Processor NodeID originating the message
SRC	node_t	Processor NodeID destination for the message
TAG	net_tag_t	Network tag for matching replies in network (optional)
CMD	net_cmd_t	Network command for generating messages in network (optional)
SIZE	net_msg_size_t	Message payload size
MSG	node_msg_t	Message payload (component message embedded)

TABLE IV
NETWORK MESSAGE FIELDS

in the router or NIU. Finally a message payload size field is used to support multiple message payload lengths.

In order to implement flow-control between the NIU and router, a simple credit-based scheme is used, that will forward tokens back to NIU when buffer slots are made available.

C. Processor Connections

A bus adapter is used to translate the processor bus operations exposed by a processor core into the standard message format. The adapter implements a master-slave bus interface on the processor side while a single bi-directional channel is exposed on the other side of the adapter. Using the common node message format, bus transactions are transformed into node messages and sent downstream to the IU. This component is specific to the processor core selected for the design as bus-interfaces typically varies between cores.

A single channel to the processor is selected to limit resource utilization and provide an easy path to using RDL for channel implementation. Two virtual channels are used to implement the processor request/replies as well as providing support for snoop request/replies. More virtual channels may be added if a coherence protocol requires additional support.

D. Cache Coherency Engine Connections

The CCE interacts with the IU by processing requests queued in the IU block and sending replies back to the IU. In the baseline configuration, the CCE implements centralized decision logic as to processing of each request incoming to the IU. These requests, which come from IO, processor, or network are serviced by the CCE, then queued for output to one of the three locations.

Further, for a CCE implementing a directory coherence scheme, the CCE will enqueue memory requests to the MC to lookup directory state stored in DRAM.

The actual implementation of the CCE can vary including a incoherent arbiter, hard-wired coherence protocol state-machine or programmable protocol processor. In general a CCE can be seen as a function taking three potential inputs (heads of the queues from IO, network, processor) and returns to one of three locations (heads of queues for IO, network processor). We provide two examples of protocol processing for a simple incoherent process as well as a coherent directory-based access.

1) Example: Incoherent Engine: In the incoherent case, the engine is a simple arbiter that prioritizes between the 3 input request queues it accepts (IO, processor, and network) for processing. The servicing of requests is controlled through a programmable address mapping that allows the engine to determine the destination for a given request (local memory, remote memory or IO).

2) Example: Coherent Engine: To illustrate the coherent case, we assume a directory-based coherency scheme using a MESI protocol with directory entries stored in the DRAM for the node. The CCE stores coherency state in a directory that maintains the list of cache block locations in the system. The local node would issues requests (read, write or coherency) to a home node that is responsible for the given address by consulting the directory. The home node, which maintains a list of all copies of the block in the system would then consults its directory to find the block location. Coherency messages are sent and collected by the home node before finally forwarding the request to a remote node which has the cache-block. The remote node then returns the block to the local node as well as informing the home node of this action.

The actual flow of messages in the operation of the coherent engine is best illustrated by a simple remote read (to get a shared copy) operation:

Requesting Node:

- CPU accesses the cache and misses → bus fill caught by adapter.
- Adapter constructs a message for IU:
  - [request][read][shared][size=cache_line] [adapter_req_tag][paddr padded]
- IU accepts processor message into the inbound processor request queue.
- CCE services outgoing replies then inspects the pending processor request
- CCE checks [addr] field and determines global remote address and home node for addr

Algorithm 1 Incoherent Engine Example

```
if mem.reply then
    check outstanding requests
    if requests[mem.reply.tag]==IO then
        enqueue mem.reply msg into outbound IO buffer
    else if requests[mem.reply.tag]==network then
        compose network message by appending
        src/dest/size fields to mem.reply
        enqueue into outbound network buffer
    else if requests[mem.reply.tag]==proc then
        enqueue mem.reply message into outbound pro-
        cessor buffer
    end if
end if
if net.request then
    decode net.request.gaddr field using addr map
    push into either the IO or memory queue
    if request requires reply then
        set net.request.tag into outstanding request buffer
    end if
end if
if proc.request then
    decode net.request.gaddr convert to gaddr using
    addr map.
    if addr=local then
        push request into outbound queue for IO or local
        memory and service as above.
    else
        mapping is to remote memory, so compose net-
        work 'Read request message'
        enqueue into outbound net buffer
    end if
end if
if net.reply then
    if requests[net.reply.tag]==IO then
        enqueue net.reply msg into outbound IO buffer
    end if
    if requests[net.reply.tag]==proc then
        enqueue net.reply msg into outbound processor
        buffer
    end if
end if
```

- CCE composes a outgoing network message to the home node:
  req [homeID][localID][proc_req_tag][node_msg]
- CCE stores tag of network request to indicate return to processor adapter on reply
- IU sends network message to NIU from outbound network queue
- Router delivers message to remote NIU

Home Node:

- NIU receives request message from network router
- IU accepts message from NIU into an inbound network buffer
- CCE processes request in network queue: read (shared) requested
- CCE-DirectoryController looks up associated entry
  - if(entry.inDRAM) CCE-DC issues [req][read][shared][size=cache_line][dc-req_tag][dir_entry_addr]
- if(entry.isExclusive)
  - CCE composes [request][coherency][shared] to force exclusive node to downgrade to shared [dest=remoteNode][src=homeNode]
  - IU sends message from outbound network queue to NIU
- else if (entry.isModified)
  - CCE composes [request][coherency][shared] to force write-back of line [dest=remoteNode][src=homeNode]
  - IU sends message from outbound network queue to NIU
- else if (entry.isInvalid | entry.isShared)
  - CCE composes [request][read][shared] and enqueues into outgoing MC queue, setting tag
- CCE waits for replies from outstanding requests

Remote Node:

- NIU receives request message from network router
- IU accepts message from NIU into inbound network buffer
- CCE processes request in network queue: [request][coherency][shared])
  - CCE composes [request][coherency][shared] and enqueues into processor inbound port (will go to snoop)
  - CCE waits for matching coherency_shared_reply for acknowledgment
  - CCE composes [reply][coherency][shared][cached_line] to home node [dest=homeNode][src=remoteNode]
- IU sends message from outbound network queue to NIU
- NIU sends reply message to the network router

Home Node:

- NIU receives message from network router
- IU accepts message from NIU into an inbound network buffer
- CCE processes request in network queue [reply][coherency][shared]
  - CCE matches tag to indicate originating source
  - CCE-DM modifies directory entry for address (remoteNode now in shared)
  - CCE composes network message [reply][coherence][shared][proc_req_tag] [dest=localNode][src=homeNode]
- IU sends message from outbound network queue to NIU
- NIU sends reply message to the network router

Local Node:

- NIU receives message from network router into a request buffer
- IU accepts message from NIU into an inbound network buffer
- CCE processes request network queue [reply][coherency][shared]
  - CCE matches tag with outstanding processor request tag
  - CCE strips network header and sends component message to outbound processor queue
- IU sends message from outbound processor queue to NIU
- Processor adapter accepts reply message and matches with adapter-req-tag
- Processor adapter completes bus request from processor to complete the transaction

E. Memory Connections

As components communicate through the standard component message format, the memory controller interface must translate accepted memory request messages into the appropriate lower-level control signals. As described in the overview, the MC provides an abstract logical interface to memory, providing support for a variety of hardware platform memories. To facilitate this the MC interface translate the message into a set of neutral memory request signals that can communicate with a native memory controller core. This decoupling also provides the ability to run the memory controller asynchronously to ensure correct timing requirements are met.

The native memory core itself is specified using a simple logical interface that abstracts the memory implementation details from the interface. This allows the

Name	Type	Description
msg_in	node_msg_t	Component message input from upstream IU
msg_out	node_msg_t	Component message output to upstream IU
req_addr	paddr_t	Physical Memory Address to DDR
req_write_mask	dmask_t	One-hot byte-enable for write transactions
req_write	mem_cmd_t	Read-Write command bit
req_write_data	data_t	Data to be written to the given req_addr
repl_read_data	data_t	Data returned from DDR for the given req_addr
repl_read_valid	bit	Data returned is valid
repl_busy	bit	Memory controller is not ready to accept transactions

TABLE V
MEMORY CONTROLLER INTERFACE

same MC block to connect to a variety of native memory controllers, providing support for multiple hardware platform boards.

IV. SOFTWARE SUPPORT

A. Operating System Support

There are three operating support styles that can be used on the system:

1) SMP Linux: Using a port of SMP-Linux for the Leon3 processor, direct access for IO, synchronization, advanced process scheduling and load balancing are all natively support. This is the ideal case as the shared memory architecture becomes transparent to the software layer. This solution is not completely portable as other processors such as the PPC405 do not currently support such an operating system as of yet.

2) Master-Slave Linux: Instead of fully porting the Linux operating system to our specific SMP environment, the other alternative is to leverage a master-slave configuration. Using a fully functional master service node running the operating system, the slave processing nodes can simply forward requests to the master for handling. This allows the processing nodes to still execute user code and system code without needing modify the operating system significantly. This approach has been used in the RAMP-Red prototype [7] using the exception vector register in the PPC405 to employ a syscall-proxy mechanism. Upon encountering a system call or associated operating system exception, the slave processor simply sends a corresponding request to the master processor who handles the request locally. The results of this operation are sent back to the slave processor for any local updating (ex. updating local TLB as a result of a page fault) and continues executing user code as before. The complications with such a solution are the lack of any thread scheduling (as linux sees only a single uniprocessor machine) and the requirement to port some user-level thread library to support transfering the context of the thread to a remote processor. In addition, this limits the scalability of the system as system calls are being serviced by a single node.

3) Multiple-Image Linux with Segmented Global Memory: To create a scaleable solution without fully porting to a SMP OS environment, multiple copies of the Linux operating system may be booted, one per node. Each node would boot out of their own private memory space, with distributed global memory allocated to a fixed segment of the global address space. The OS would provide shared memory access to this region through a mmap system call. This allows the treatment of the shared global address space as a single large peripheral from the application, providing a map from its virtual space to the physical shared space.

B. Configuration Network

Given the large number of processors to be supported by the platform, a method of configuring the platform (with multiple FPGAs per board and multiple boards) is required. With the use of RDL, much of this complexity is reduced due to the provisioning of debug and monitoring channels as well as configuration of the FPGA platform itself.

V. FAST ON RAMP

In this section, we describe usage models that can be run on the RAMP-White platform.

The first and most obvious option is to run applications natively on the two embedded PPC405 cores. This option will clearly be the one with the highest performance. However, we will be constrained by the relatively stripped-down PPC405 ISA, which is mainly targeted for embedded environments. In additionm, We will not any capability to modify the ISA, leaving the user with no flexibility at all.

Another option is to make use of off-the-shelf emulators to emulate the functionality of a different machine. An emulator will allow us to run on any ISA, including the full blown PowerPC ISA. We could run cross-architecture simulations, where applications on different cores run on different ISAs while still sharing memory. We could even run one application on the emulator and another application on the embedded PPC405 hard-core natively. Such heterogeneous platforms could be made possible through time dilation. The RDL connectors are capable of differentiating host cycles and target cycles, making time dilation easily implementable. Emulators also allow us to modify the ISA and experiment with new or altered instructions. As can be seen, emulators provide us with an enormous amount of flexibility in experimenting on the RAMP-White platform.

Several emulators exist, including QEMU[1], Bochs[6], SimICS[5], M5[2] and Mambo[3]. QEMU is a full system open source emulator that supports almost all modern ISAs including x86, PowerPC, SPARC, ARM and MIPS, making itself a very attractive option. It exploits dynamic translation techniques and a translation cache, eliminating the need to translate a basic block when executed more than once. Through these techniques, QEMU runs at 10 to 20% host CPU speed. With the QEMU accelerator, which tries to run the target code natively on the host processor, QEMU runs at near native speed. This accelerator, however, is only available for x86 emulated on x86 host processors.

While working on the FAST simulators[4], we have ported QEMU to run on the embedded PPC405 and are able to boot unmodified x86 Linux images and run unmodified x86 applications. On average, it runs at about 5 MIPS.

In order to correctly model the SMP environment, synchronization operations need to be implemented precisely. Since the PPC405 caches are incoherent, they need to be turned off. The Load and Reserve instruction (lwarx) and Store Conditional instruction (stwcx) pair is known only to guarantee local atomicity. Memory operations across the bus may get interleaved or out of order. Hence, it is impossible to implement an SMP environment without additional mechanisms. Message passing protocols can be used to ensure atomicity.

VI. RAMP-White Status and Future Plans

The development of the RAMP-White prototype is currently ongoing at UT-Austin. The current development platform utilizes a Xilinx XUP development board which holds one Virtex-II Pro 30 FPGA with two PowerPC 405 cores. We are presently working on validating and verifying the functionality of a baseline configuration platform. The platform is configured as a dual-core PPC405 shared-memory machine that uses the intersection unit and network interface to share memory. The network interface communicates through a ring topology.

Future plans for the development of the platform will include further integration of additional IP cores including a parametrizable network router, processor support for the Leon3 Sparc V8 soft-core, as well as peripheral support from the open-source GRLib IP library.

References

[1] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[2] Nathan L. Binkert, Erik G. Hallnor, and Steven K. Reinhardt. Network-Oriented Full-System Simulation using M5. In *Sixth Workshop on Computer Architecture Evaluation using Commerical Workloads (CAECW)*, February 2003.

[3] Patrick Bohrer, James Peterson, Mootaz Elnozahy, Ram Rajamony, Ahmed Gheith, Ron Rockhold, Charles Lefurgy, Hazim Shafi, Tarun Nakra, Rick Simpson, Evan Speight, Kartik Sudeep, Eric Van Hensbergen, and Lixin Zhang. Mambo: a full system simulator for the powerpc architecture. *SIGMETRICS Perform. Eval. Rev.*, 31(4):8–12, 2004.

[4] Derek Chiou, Huzefa Sanjeliwala, Dam Sunwoo, John Zheng Xu, and Nikhil Patil. FGPA-based Fast, Cycle-Accurate, Full-System Simulators. In *Proceedings of the second Workshop on Architecture Research using FPGA Platforms, held in conjunction with HPCA-12, Austin, TX*, February 2006.

[5] Peter S. Magnusson et al. Simics: A Full System Simulation Platform. In *IEEE Computer*, pages 50–58, February 2002.

[6] Kevin P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux J.*, 1996(29es):7, 1996.

[7] Njuguna Njoroge, Jared Casper, Sewook Wee, Teslyar Yuriy, Daxia Ge, Christos Kozyrakis, , and Kunle Olukotun. Atlas: A chip-multiprocessor with transactional memory support. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE), Nice, France, April 2007*, pages ??–??, 2007.

[8] David Patterson, Arvind, Krste Asanović, Derek Chiou, James C. Hoe, Christoforos Kozyrakis, Shih-Lien Lu, Jan Rabaey, and John Wawrzynek. RAMP: Research Accelerator for Multiple Processors. In *Proceedings of Hot Chips 18, Palo Alto, CA*, August 2006.

VII. Appendix A: Implementation Specification

A. Abstract Type Definitions

The bit widths of the various interfaces and messages are presented for a baseline configuration of RAMP White.

Type	Size	Description
paddr_t	32	Processor address - initially defined to be 32bits to directly support the 32bit processors selected for the system
data_t	64 to variable	Data word used in the system
dmask	8	One-hot byte-enable for memory word writing
mem_cmd	1	Read_write command for MC
msg_t	4	Network message type
gaddr_t	32	Global address - Node based address to allow for segmentation of the address space into multiple regions and allow support large memories
node_t	10	Node identifier allows addressing of 1024 processing elements
net_tag_t	4	Network message tag for matching
net_msg_size_t	2	Size of the network message payload
node_msg_t	variable	Component message with variable length to support variable data sizes

TABLE VI
GENERAL ABSTRACT TYPE DEFINITIONS

B. Processor

1) PowerPC: The PPC405 implements a IBM Core-Connect PLB interface to connect to the data and instruction cache controllers on the core. The PPC405 uses non-coherent I/D caches that are only accessible from inside the processor. The lack of a snoop port prevents the direct usage of the hard I/D caches.

2) Leon 3: The Leon3 uses a ARM AMBA interface as its bus interface. There is a unified bus interface for instruction and data requests that interfaces with the MMU in the core. In addition, the core supports a AMBA slave interface that is used to implement snooping for cache coherency purposes.

Snooping support in the Leon3 is limited to a simple invalidation protocol coupled with a write-through cache. When snooping support is enabled, the data-cache is synthesized as a dual-ported Block RAM and the snooping logic through the AMBA slave interface is enabled. Eventually, we will want a coherent write-back L2 cache for Leon3 (coupled with a write-through L1). As of Jan07, there is no support for both a MMU-enabled core and snooping due to the virtual tagging used in the core. This is expected to change soon with the addition of physical tags and the porting effort to support SMP Linux in the near future by Gaisler Research.

When using the Leon3, we can see the bus adapter would implement a AMBA AHB slave interface (to service processor initiated requests) and a AMBA AHB master interface (to implement a snoop channel into the L1 caches). Externally, the bus adapter presents a simple two unidirectional channels that simply transfer component messages over the channel.

C. Memory Controller (MC)

The memory controller provides a consistent interface to the physical memory attached to the node, independant of the actual implementation technology supported. As it is necessary to support both the XUP as well as the BEE2 board, both DDR and DDR2 memory must be supported with a common interface to presented externally. With the BEE2 board, 4 DDR2 DIMMs are connected to a memory controller @ 200Mhz on a single FPGA. With the XUP board, 1 DDR DIMM is connected to a memory controller @ 100Mhz on a single FPGA.

Name	Direction	Description
sys_signals	out	Power Management and CPU Control signals
core_clocks	in	Core Clock Management (plb,core_clk,JTAG,timer,clkgen, etc)
rst_req	out	Reset requests from processor (chip,core,system)
rst_state	in	Reset status of platform (chip, core, system)
iplbmi	in	Instruction PLB master input protocol, 64bit data
iplbmo	out	Instruction PLB master output protocol, 30bit addr
dplbmi	in	Data PLB master input protocol, 64bit data
dplbmo	out	Data PLB master output protocol, 32bit addr, 64bit data
iocmi	in	Instruction scratchpad memory input protocol
iocmo	out	Instruction scratchpad memory output protocol
docmi	in	Data scratchpad memory input protocol
docmo	out	Data scratchpad memory output protocol
dcrmi	in	Device control register master input protocol
dcrmo	out	Device control register master output protocol
irqi	in	Interrupt inputs
jtagi	in	JTAG input protocol
jtago	out	JTAG output protocol
dbgi	in	Debug input protocol
dbgo	out	Debug output protocol
trci	in	Debug trace input protocol
trco	out	Debug trace output protocol

TABLE VII
PPC405 PORT DEFINITIONS

Name	Direction	Description
clk	in	Core clock
rstn	in	Core reset
ahbi	in	AMBA master-in protocol
ahbo	out	AMBA master-out protocol
ahbsi	in	AMBA slave-in protocol
ahbso	out	AMBA slave-out protocol
irqi	in	Processor interrupt inputs
irqo	out	Processor Interrupt request
dbgi	in	Debug channel input
dbgo	out	Debug channel output

TABLE VIII
LEON3 PORT DEFINITIONS

Name	Type	Direction	Description
ahbsi	ahbsi_t	proc to adapter	AHB slave inputs from processor for requests
ahbso	ahbso_t	adapter to proc	AHB slave outputs to processor for replies
ahbmi	ahbmi_t	proc to adapter	AHB master inputs from processor snoop for replies
ahbmo	ahbmo_t	adapter to proc	AHB master outputs to processor snoop for requests
msg_in	node_msg_t	cache to adapter	Component message input from downstream cache or IU
msg_out	node_msg_t	adapter to cache	Component message output to downstream cache or IU

TABLE IX
PROCESSOR BUS ADAPTER