Extending Open Core Protocol to Support System-Level Cache Coherence

Konstantinos Aisopos
Dept. of Electrical Engineering
Princeton University
Princeton, NJ 08544
kaisopos@princeton.edu

Chien-Chun Chou
Sonics Inc.
Milpitas, CA 95035
joechou@sonicsinc.com

Li-Shiuan Peh
Dept. of Electrical Engineering
Princeton University
Princeton, NJ 08544
peh@princeton.edu

ABSTRACT

Open Core Protocol (OCP) is a standard on-chip core interface specification. The current release is flexible and configurable to support the communication needs of a wide range of Intellectual Property cores, and is now in widespread use. However, it does not support system-level coherence. This paper summarizes an effort within the OCP-IP cache coherence working group on incorporating cache coherence extensions into OCP, which is expected to have strong impact on the MPSoC industry. In this paper, we propose a backward-compatible coherent Open Core Protocol interface and discuss the design challenges and implications introduced. This interface is flexible and can support a range of coherence protocols and schemes: we show how it can specify a snoopy bus-based scheme as well as a directory-based scheme. The correctness of the specification and models was verified using NuSMV, via exploring the entire state space for the two basic coherence schemes.

Categories and Subject Descriptors

B.4.0 [**Hardware**]: Input/Output and Data Communications—*General*

General Terms

Design, Standardization

1. INTRODUCTION

Open Core Protocol (OCP) [7] [13] is a common standard for Intellectual Property (IP) core interfaces or sockets, introduced by the OCP International Partnership (OCP-IP) organization in 2000. OCP facilitates IP core plug-and-play and simplifies reuse, by decoupling the cores from the Network-on-Chip (NoC) and from one another, using a clearly-specified core interface protocol. This interface allows IP core developers to focus on core design, without having to know any details about the System-on-Chip (SoC) that the core might eventually be used in. Since OCP became available to the design community, widespread adoption has occurred, and it is gradually becoming the universal on-chip socket standard for diverse SoC architectures and cores. About 150 companies, including MIPS, Toshiba, nVIDIA, Siemens, and Nokia, have adopted OCP for their system designs. The evolution of OCP is industry-driven, meaning that companies provide constructive feedback by requesting future OCP releases to support specific features, allowing specification updates to reflect technology advancements. This ability to support the continuous evolution of market, while maintaining backwards compatibility, is OCP-IP's greatest strength.

Recently, numerous academic publications pointed out the benefits of hardware coherence in MPSoCs [9] [10] [4], citing reasons such as enhanced performance versus software approaches, faster time-to-market, flexibility, and ease of programming. Numerous cache-coherent multi-core MPSoC products, using a variety of embedded processor cores, have been released: Freescale with PowerPC cores, PMC-Sierra, Cavium and Broadcom with MIPS64 cores, and ARM with ARM MPCores. However, OCP 2.2, as well as other core interfaces for MPSoCs (i.e., AMBA 3.0 AXI [3] and VSI 2.0 [12]), do not support cache coherence, leading to IP providers urgently pressing for OCP-IP to update the protocol specification [5]. Keeping the caches coherent requires the interface to support extra signals to generate/receive coherence messages for invalidating copies or passing cache line ownership between cores. At the same time, this interface should be backward-compatible with previous OCP releases and satisfy on-going industrial designs. It also needs to be flexible to support any specific coherence protocol, even multiple coherence protocols within the same SoC (Section 4.4). In addition, it must be adaptable to cache systems of different configurations. All these features have to be supported while guaranteeing correctness and without sacrificing performance, thus enabling the system architect to apply regular performance optimizations like 3-way communication and write back without response (Sections 4.2 and 4.3 respectively).

In this paper, we propose coherence extensions to support an OCP coherent interface, and walk through the trade-offs and experiences of upgrading the OCP protocol. The structure of the paper is as follows: Section 2 gives a brief background about OCP, Section 3 introduces the proposed extended OCP specification, and Section 4 discusses the design challenges. Section 5 describes the verification methodology and, finally, Section 6 concludes the paper.

2. OCP BACKGROUND

In an OCP system, communicating components (e.g., processors, memory modules, and I/O devices) need a wrapper which implements the Open Core Protocol interface [13]. This interface enforces a point-to-point unidirectional communication: when two components are communicating, the one that utilizes a "master" OCP port sends to the other that utilizes a "slave" OCP port. This section will focus on dataflow communication, though OCP supports optional sideband and test signals to handle interrupts, status flags, errors, control/status inputs, and JTAG signaling between master and slave.

Master-slave model. A master generates requests and injects them into the NoC, while a slave receives these requests, serves them, and responds to the master. To implement multidirectional communication, two components should instantiate both ports: a master port to send requests and a slave port to receive requests. There are four basic OCP requests: *read*, *write*, *read-exclusive* and *broadcast*. These requests enable masters to communicate data to slaves via the shared memory address space: every request is accompanied by an OCP memory address, while a predefined mapping of the address space to slaves indicates to which slave to route to. However, the protocol does not require the response to indicate the master, as it is automatically identified by the NoC. *Read* requests the NoC to return the data corresponding to the provided OCP memory address. *Write* requests the NoC to write data to the OCP memory address. *Read-exclusive* is a synchronization primitive, used for atomic read-modify-write accesses. A "lazy synchronization" scheme, where the shared resource is not blocked between read and write accesses, is also available in later revisions of the OCP specification [7], via two additional requests: *read-linked* and *write-conditional*. *Broadcast* is similar to *write*, but the request can be routed to all slave OCP ports that have been instantiated in the system.

Signals. Figure 1 depicts the signals required for this communication, when a blocking hand-shake protocol is used between a master and a slave. When master generates a *write/broadcast* request, *Write Data* accompanies the *Request*. For *read* and *read-exclusive* requests, *Read Data* accompanies the *Response*. Because of the protocol being blocking, once the master asserts the signals associated with a request, it has to maintain these values until the slave confirms the reception of the request, through the assertion of the *Accept Request* signal (i.e., SCmdAccept). The transaction is completed upon this confirmation. The request may also be completed at this point, if it is marked as "posted". A *write/broadcast* request is marked as "posted" when no response is required to acknowledge its reception by the slave. In contrast, a *read/read-exclusive* request should always be "non-posted", because a response is required to provide the read data. Note that the response channel is also using a blocking hand-shake protocol, thus the master has to assert the *Accept Response* signal (i.e., MRespAccept) to confirm the reception of the response.

A master may have multiple outstanding requests for different cache lines, but each will create a separate record of a pending response. The stream of pending responses should be returned in-order, to keep the protocol simple[1].

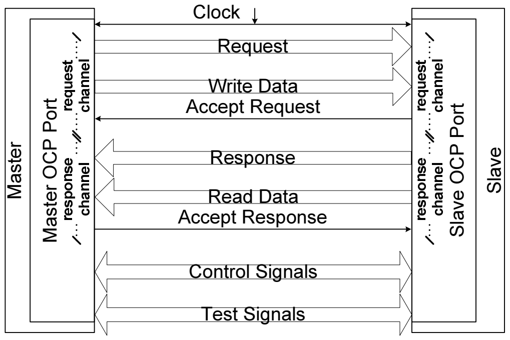

Figure 1: OCP interface [13].

[1]OCP supports out-of-order responses through the usage of multiple threads, or multiple tags within a thread. Cores are free to re-order requests and responses belonging to distinct threads, or belonging to distinct tags within a thread. However, this approach significantly increases complexity and can lead to deadlocks in a cache coherent system. In this paper, we are focusing on single-threaded OCP interfaces without tags.

3. PROPOSED COHERENCE EXTENSIONS

3.1 Coherent Wrappers and Their Interfaces

Introducing OCP coherence extensions (OCPce) port. In a coherent OCP system, communicating components need an OCP wrapper preserving all OCP 2.2 requests described in Section 2, which are called "legacy requests", while supporting new "coherent requests". The coherent requests enable the processor cores, which now maintain cache line state information[2] for each cache line, to request state changes from the NoC. The proposed specification introduces a new port with this augmented functionality: the OCP coherence extensions (OCPce) port (shown in top-half of Figure 2). The wrapped components that generate coherent requests (e.g., processor cores, coherent I/O devices, and directory modules) are now called "coherent masters", and the components that receive and serve coherent requests (e.g., memories, directory modules, and chipset modules) are called "coherent slaves". Note that a directory acts as a coherent slave when receiving requests from coherent masters and as a coherent master when sending requests to coherent slaves.

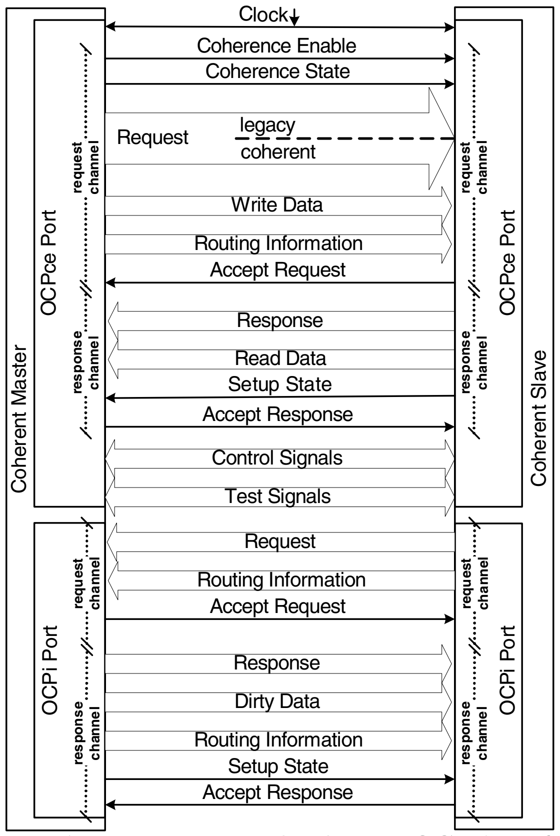

Figure 2: Proposed coherent OCP interface.

The OCPce port includes a boolean signal, *Coherence Enable*, which indicates if the *Request* signal is to be interpreted as legacy or coherent. When *Coherence Enable* is not set, the protocol needs to be identical to the legacy OCP protocol, for backward-compatibility. All introduced coherent requests through the OCPce port's *Request* signal (*upgrade, write_back, read_for_ownership, read_for_shared, invalidate*) are listed in Table 1[3], along with the CPU request (*read/write*) and cache states (*Modified, Shared, In-*

[2]The proposed interface supports the coherence states *Modified, Owned, Exclusive, Shared, Migratory, and Invalid.*

[3]To avoid running into intellectual property issues, generic names were used instead of the ones that will be used in the specification in the future.

request	cache state		outgoing OCPce port request(s)
CPU write	cache hit	Modified	–
		Shared	upgrade
	cache miss	Modified	write_back & read_for_ownership
		Shared	read_for_ownership
		Invalid	read_for_ownership
CPU read	cache hit	Modified	–
		Shared	–
	cache miss	Modified	write_back & read_for_shared
		Shared	read_for_shared
		Invalid	read_for_shared
purge			invalidate

Table 1: Algorithm to generate OCPce port requests

OCPi port request	type of request	outgoing OCPi port response
self intervention	any request	ack & SetupState

system intervention	read_for_ownership
	read_for_shared
	upgrade
		if (dirty copy)
		ack & SetupState & data
		else
		ack & SetupState

	write_back invalidate	ack & SetupState

Table 2: Algorithm to respond to OCPi port requests

valid) that will trigger the injection of these messages, in an MSI protocol. To illustrate the generation of the coherent requests, we will walk through simple examples of CPU requests in a coherent system. More detailed examples are shown in Section 3.2. For instance, upon a CPU write, ownership of the cache line is required. If the line does not exist (cache miss), a *read_for_ownership* request is generated. If the currently cached line is dirty (in *Modified* state), a *write_back* precedes that request. If the line exists (cache hit) but the processor does not currently own the right to modify it (in *Shared* state), then an *upgrade* request is generated. Upon a CPU read request, ownership is not required, thus a *read_for_shared* request is generated only if the line does not exist (cache miss). In addition to CPU reads and writes, it is not unusual for a CPU (or a coherent I/O device) to request to purge outstanding cache line(s) before executing a special operation. This will generate an *invalidate* request.

Responses are always accompanied by a *Setup State* signal, indicating the new cache line state. Another optional[4] signal, *Coherence State*, may be provided by the master in some coherence protocols, to state explicitly the current state of the cache line the request refers to.

Introducing OCP intervention (OCPi) port. The previous paragraphs introduced coherent master's outgoing coherent requests. However, all coherence protocols also require coherent masters to receive messages. These messages are used, for instance, to notify a master when other masters, which cache the same copy, request a cache line state change. Here, there are serious compatibility issues with the legacy OCP port. One of the fundamental principles of the OCP protocol is the request-response dependency: every non-posted request requires a response, and the stream of pending responses should be returned in-order. However, incoming coherence messages are not in response to any request. These non-deterministic notifications cannot be supported by the OCPce port. Thus, masters and slaves that participate in coherence must instantiate a second port, called the OCP intervention (OCPi) port (shown in the bottom-half of Figure 2). This port handles coherent masters' incoming communication to enforce coherence, via coherence messages called "intervention port requests".

When intending to change the coherence state of a cache line, the corresponding coherent master sends an OCPce port request, which is converted and routed to each of the coherent masters caching this cache line, as an intervention port request. Intervention port requests delivered to coherent masters other than the requestor are named "system intervention requests" or "system interventions". The intervention request delivered to the requestor's OCPi port is named "self intervention request" or "self intervention". The proposed specification defines intervention port requests as non-posted, thus they trigger responses acknowledging their receipt.

System interventions. When a coherent master receives a *read_for_ownership* or *upgrade* system intervention, this implies that some other coherent master requests the cache line for ownership. Thus, it should invalidate this cache line. In contrast, when receiving a *read_for_shared* system intervention, it can still cache a *Shared* copy of the cache line. In both cases, if the current copy is in *Modified* (dirty) state, the acknowledgement should also provide the up-to-date copy. *Invalidate* system intervention request has similar functionality with *read_for_ownership*, but does not require the response to include the up-to-date copy. *Write_back* system intervention request has no effect in master's cache line states because, if not in a race condition, a master should not cache a copy that another master is writing back.

Self interventions. Self intervention requests differ from system interventions, because the requestor is the master who received the intervention itself. Note that the self intervention is not a response to the OCPce port request, but a notification from the NoC that the OCPce port request has been received. Self intervention is of greatest importance to serialize the request stream, to be discussed in Section 4.1.

All intervention port responses also provide a *Setup State*, which is the new cache line state for the recipient of the intervention port response. The algorithm for generation of intervention port responses by masters is shown on Table 2. Each coherent master is now equipped with an outgoing OCPce port and an incoming OCPi port. A coherent slave, on the other hand, is equipped with an incoming OCPce port and an outgoing OCPi port, as shown in Figure 2. OCPi port's compliance to the basic OCP model is more relaxed: the absence of backward-compatibility requirement implies that basic principles, like the request-response dependency, may be modified for performance benefits.

3.2 Snoopy Bus-based and Directory-based Protocol Example

This section provides walk-through examples, where two coherent systems are modeled using the proposed OCP specification. It serves to demonstrate the specification's effectiveness, as well as to explain the intricacies. In both examples, three coherent masters are involved: A master that initiates a coherent transaction (initiating coherent master) by requesting to read a cache line and obtain shared ownership (Fig 3-Master A), a coherent master that has no cached copy of this cache line (Fig 3-Master B), and a coherent master having a dirty copy of the cache line (Fig 3-Master C). A coherent slave is also present: a memory module for a snoopy bus-based system, and a module representing both directory and memory for a directory-based system.

Snoopy bus-based system. In a snoop-based coherent system, the following transactions take place (Figure 3):

1 Coherent master A sends a coherent request (*read_for_shared*) on the OCPce port (labeled as OCPce Req), to gain shared ownership on a memory address.

2a This coherent request is delivered to a memory slave (the home of the read address), on its OCPce port (labeled as OCPce Req).

2b Because of the snoopy bus being a broadcast scheme, all coherent masters will receive this request. Thus, the coherent request is turned into a corresponding system intervention request, which is delivered to Master B's and Master C's OCPi ports (labeled as OCPi Req). The coherent request is also delivered to the initiating coherent master's (Master A's) OCPi port (labeled as OCPi Req), as a self intervention request.

3 Upon receiving the *read_for_shared* intervention request, memory is triggered to read the addressed data line.

[4]Some coherence protocols have processor-side hidden transitions (eg. MOESI), meaning that the generated outgoing coherence message does not imply the state of the cache line; in such cases the state of the cache line might become visible to the NoC through this signal.

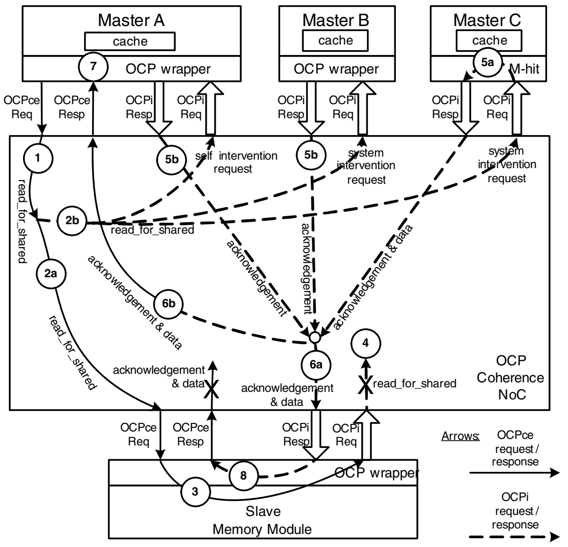

Figure 3: Snoopy Bus-based Example.

After retrieving this data line, the OCP wrapper needs to hold the data, waiting for further instruction coming from its OCPi port (see Step 8), to decide whether to drop or return the data line. Also, an intervention port request is generated, to broadcast the initiating coherent master's request to all coherent masters.

4 However, in a snoopy-bus system this request is not needed, because the request has already been broadcasted in step 2b. So this request, being generated in the snoop phase rather than the data request phase, is dropped by the OCP NoC module.

5a The coherent master, which has the ownership and the only dirty copy of the data for the memory address (Master C), acknowledges the intervention port request and returns the dirty data with coherence state information (*Setup State*). This response relinquishes its exclusive ownership of the memory address.

5b All other coherent masters also acknowledge, by responding to the intervention port request.

6a Once every coherent master has responded, the design requires the NoC to deliver Master C's intervention response with data to the memory slave's OCPi port. This will update the slave.

6b Master C's intervention response with data is also translated into a corresponding OCPce port response with data, and sent back to the original initiating master.

7 Finally, the initiating master receives the latest data response and updates its coherence state for the memory address accordingly.

8 Note that upon reception of the intervention port response, slave returns an OCPce port response with data and coherence state information back to the initiating master. This response is redundant and is dropped by the OCP NoC module. If no coherence master had provided data with the acknowledgement, this would imply that memory has the up-to-date copy and this OCPce port response would be valid.

Directory-based system. In a directory-based coherence environment, the same read for shared ownership transaction can trigger different communication messages (Figure 4):

1 Coherent master A sends a coherent request (*read_for_shared*) on the OCPce port, to gain shared ownership on a memory address. This coherent request is delivered to the OCPce port of a memory/directory slave (the home of the read address).

2 On the memory/directory slave, the directory-based coherence logic realizes, through checking sharing vectors, that it does not have the latest data for the memory address and master C has the latest dirty data.

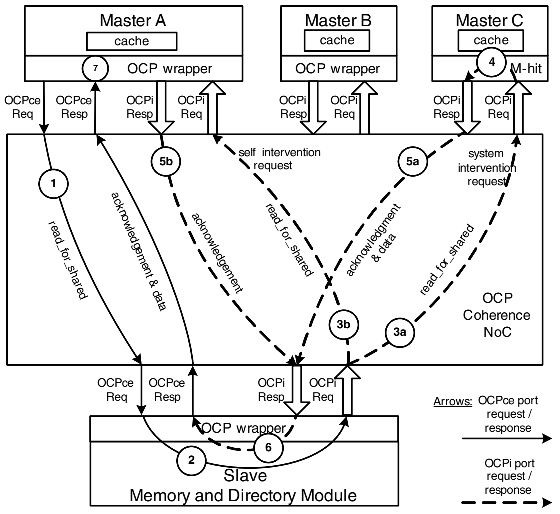

Figure 4: Directory-based Example

3a Consequently, an intervention request is sent from the OCPi port of memory/directory slave to the OCPi port of master C, in order to retrieve the most recently modified data.

3b At the same time, a self intervention request is returned to the initiating master.

4 Master C acknowledges the intervention port request and returns the dirty data, while relinquishing its exclusive ownership of the memory address.

5a The acknowledgement with data is routed back to the memory/directory slave's OCPi port.

5b A self intervention response is generated by the initiating master and is also routed back to memory/directory slave's OCPi port.

6 Upon receiving all responses and the up-to-date copy, both memory and directory are updated. In addition, the memory/directory slave returns an OCPce port response with data and coherence state information (*Setup State*) back to the initiating master.

7 The initiating master receives the latest data response and updates its coherence state for the memory address accordingly.

4. CHALLENGES

4.1 Serialization

Blocking to enforce in-order completion. All coherence schemes have a serialization point to guarantee consistency. In other words, there is always a mechanism, so every coherent master/slave participating in the coherence scheme has the same notion of the request sequence. Also, there is an implicit agreement, among all entities, that every request reaching the serialization point will be served before any other upcoming conflicting request. Once a request reaches the serialization point, requestor's OCPi port takes care of the in-order completion of conflicting requests, via blocking upcoming conflicting requests. Consider the snoopy bus example in Figure 3. Once an OCPce port request shows up on bus (which is the serialization point), all coherent masters receive concurrent intervention requests (step 2b), implying that this request is now served. Upon the reception of self intervention, requestor's OCPi port "blocks" the processing of any upcoming request for conflicting address, until it receives the OCPce port response. Given that the completion of an OCPce port request requires all OCPi ports to acknowledge (step 5), no upcoming conflicting OCPce port request will be completed, because the blocked OCPi port will not acknowledge its corresponding system intervention. In other words, if two conflicting OCPce port requests are issued concurrently, the first to be turned to intervention

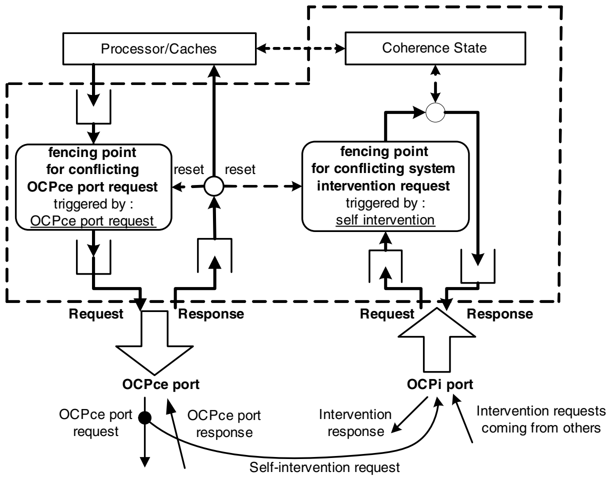

Figure 5: hardware for serialization

port request and show up on bus will block the second; once the first one is completed, the requestor's OCPi port will unblock and the second system intervention port request will be acknowledged, enabling the generation of the second OCPce port response.

Hardware implementation. Figure 5 depicts a hardware implementation of the serialization mechanism in the coherent master: both OCPce port and OCPi port have incoming and outgoing queues, which serve as temporal buffers. Once a coherent request is injected from the OCPce port's outgoing queue into the NoC, the coherent master cannot generate any more OCPce port requests for conflicting addresses. These requests will be blocked in the OCPce port's outgoing queue (denoted as "fencing point for conflicting OCPce port request" in Figure 5), until OCPce port receives a response for the initial request. However, OCPi port can receive, process, and respond to system interventions for conflicting addresses: this will not cause correctness issues until the initial OCPce port request reaches its serialization point in the coherent system. Once that happens, the coherent master will receive a self intervention and will not process any more system interventions for conflicting addresses: no system intervention for a conflicting address will cross the "fencing point for conflicting system intervention request", until the initial request's OCPce port response is received and releases these pending requests.

4.2 Message Routing

Master identification. In an OCP system, masters communicate data to slaves via the shared memory address space: every request is accompanied by an OCP memory address, indicating the slave the request is routed to. The introduction of OCPi port complicates this simple model. When considering non-broadcasting coherence schemes, the identification of masters cannot be implied by the memory address for intervention requests. A typical example is the directory-based scheme: directory should be able to send intervention requests to sharers, for memory addresses they cache but are mapped in the directory's home memory. The NoC should be able to use information other than memory address to route an intervention request to a coherent master. Consequently, there is a need to uniquely identify the masters by adding the *Routing Information* signal to the intervention port request (Figure 2, from the coherent slave's OCPi port request channel to the coherent master's OCPi port request channel), indicating the recipient of the message.

3-way communication. A typical optimization that a system architect applies to non-broadcasting coherence schemes is 3-way communication (or 3-party transaction). 3-way communication enables a coherent master, who is forced to write back a dirty copy due to a system intervention, to forward the intervention response directly to the requestor. In other words, the intervention port response is routed from the cache line owner's OCPi port to the response channel of the requestor's OCPce port. This transaction requires the *Routing Information* signal in the OCPi port request channel to indicate both "who to route to" and "whom to forward to". In addition, there should be a *Routing Information* signal at the intervention port response, indicating "who is the recipient". The *Routing Information* signal should be included in the OCPce port request as well, to indicate "to whom to respond", if 3-way communication is used. It is notable that OCP specification has several generic signals left open for the designer to specify (e.g., MReqInfo, SRespInfo). So, alternatively, the *Routing Information* signals do not need to be included in the proposed specification, and the system designer could map the needed routing information to generic signals. The motivation to include them in the basic interface is that they are expected to be heavily used in future systems.

4.3 Posted Write Back

Many system architects design *write_back* request as posted (doesn't require response). However, such an implementation may cause race conditions in an OCP system: suppose that coherent master A has a *Modified* copy and is generating a *write_back* OCPce port request while, at the same time, coherent master B is generating a *read_for_ownership* OCPce port request to get the copy. If coherent master A's *write_back* request completes upon injection into the NoC and the status of the cache line is changed to *Invalid*, consistency issues may arise. Suppose that coherent master B's *read_for_ownership* OCPce port request reaches the serialization point first; it will be unable to get the up-to-date value of the copy, because coherent master A has marked its copy as *Invalid* and the copy that resides in memory is stale. There are two ways to overcome this race condition: the "complex arbiter" solution and the "data from OCPi port" solution.

In the "complex arbiter" model, NoC embeds a coherence manager, which is intelligent enough to detect race conditions: this is done by memorizing coherent master B's *read_for_ownership* OCPce port request and dropping the stale memory's response when detecting the conflicting *write_back* OCPce port request from coherent master A. After *write_back* completes, coherence manager re-sends a read request to memory, so as to get the up-to-date copy. The "data from the OCPi port" model implies that coherent master A is not sending any data when generating a *write_back* OCPce port request. The data will be provided by the OCPi port, once the coherent master responds to the self intervention. At that point the request has already reached the serialization point and data can be safely invalidated. The choice of either scenario is a typical system architecture trade-off: if the designer opts for a simpler NoC, then the interface will be more complex, because data may be provided from different ports for different OCPce port requests. On the other hand, if the NoC integrates hardware to detect race conditions, then the protocol can be as simple as a posted *write_back* from the coherent master's point of view.

4.4 Flexibility

One of the most critical requirements for MPSoC platform design is flexibility; OCP has been serving its purpose well by enabling a wide range of IP cores that have different communication needs to communicate through a highly configurable interface protocol. IP cores that have different data word widths, endianness, and address widths may co-exist in a system when interfacing with OCP.

The flexibility characteristics of OCP should be maintained and enhanced when extending the specification to support cache coherence: coherent masters may have different cache-line sizes, and even use different coherence protocols: MOESI, MESI, MOSI, or MSI. The proposed coherent interface supports different cache line widths, while the NoC is responsible for merging multiple cache lines, for coherence purposes, so as to match the system's maximum cache line width. There's also support for all existing invalidate-based coherence protocols. Even when masters have different coherence protocols, the specification enables the NoC to apply various algorithms to efficiently maintain coherence [8] [10].

On the other hand, a variety of modules can be plugged in an OCP coherent system using the extended interface: non-coherent/coherent cores with/without caches. Whenever accessing coherent regions of the memory, NoC monitors the requestor and raises an exception if it is not coherent: thus not privileged to access this area. Note that although a non-coherent (without OCPi port) master cannot access a coherent region, a coherent master can access a non-coherent region, because it can generate all the legacy requests.

5. FORMAL VERIFICATION

Model checking is a formal verification method, verifying systems' functionality by exploring the entire state space [6]. In order to verify an OCP coherent system, the communicating components (i.e., coherent masters, slaves, and directory module) are modeled as FSMs. The interconnection medium (i.e., bus/NoC) is modeled as a non-deterministic multiport "black box", receiving requests / responses from its ports, randomly reordering them, and delivering them to their destination. This verification approach does not capture timing information and transaction phases, but works as a high-level protocol checker, that can point out race conditions and protocol bugs. A widespread model checking tool, NuSMV [1], that was developed by ITC-IRST and CMU, is used to model the communicating components and the NoC. NuSMV is a structural language similar to HDLs.

Abstractions. Because of the huge state space, certain abstractions need to be adopted to verify the design in a reasonable runtime: considering the master's model, processor cores non-deterministically decide to read or write, but there is no need to read/write "real" data. The actual data value may be a boolean variable marked as "up-to-date" or "stale". All requests are for the same memory address; non conflicting requests can not cause race conditions. Thus, single entry caches and a single entry memory is modeled, representing the highest shared level of the memory hierarchy. Finally, three processor cores are considered an adequate number for coherence verification purposes. Even for such a simplified system (1K lines of code), the full exploration of state space takes about four days.

Properties verified. Successful verification implies that a set of properties, written in computational tree logic (CTL) [2], are satisfied, while testing all reachable states. Four groups of properties have been validated: 1) correct responses/requests and transitive states: while a response is pending, the cache line is in a "transitive" state. This transitive state should be consistent with the instruction that generated the request. Also, a self intervention request should only be received when there's a pending response. The OCPce port response should be received after the self intervention response and be the one implied by the transitive state. 2) Mutual Exclusion: when a coherent master caches a Modified/Exclusive copy, no other master can cache this copy. When a Shared copy is cached, no other master can have a Modified/Exclusive copy. 3) Staleness: responses should only be accompanied by up-to-date data and when the cache line state is not *Invalid*, the copy should be up-to-date. 4) Liveness: every instruction should complete (no deadlock, no livelock).

Schemes verified. The verification of a coherent interface involves selecting a representative subset of coherence protocols and schemes to verify. The proposed specification was verified for the above properties by Sonics Inc. for a MSI directory-based coherence scheme, and MIPS Inc. for a MESI snoopy bus-based coherence scheme [11].

6. CONCLUSIONS

In this paper, an extended backward-compatible Open Core Protocol interface was introduced. To the best of our knowledge, this is the first interface to enable hardware cache coherence in MPSoC designs. We walked through the design challenges of serializing the requests to satisfy the sequential consistency model and identifying cores to enable the routing of requests. We also explored how the interface enables system architects to apply critical performance optimizations, like 3-way communication and write-backs without responses. This interface enables cores with caches of different configurations, managed by different coherence protocols, to co-exist in the same system. At the same time, various coherence schemes can be adapted. Model checking results verified protocol correctness for the entire state space of a snoopy bus-based scheme and a directory-based scheme.

7. ACKNOWLEDGMENTS

The authors would like to thank the OCP-IP Specification Working Group, David Lau, Yasuhiko Kurosawa, Jay Jayasimha, Drew Wingard, Wolf-Dietrich Weber, Steve Krueger, and Sanjay Vishin for their guidance and contributions to the proposed OCP coherence extensions. We also wish to thank the Computer Architecture Reading Group in Princeton University, and especially Carole-Jean Wu, Ilias Tagkopoulos, and Emmanouil Koukoumidis for their valuable comments on this work.

8. REFERENCES

[1] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Proc. Computer Aided Verification*, pages 241–268, Jul 2002.

[2] E. A. Emerson. Temporal and Modal Logic, Handbook of Theoretical Computer Science, volume B. *Elsevier and MIT Press*, 1990.

[3] D. Flynn. Amba: Enabling reusable on-chip designs. *IEEE Micro*, 17(4):20–27, Jul 1997.

[4] M. Loghi, M. Poncino, and L. Benini. Cache coherence tradeoffs in shared-memory MPSoCs. *ACM Transactions on Embedded Computing Systems*, 5(2):383 – 407, May 2006.

[5] I. Mackintosh, OCP Chairman and President. MPSoC and 'The Vision Thing'. *EDA Tech Forum Journal*, 4(3):6–7, Sep 2007.

[6] K. L. McMillan. Symbolic Model Checking. *Kluwer Academic Publishers*, Norwell, MA, USA, 1993.

[7] OCP International Partnership. Open core protocol specification, release 2.2, Jan 2007.

[8] T. Suh, D. M. Blough, and H.-H. S. Lee. Supporting cache coherence in heterogeneous multiprocessor systems. In *Proc. Design, Automation and Test in Europe Conference and Exhibition*, volume 2, pages 1150–1155, Feb 2004.

[9] T. Suh, D. Kim, and H. S. Lee. Cache coherence support for non-shared bus architecture on heterogeneous MPSoCs. In *Proc. Design Automation Conf.*, pages 553 – 558, Jun 2005.

[10] T. Suh, H.-H. S. Lee, and D. M. Blough. Integrating cache coherence protocols for heterogeneous multiprocessor systems, part 1. *IEEE Micro*, 24(4):33–41, Jul 2004.

[11] S. Vishin. Design and verification of a cache coherence protocol for embedded Systems-on-Chip. *MIPS Inc.*, internal documentation.

[12] VSI Alliance. Virtual Component Interface Standard Version 2 (OCB 2 2.0), Apr 2001.

[13] W.-D. Weber. Enabling reuse via an IP core-centric communications protocol: Open Core Protocol. In *Proc. IP 2000 System-on-Chip Conference*, pages 217–224, Mar 2000.