A Memory System Design Framework: Creating Smart Memories

Amin Firoozshahian*, Alex Solomatnikov*
Hicamp Systems, Inc.
{aminf13, solomatnikov}@gmail.com

Ofer Shacham, Zain Asgar, Stephen Richardson, Christos Kozyrakis, Mark Horowitz
Stanford University
{shacham, zasgar, steveri, kozyraki, horowitz}@stanford.edu

ABSTRACT

As CPU cores become building blocks, we see a great expansion in the types of on-chip memory systems proposed for CMPs. Unfortunately, designing the cache and protocol controllers to support these memory systems is complex, and their concurrency and latency characteristics significantly affect the performance of any CMP. To address this problem, this paper presents a microarchitecture framework for cache and protocol controllers, which can aid in generating the RTL for new memory systems. The framework consists of three pipelined engines—request-tracking, state-manipulation, and data movement—which are programmed to implement a higher-level memory model. This approach simplifies the design and verification of CMP systems by decomposing the memory model into sequences of state and data manipulations. Moreover, implementing the framework itself produces a polymorphic memory system.

To validate the approach, we implemented a scalable, flexible CMP in silicon. The memory system was then programmed to support three disparate memory models—cache coherent shared memory, streams and transactional memory. Measured overheads of this approach seem promising. Our system generates controllers with performance overheads of less than 20% compared to an ideal controller with zero internal latency. Even the overhead of directly implementing a fully programmable controller was modest. While it did double the controller's area, the amortized effective area in the system grew by roughly 7%.

Categories and Subject Descriptors

B.3.m [**Memory Structures**]: Miscellaneous

B.5.1 [**RTL Implementation**]: Design – *Control design.*

General Terms

Performance, Design, Verification

Keywords

Multi-core processors, Memory Systems, Reconfigurable Architecture, Memory Access Protocol, Protocol Controller, Cache Coherence, Stream Programming, Transactional Memory.

ISCA'09, June 20–24, 2009, Austin, Texas, USA.
Copyright 2009 ACM 978-60558-256-0/09/06…$5.00.

1. INTRODUCTION

With the advance to the multi-core era and replication of processor cores on a single die, the surrounding "un-core" logic, such as cache, memory controllers, and network interfaces, is growing in importance. In particular, implementing the necessary data sharing and communication protocols for multi-core processors involves handling a large amount of transient state that is not necessarily visible to the high-level protocol. As a result, the design of controllers that implement such protocols is usually complex, because they are part of the large distributed on-chip memory system and must provide global guarantees on consistency, ordering and forward progress. Moreover, since the system's programming model defines the data sharing and communication semantics and the realization of this model is often tailored to a specific system, the implementation differs from one system to another, preventing controllers from being reused.

To address these issues, this paper proposes a microarchitectural framework for the design of on-chip memory systems and, in particular, protocol controllers. This approach is based on breaking down the functionality of the on-chip memory system into a set of basic operations and providing the necessary means for combining and sequencing these operations. The system is programmed to perform protocol actions in the memories and controllers by appropriately combining these basic operations. Having such a framework in place provides multiple benefits: first, it reduces the design time for creating and implementing controllers for a specific protocol, by converting the hardware design problem into a software programming problem. Programmed values in the memory system are converted into constants and propagated into the logic at synthesis time, facilitating generation of controllers for multiple protocols. Second, it enables run-time alteration of the memory system behavior to fix or patch design errors after fabrication, even after deployment of the system, as well as enabling run-time tuning of the memory system performance. Third, a direct implementation of the RTL for the whole framework along with the "program storage" effectively creates a flexible, polymorphic memory system that can support a broad class of memory models. Last but not least, this framework provides a platform for developing future tools such as protocol checkers or optimizers for increasing the verification coverage and performance of the generated system.

To validate this approach, we directly implemented our

*This work was accomplished while the authors were attending Stanford University framework as a configurable controller along with eight Tensilica [15][16] processors to create a polymorphic CMP, which we fabricated using ST 90nm CMOS technology. Before the chip was taped out, we validated the resulting RTL design (and associated memory system configurations) to ensure that the system would correctly implement three distinct memory models: cache-coherence shared memory, streaming and transactional memory. The overhead for using this design approach was small. For all three memory systems, performance with the generated protocol controller is within 20% of the performance of an idealized controller, where internal protocol operations were assumed to take place at zero latency. The area overhead of directly implementing the flexible controller was modest, less than twice the area of a controller optimized at design time for a specific protocol. While this overhead might seem large, the resulting protocol controller area was only 14% of the core area.

The next section reviews some of the previous work on memory system design and programmable protocol controllers, focusing on the work relevant to Smart Memories project [8] that we build upon. Section 3 then presents our memory system architecture, and Section 4 describes the overall organization and microarchitecture of the protocol controller that makes everything work. Section 5 then maps a number of distinct memory models onto this architecture, and Section 6 evaluates the performance and area overhead of this approach to controller design.

2. BACKGROUND

All memory access protocols at the hardware implementation level perform a limited set of operations: they move data from one physical location to another; they associate and update state information with data that guides the data movement operations, and they preserve necessary ordering between different operations in order to conform to the high-level protocol properties. Controllers, as the primary engines executing protocol actions in the memory system, are responsible for completing these actions, and have been extensively studied in the literature.

2.1 Controller Design

There have been many proposals for designing high-performance, low-latency protocol controllers, especially for coherence protocols, and several micro-architectural techniques for both hard-wired and programmable controllers have been developed [24][25][26]. Particularly, programmable protocol controllers have been the subject of extensive research and have successfully been implemented in many academic and industrial projects: FLASH [13], Typhoon [12], Impulse [18], S3.mp [19] and Alewife [20] are just a few examples. While our philosophy toward programmable controllers is similar to this previous work, and we leverage some of their approach (event driven execution, dispatch on message types, etc.), our goal is to create a programmable memory hierarchy all the way down to the processor’s first level cache interface. Hence protocol controllers in our system have to sustain a very high throughput and are very latency sensitive. Therefore using a general-purpose processor for executing protocol actions (the approach of the MAGIC chip in FLASH, or the NP in Typhoon) would not be adequate.

It is not surprising that commercial multi-core processors primarily use hardwired solutions for control of their on-chip memory resources, although these solutions often include the same kind of message driven execution seen in the programmable machines. For instance, Niagara's memory controller uses one or two coded packets (a kind of microcode) sent across a crossbar to manage the transaction according the packet's code [29].

In addition, in many controllers complex operations are broken into many smaller operations. In the controller for AMD's Opteron processor, a single read transaction generated du to a cache miss might result in thirteen messages from three different message classes: two Request messages, three Probe messages, and eight Response messages [28]. Similarly, the IBM Cell processor's Memory Flow Controller (MFC) transfers data to and from each compute element by way of a set of primitive commands [27].

Another old controller idea that we use is the notion of patchable microcode. The use of microcode for patching and detecting design errors in the processor and memory system has also been proposed both in industry and academia and is widely used in processors [21][22][23]. Creating a hardware framework for the on-chip memory system that we can microcode, allows us to implement the protocols in software, which also allows to use software patches to fix memory system errors after deployment.

2.2 Protocol Design

There has also been a lot of work in the literature in creating novel cache/local memory systems. Discussions about message passing and shared memory protocols have given way to proposed new programming models such as streams and Transactional Memory (TM). Streaming systems such as Imagine [3] and IBM Cell [4] share some characteristics with message passing machines, in that all communication is explicit, and also share some traits of shared memory machines, in that they generally have a shared address space and use high performance, low latency networking to connect the processors to each other and to the memory. However, the total local memory is often small, so these machines implement their entire local memory in fast on-chip SRAM, and forgo building a cache hierarchy entirely. To fill and spill this local memory, implementations often contain sophisticated DMA engines that support gather/scatter operations as part of the memory hierarchy [3][4].

On the cache coherent front, modern shared memory machines are moving to support a larger number of threads to help hide memory latency [1][2], which requires the memory system to sustain and track several memory requests from different threads in order to tolerate long memory access latencies. There has also been a lot of research in extending speculative execution techniques to the memory system. Thread Level Speculative (TLS) systems such as Hydra [5] and Stampede [6] extend conventional memory systems with mechanisms to track and buffer results of speculative computations and to detect logical data dependencies between speculative threads. Most recently Transactional Memory (TM) has generalized and formulated these systems into a transactional programming model [7][10][14]. There have been many proposals for implementing transactional systems, in hardware. Since any implementation must either buffer the speculative values that are written, or the old values that have been overwritten by speculative data, the hardware must store a significant amount of information to track dependencies and it must also support inter-transaction

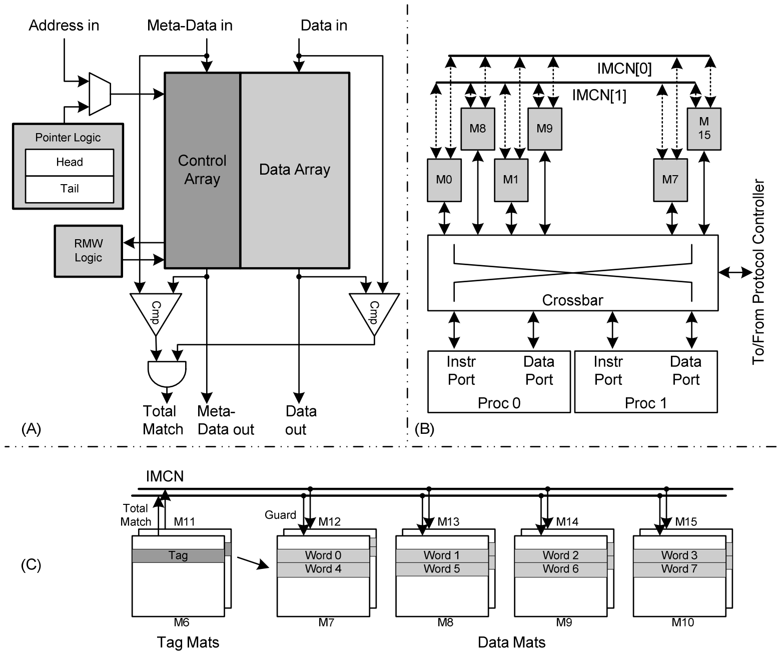


Figure 1. Memory organization of the SM Tile. (A) Block diagram of the memory mat. (B) Tile crossbar and IMCN. (C) Example cache configuration.

communication (such as write-set broadcasting) to commit or flush data. We use all of these memory models to demonstrate the capabilities of our memory control framework.

2.3 Smart Memories

Our work on creating a protocol controller framework builds upon ideas from polymorphic computing, which tries to map different programming models to a malleable hardware substrate. For example, the TRIPS polymorphic architecture can be configured to better leverage instruction, data or thread level parallelism [9]. The Stanford Smart Memories (SM) project took a different approach where they directly mapped a stream machine and a thread-level speculative machine on a reconfigurable hardware substrate [8]. We build on the SM work, which showed how storing a small number of state / meta-data bits and updating them on each access could maintain state information needed for a wide variety of memory models. Unfortunately this work focused on the processor “Tile” and did not explain how to flexibly implement the different protocols that are needed for these different memory models. The approach in this paper addresses this limitation, providing a method of constructing the needed protocol controllers. Since our controllers assume local memories contain state similar to what was described in the Smart Memories system, we review this information next.

As described in [8] the basic unit of the architecture is the Tile. Each Tile contains two processor elements, 16 local memory blocks and a crossbar interconnect to connect memories to processors and the outside world (Figure 1). The memory mats are the basic storage element in the system and are connected through an Inter-Mat Communication Network (IMCN)—a fast path for exchanging memory control and state information.

Memory mats are aggregated (using IMCN) to implement composite storage structures such as instruction and data caches (Figure 1c). Meta-data bits in the tag storage are used to encode state information according to the protocol, such as cache line state and LRU information in shared memory model, or a transaction’s read and write sets in a TM model. When implementing a streaming model, the memory mats are aggregated into addressable scratchpads. Moreover, having head/tail pointers in the memory mats allows them to efficiently implement hardware FIFOs, which can be used to capture producer-consumer locality between processors. It also can simplify some protocol/runtime operations. For example, hardware FIFOs are used to augment cache structures in order to store addresses of a transaction’s write set, which is used at transaction commit time for broadcasting address/data pairs [10].

The protocol controller described in the next section assumes that local memories have the needed meta-data bits to store protocol state and also have simple hardware that can modify the state, if needed, on each access. We also assume that the local memory or processor can create a small number of request types when it needs help from the protocol controller.

3. PROTOCOL CONTROLLER FRAMEWORK

In our attempt to create a design framework for memory systems, we associated meta-data with the local storage and decided to take a RISC-like approach for the protocol controller design: instead of providing complex pre-defined operations, we provided a small number of basic operations and implemented complex data and state manipulations by executing a set of these basic operations. As was mentioned earlier, a general-purpose RISC processor would be too slow, but fortunately only a small number of primitive operations are needed to support all the models that we investigated, and these could be accomplished in a small number of programmable FSM/pipelined engines.

Table 1. Similarities between actions taken by different memory protocols

#	Model	Protocol	Action	Similar to
1	Streaming		DMA block read (main memory to local memory)	5
2			DMA block write (local memory to main memory)	6
3			DMA transfer from a local memory to another	7
4			DMA indexed scatter	10
5	Coherent Shared Memory	Any	Cache refill	1
6		Writeback caches	Write-back (cache spill)	2
7		Invalidation based	Cache to cache transfer	3
8		Invalidation based	Snoop, coherence degrade/invalidate	11, 12
9		Update based	Updating word in destination caches	10
10	HTM	TCC [10]	Commit - updating data in other caches and main memory	4, 9
11		TCC	Conflict detection (lazy) - checking for violation in destination cache upon commit	8
12		LogTM [11]	Conflict detection (eager) - checking for violation upon receiving coherence request	8

This approach works because across many different memory models the functions of all protocol controllers are very similar: at their core all protocol engines track and move data. One can recognize such similarity at two levels: at the high level, many protocol actions that implement a memory model have the same conceptual functionality. Table 1 lists a few of these actions, indicating which other protocol actions they resemble.

At a lower level, the hardware operations that are combined to form the protocol actions are also the same. These primitive operations can be categorized into five different classes:

- Data/State read and write – Accessing data and state storages for performing data transfers, state inquiries and updates, according to the specific protocol action
- Communication – Sending and receiving messages over available communication infrastructure
- Ordering – Guaranteeing a specific order between requests from same processor or different processors, according to the specific protocol or memory consistency model
- Tracking – Keeping track of the outstanding requests in the system so that each request can be completed after corresponding reply is received. This is also necessary for enforcing ordering between different requests
- Association and interpretation of state information – This is the major differentiating factor among memory models; indicates how the state associated with data is interpreted and controls the flow of operation according to the specific protocol

These operations are essentially the RISC instructions, the basic blocks, for composing protocol actions. One can describe the activities occurring in the memory system hardware upon receiving any protocol request/reply message as a composition of the above operations in an appropriate sequence. We implement these operations in two structures, the local memory hardware which is associated with each processor, and the protocol controller that connects a number of local memory hardware units to the network. Having described the local memory earlier, we describe the protocol controller next.

4. PROTOCOL CONTROLLER DESIGN

In our framework, processors and main memory controllers communicate with the protocol controller by sending and receiving request/reply messages. Each request message when received invokes a “subroutine” in the controller that executes a series of basic memory operations. One creates a memory model by defining the set of messages that the protocol controller needs to handle, and then composing the required actions for each message from the basic operations described above.

4.1 Organization

Figure 2 illustrates the internal organization of the protocol controller. The execution core of the controller consists of three major units: Tracking and Serialization (T-Unit) serves as the entry point to the execution core of the controller. It stores and retrieves tracking information of the outstanding memory requests in the appropriate tracking structures. The Miss Status Holding Registers (MSHR) provide storage for cache misses and memory operations that require some form of ordering. It supports a lookup operation based on a request’s address or source processor

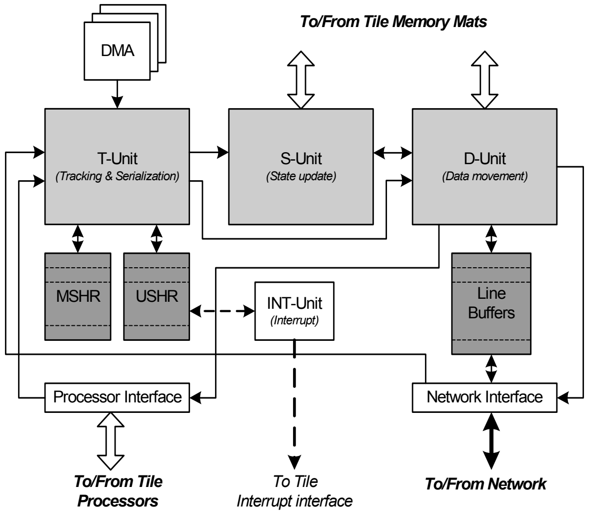
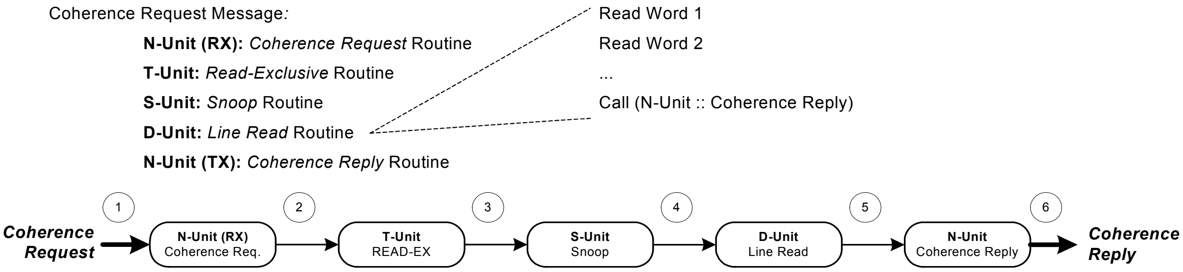


Figure 2. Organization of the protocol controller



Coherence Request Message:
N-Unit (RX): *Coherence Request* Routine
T-Unit: *Read-Exclusive* Routine
S-Unit: *Snoop* Routine
D-Unit: *Line Read* Routine
N-Unit (TX): *Coherence Reply* Routine
Read Word 1
Read Word 2
...
Call (N-Unit :: Coherence Reply)
1
2
3
4
5
6
Coherence Request
N-Unit (RX) Coherence Req.
T-Unit READ-EX
S-Unit Snoop
D-Unit Line Read
N-Unit Coherence Reply
Coherence Reply

Figure 3. Conceptual execution model of the protocol controller

number, which is used to detect conflicting requests and enforce serialization. The Uncached-request Status Holding Registers (USHR), on the other hand, keep the tracking information of requests that do not require such serialization, for instance DMA transfers. Tracking resources are sized according to the expected occupancy of the controller. For example, one can determine appropriate sizes by simulating the controller with an infinite number of tracking registers and statistically determining the minimum required size for a given performance level.

The State Update (S-Unit) performs read, write and manipulation of the state information associated with data blocks, such as cache tags and cache line states. It has a dedicated port to Tile memory mats in order to read and write data and meta-data bits. The Data Movement Engine (D-Unit) provides necessary functions for reading and writing data blocks from memory mats into an internal data buffer (Line Buffer). It also checks and updates meta-data bits that are associated with an individual data word.

Communication primitives are implemented in processor and network interface units. The Processor interface unit (P-Unit) receives and decodes request messages from all processors sharing this controller and passes them to the execution core. The Network interface unit (N-Unit) consists of separate transmitter and receiver logic that composes and decodes messages that are communicated over the system network. It has necessary interfaces to the line buffer to read and write blocks of data that are transmitted or received over the network.

The controller is also equipped with independent DMA channels, which are programmable request generator engines. Each channel is associated with a processor supported by the controller and is programmed by writes into control registers. A dedicated interrupt unit (INT-Unit) can send individual interrupt requests to any processor in the group when necessary. Processors can also generate interrupts for one another by writing into control registers in the INT-Unit.

4.2 Programming

The conceptual programming model of the controller is the execution of a set of subroutines triggered by an input message to the controller. Each subroutine comprises a few basic operations and is executed by one of the internal functional units. After executing the subroutine, each functional unit invokes another subroutine in the next functional unit(s) by passing an appropriate request type to it. Subroutines are chained to one another until processing of the input message is completed. Sequential execution semantic is maintained within each subroutine.

Figure 3 depicts a conceptual execution model in the controller, presenting the steps of processing a coherence request message. The request message invokes a coherence request subroutine in the network receive unit. This subroutine calls the Read-Exclusive subroutine in the T-Unit. After performing necessary serialization actions, this subroutine calls the Snoop subroutine in the S-Unit to snoop cache tags. It is also possible for a functional unit to call two or more subroutines in different units concurrently to parallelize processing of different parts of the same request message. Processing ends after the the N-Unit transmitter sends a coherence reply message to the main memory controller.

4.3 Micro-architecture

The functional units form a macro-level pipeline to process incoming messages. Requests are passed from one functional unit to another after the executing subroutine completes its necessary steps and invokes the next subroutine. Each of the functional units is organized as a shallow pipeline. The pipelining allows the unit to overlap processing steps of different requests internally and further increases the throughput of the controller.

The first stage of each unit's pipeline contains a configuration memory that holds the controlling microcode for the pipeline. This memory is indexed by the type of the request passed to the unit and dictates all the operations that are performed in that unit. At the output of the unit a shallow queue ensures that the pipeline can be completely drained and all the in-flight operations can be completed. An arbiter located in front of each unit decides what request is passed into the unit at each clock cycle.

Figure 4 shows the S-Unit pipeline as an example. Configuration memory is located at the first stage of the pipeline (Access Generator). It generates necessary signals for accessing memory mats in the Tiles. In addition to identifying mats that need to be accessed, these access signals individually control operations on the data and meta-data arrays, as well as the read-modify-write logic that allows the mats to update their own state bits. Up to two mat accesses can be generated in the AG stage. The S-Unit can send a mat access to a single Tile, all Tiles simultaneously, or can send one access to a specific Tile and another access to the remaining Tiles. Two pipeline stages are used for accessing the memory mats which include the roundtrip time over the Tile crossbar. At the end of the pipeline a lookup table serves as the decision-making logic. It analyzes the information retrieved from memory mats and generates necessary requests for next functional unit.

Figure 5 shows the micro-architecture of the data movement engine. It consists of four parallel data pipes, one connected to each Tile. Each data pipe has its own input and output queues. A dispatch unit decodes an incoming request and uses a lookup table to issue necessary operations to each data pipe. For example, a block transfer request is converted into a block read operation on the source pipe and a block write operation on the destination

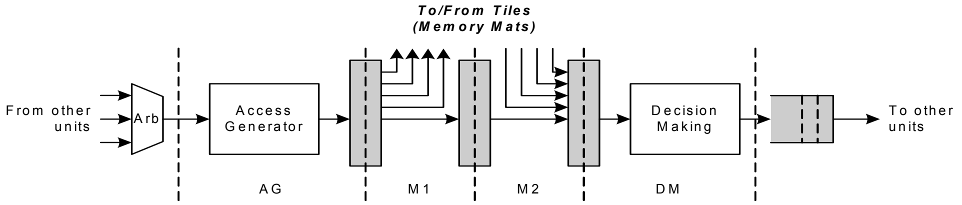


Figure 4. S-Unit pipeline

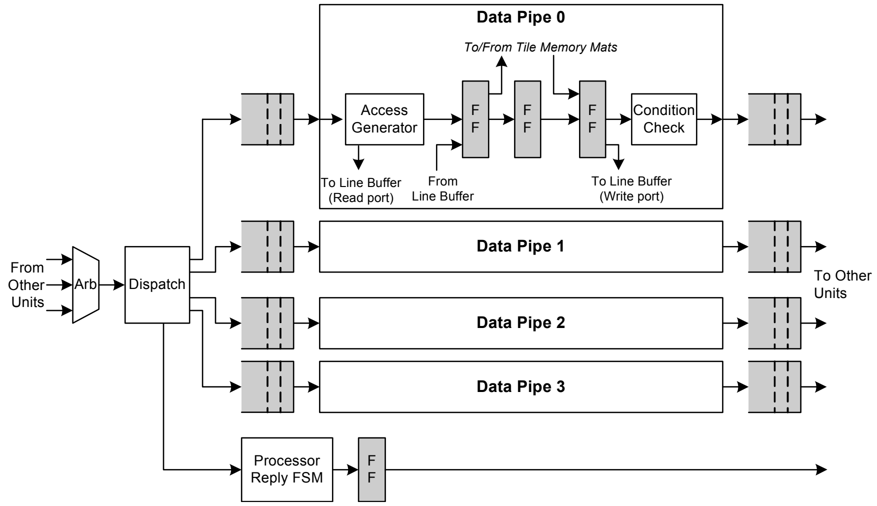


Figure 5. Data movement engine (D-Unit)

pipe. All data pipes are connected to the line buffer and data is staged through the line buffer between the two pipes. A small FSM generates necessary replies for the processors if the data transfer was due to a processor's request.

Each data pipe has four stages and is very similar to the S-Unit: the Access Generator has a configuration memory that generates the necessary memory mat access signals. Condition Check logic is a lookup table that compares the meta-data bits collected from the mats with predefined patterns and generates necessary subroutine calls for subsequent units. Data pipes supports 32-bit (single mat) and 64-bit (double mat) accesses. 64-bit accesses use two adjacent memory mats for faster data transfers.

4.4 Avoiding Deadlocks

Deadlocks are avoided in the system by carefully considering a set of constraints. First, the controller provides some deadlock avoidance guarantees in the hardware. The output queue at the final stage of each functional unit guarantees that the in-flight requests in that pipeline can be completed without needing to stall the functional unit. The arbiter in front of each unit considers necessary buffering space in the output queue and throttles the input requests if there is no buffering space available.

The network interconnect supports multiple virtual channels. Priorities for processing virtual channel messages are adjustable at the network interfaces. However, it is the designer's responsibility to distinguish between protocol request and reply messages [17] and assign them to different virtual channels. Since the connection between the S-Unit and the D-Unit forms a loop inside the controller, there is a potential that the controller can live-lock, where subroutines in the S-Unit and D-Unit invoke each other repeatedly. Once again it is the responsibility of the protocol designer to ensure that the controller is programmed such that all possible subroutine chains eventually terminate. Live-lock avoidance can be guaranteed by simply forbidding subroutine *foo* in, for example, the D-Unit from calling subroutine *bar* in the S-Unit that might possibly invoke *foo*. In the controllers we have implemented, this has not been a difficult constraint to satisfy. An overview of these protocols is described in the next section.

5. MEMORY MODEL IMPLEMENTATION

When implementing a memory protocol, operations are divided between different parts of memory system, namely processor interface logic, local memory mats, protocol controller and main memory controller. These components communicate by sending and receiving messages. Table 2 lists and describes all the messages exchanged between protocol controller and processors as well as main memory controller for our three different memory models. In this section we demonstrate how the protocol controller is programmed to handle a few of these messages.

Figure 6 shows an example of an indexed DMA scatter operation to main memory. First, the DMA channel issues an index read operation that returns the address of the destination memory to it. After identifying the destination, the DMA channel generates line-size requests for transferring a single data element (source address, element size and number of elements are programmed by writing DMA control registers). The T-Unit allocates a tracking register and stores tracking information for the operation and passes it to the D-Unit. The D-Unit extracts the appropriate memory block from the source memory into the line buffer entry and requests the N-Unit to send scatter message to main memory. The N-Unit reads the data from the line buffer entry and sends it over the network to the main memory controller. After data is

Table 2. Request/Reply messages handled by protocol controller (MC=Main Memory Controller)

Model	Source	Message	Description
Shared Memory (MESI coherence)	Processor	Cache Miss	Read/Write miss request from a processor
		Upgrade Miss	Upgrade miss request (request for ownership)
		Prefetch	Prefetch for read or write from a processor
		Cache Control	Invalidate/Writeback a specific cache line
	MC	Coherence Request	Read, Read-Exclusive or Invalidate request for specific cache line
		Refill	Returns cache line data to be refilled
		Upgrade	Returns cache line ownership (no data)
Streaming	Processor	Un-cached Access	Direct access of a memory in another Tile or Quad
	DMA Channel	Index Read	Read of index memory (indexed transfers)
		DMA Gather	Request for gathering data from another Quad or main memory
		DMA Scatter	Request for scattering data to another Quad or main memory
	MC / Another Quad	Gather Reply	Reply for a gather request, contains actual data
		Scatter Reply	Acknowledgement for a previous scatter
		Un-cached Reply	Reply for direct memory access from processor
		Net Gather	Gather request from another Quad's DMA
		Net Scatter	Scatter request from another Quad's DMA
Transactions (TCC [10]])	Processor	Cache Miss	Read/Write miss request from a processor
		FIFO Full	Address FIFO full indicator, overflow occurred
	DMA Channel	FIFO Read	Read store address from FIFO
		Commit Read	Read committed data from source cache
		Commit Write	Write committed data to other caches
	MC	Refill	Returns cache line data to be refilled
		Net Commit	Committed data word from another Quad's transaction

written to the destination address, a scatter reply message confirms completion of the operation. This message is decoded by the network interface unit and is passed to the T-Unit. The T-Unit retrieves the tracking information, releases the tracking register and sends an acknowledgement to the originating DMA channel.

Figure 7 shows another example of servicing a cache miss request from processor. A cache miss is received and decoded by the processor interface unit and is then passed to the tracking unit. The T-Unit looks up the tracking information of outstanding cache misses to serialize the cache miss properly against prior requests. If no collision is found, it allocates an MSHR entry and saves the tracking information of the cache miss. The cache miss is then passed to the S-Unit, which evicts a cache line in the

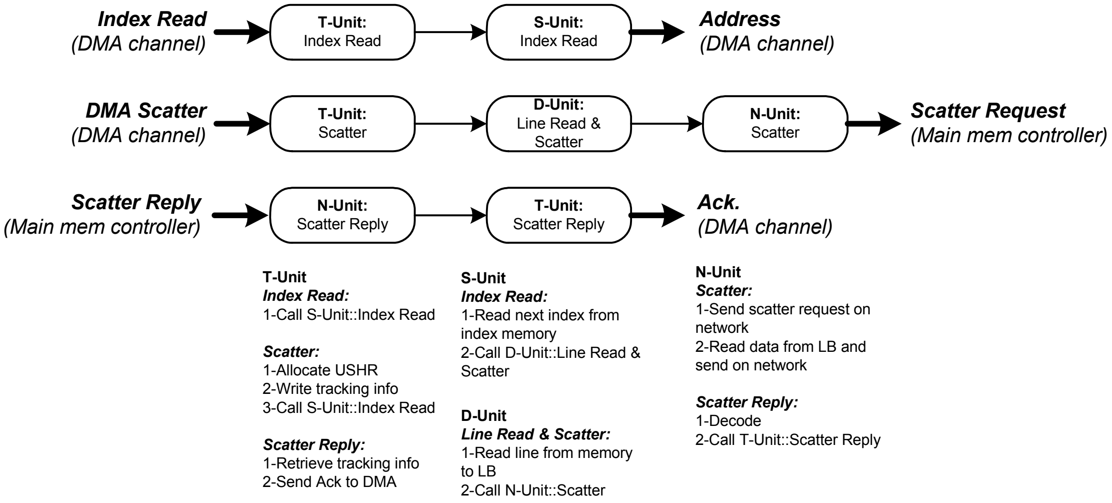
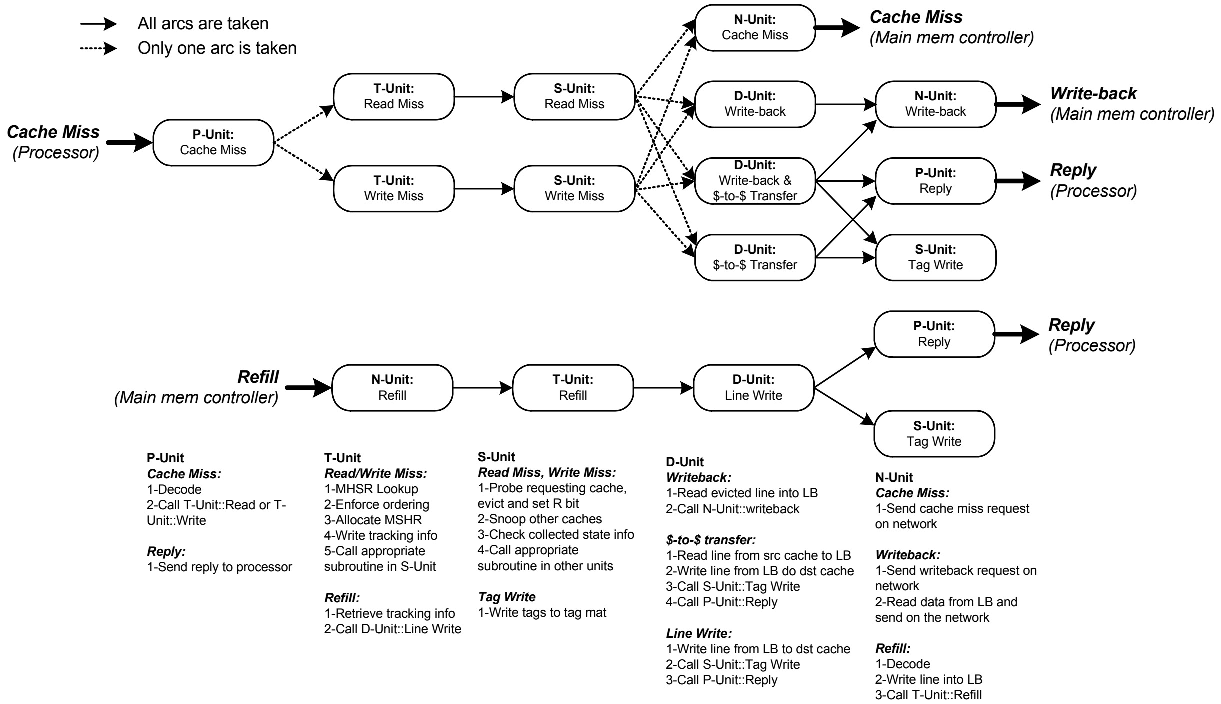


Figure 6. Steps for processing and indexed DMA operation



P-Unit
Cache Miss:
1-Decode
2-Call T-Unit::Read or T-Unit::Write

Reply:
1-Send reply to processor

T-Unit
Read/Write Miss:
1-MHSR Lookup
2-Enforce ordering
3-Allocate MSHR
4-Write tracking info
5-Call appropriate subroutine in S-Unit

Refill:
1-Retrieve tracking info
2-Call D-Unit::Line Write

S-Unit
Read Miss, Write Miss:
1-Probe requesting cache, evict and set R bit
2-Snoop other caches
3-Check collected state info
4-Call appropriate subroutine in other units

Tag Write
1-Write tags to tag mat

D-Unit
Writeback:
1-Read evicted line into LB
2-Call N-Unit::writeback

$-to-$ transfer:
1-Read line from src cache to LB
2-Write line from LB do dst cache
3-Call S-Unit::Tag Write
4-Call P-Unit::Reply

Line Write:
1-Write line from LB to dst cache
2-Call S-Unit::Tag Write
3-Call P-Unit::Reply

N-Unit
Cache Miss:
1-Send cache miss request on network

Writeback:
1-Send writeback request on network
2-Read data from LB and send on the network

Refill:
1-Decode
2-Write line into LB
3-Call T-Unit::Refill

Figure 7. Steps for processing a cache miss request

source cache to open up space for the cache refill. The S-Unit simultaneously snoops the tags of caches in other Tiles to enforce coherence properties and to consider the possibility of a cache-to-cache transfer. The Data movement engine performs the necessary writeback or cache-to-cache transfer based on the decision made in the S-Unit. The Network interface sends the write back request to main memory controller as well as the cache miss request. Refills are handled similarly.

6. EVALUATION

We evaluated the performance overhead of our protocol controller framework by simulating our reconfigurable CMP, which directly implemented it. The processor simulator and software tool chain were supplied by Tensilica [15][16] while the memory system and interconnect simulator was developed using the Xtensa Modeling Protocol (XTMP). In order to evaluate the performance impact of the framework, we back-annotated the memory system simulator with latency numbers extracted from the actual system RTL, and compared the results to simulations where all internal controller operations took zero cycles, but external operation timing (e.g. mat read) remained unchanged. For correctness checking, the RTL was extensively checked using both applications and random stress cases as part of the tape-out flow.

Table 3 lists the benchmark applications for three different models that we used to evaluate system performance. Table 4 describes system parameters used for performance simulations.

Figure 8 shows speedups for three different memory models. For almost all benchmarks, our system shows good performance scaling, achieving at least 50% parallel efficiency. In cache

Table 3. Benchmark applications

Model	App.	Problem Size	Description
Cache coherence	barnes	16K particles	Barnes-Hut hierarchical N-body method
	mp3d	30K particles	Rarefied fluid flow simulation
	fmm	16K particles	N-body adaptive fast multi-pole method
	fft	2^{16} data points	Complex 1-D Fast Fourier Transform
	mpeg2enc	352x288 CIF image	MPEG2 video encoder
Streaming	179.art	SPEC reference data set	Image recognition
	mpeg2enc	352x288 CIF image	MPEG2 video encoder
	bitonic	2^{19} 32-bit keys	Bitonic sort
	depth	352x288 CIF image	Stereo depth extraction
Transactions	barnes	8K particles	Barnes-Hut hierarchical N-body method
	mp3d	30K particles	Rarefied fluid flow simulation

Table 4. System parameters used for simulation

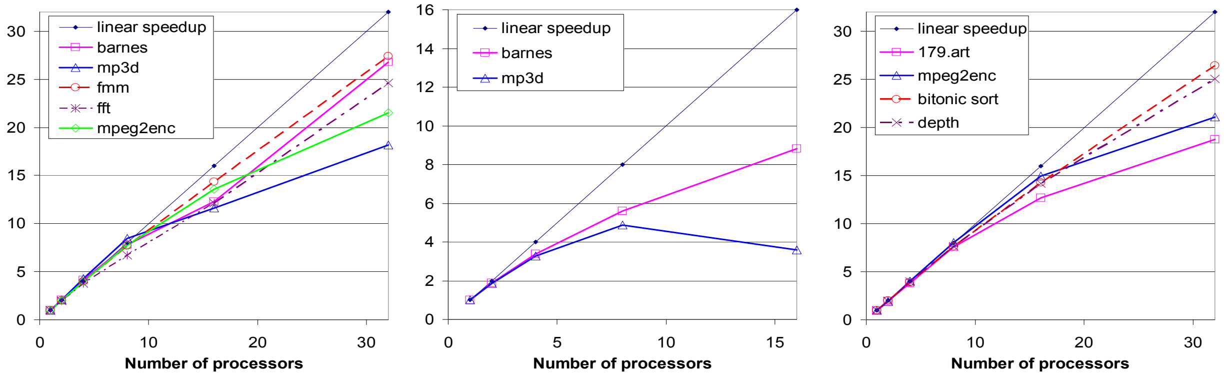
Shared Memory	I-cache	8KB, 2-way associative, 32B line size, 1 port (per processor)
	D-cache	16KB, 2-way associative, 32B line size, 1 port (per processor)
Streaming	I-cache	8KB, 1-way associative, 32B line size, 1 port (per processor)
	D-cache	4KB, 1-way associative, 32B line size, 1 port (per processor)
	Local Memory	20KB per processor, 4KB shared between all processors
TCC	I-Cache	16KB, 2-way associative, 32B line size, 1 port (shared between two processors)
	D-Cache	32KB, 4-way associative, 32B line size, 1K entry Store Address FIFO, 1 port (shared)
	Local Memory	4KB, shared between all processors
Common	Protocol controller	28 MSHRs (24 for processor requests, 4 for coherence requests)
	L2-cache (unified)	4MB, 4-way, 32B line size, 10 cycle latency, banked among main memory controllers
	Switch latency	5 cycles
	Memory controller	2 controllers per Quad, 32 MSHRs each
	Main memory	100 cycle access latency

coherent mode, speedups range from 18 to 27 for a 32-processor configuration; in streaming mode, speedups range from 18 to 26. The exception is mp3d in transactional mode, which does not scale beyond 8 parallel processors. The reason for this is frequent accesses to shared data structures, which cause transaction dependency violations and transaction re-execution. On the other hand, in a cache-coherent version of mp3d, these accesses are not protected by locks and as a result they cause data races. Since mp3d performs randomized computation and reports results only after statistical averaging of many steps of computation, the data races should not alter the results significantly. The reason for this uncommon programming decision is performance: in conventional cache-coherent architectures, fine-grain locking is expensive.

To estimate the performance overhead of reconfigurability, we repeated the same set of simulations on a "zero latency" model, where internal protocol operations take zero cycles, and calculated the difference with the real case. Note that this difference is an upper bound for the overhead estimate, because in any realistic fixed function design the protocol controller latency cannot be zero. Figure 9 illustrates the performance scaling of benchmark applications for both real and “idealized” controllers, with Table 5 summarizing this information. For each benchmark the overhead is averaged for system configurations ranging from 1 to 32 processors. In most cases the difference is less than 20%. The exception is the cache-coherent version of mp3d. The reason for this is once again frequent accesses to shared data structures without locks which cause frequent data races and put significant stress on the memory system. On the other hand, as one might expect, streaming applications are least sensitive to the controller latencies because of their latency tolerant nature.

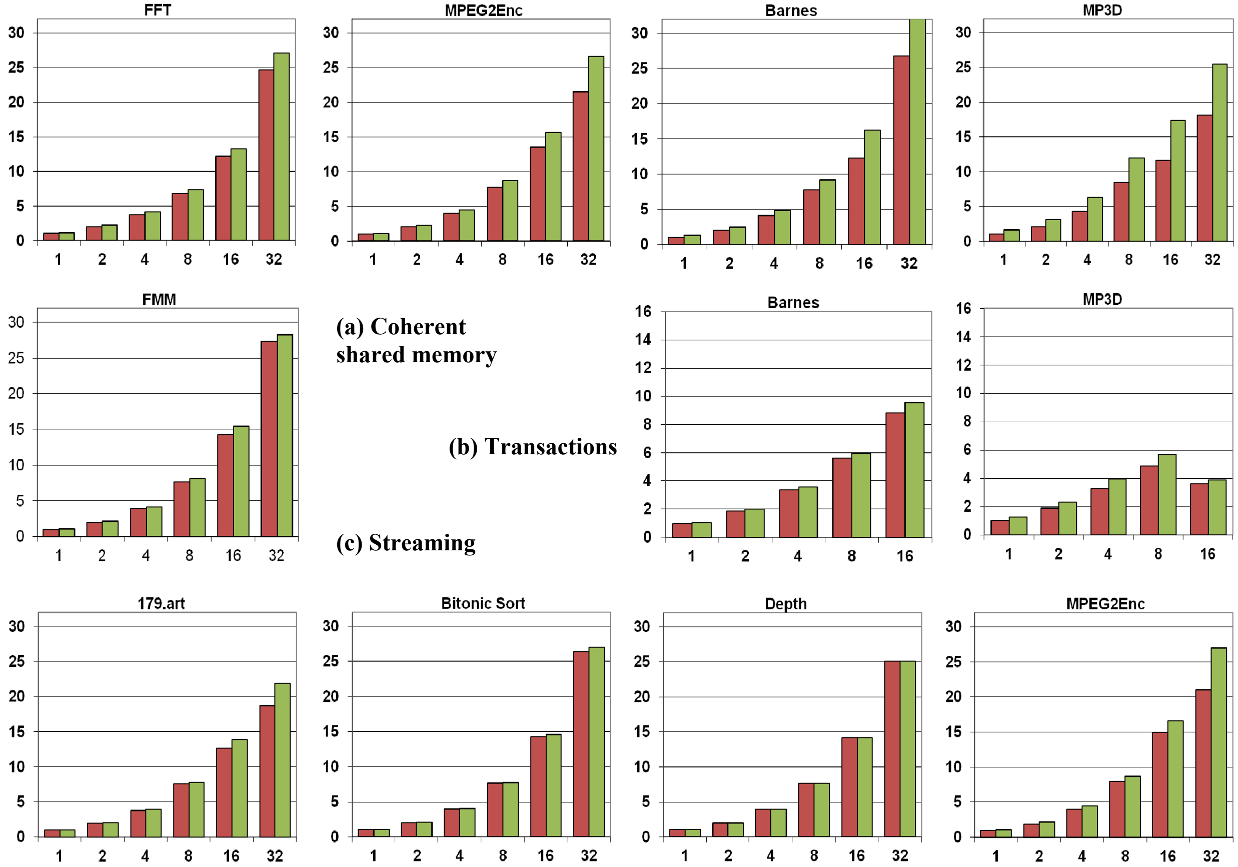
The 8-processor polymorphic CMP test chip parameters are summarized in Table 6. The test chip contains 4 Tiles, each with 2 Tensilica processors, and a shared protocol controller. The total chip area is 60.5 mm^2, and the core area, which includes tiles and protocol controller, is 51.7 mm^2 (Table 7).

To evaluate the hardware overhead of building a reconfigurable protocol controller rather than using it to generate the desired controller, we performed a series of simple experiments, in which we tailored the protocol controller to a specific memory protocol by converting all the internal configuration memories into constant values. Our synthesis tool then removed the memories and propagated the constant values into the logic, eliminating unnecessary parts and creating an “instance” of the controller tailored to that specific memory protocol. Figure 10 compares the



a) Cache coherence **b) Transactions** **c) Streaming**

Figure 8. Performance scaling



(a) Coherent shared memory

(b) Transactions

(c) Streaming

Figure 9. Performance comparison between real (dark) and idealized controllers (light).

area of the baseline controller with the specialized instances. The average reduction is around 50%. As illustrated, almost half the area savings is achieved by removing the configuration memories (flip-flops in Figure 10). This large area overhead was mainly caused by using flops to store the controller microcode. Future designs could reduce this overhead by 3x, which would greatly reduce the area cost of programmability. Even with the 50%

Table 5. Performance overhead

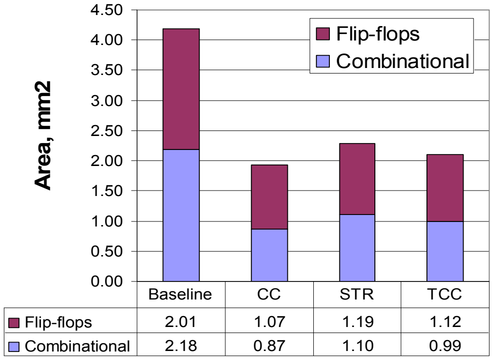
Model	App.	Overhead %	Average %
Cache coherence	barnes	24.29	20.99
	mp3d	48.59	
	fmm	6.93	
	fft	10.64	
	mpeg2enc	14.51	
Streaming	179.art	7.58	5.91
	mpeg2enc	14.14	
	bitonic	1.88	
	depth	0.06	
Transactions	barnes	8.33	14.18
	mp3d	20.03	

Table 6. Test chip parameters

Technology	ST 90nm-GP (General Purpose)
Supply voltage	1.0 V
I/O voltage	2.5 V
Dimensions	7.77mm × 7.77mm
Clock cycle time	5.5 ns (181 MHz)
Transistors	55 M
Gates	2.9 M (500K in protocol controller, 600K in each Tile)
Memory macros	128 (32 per Tile)
Pins	202 signal and 187 power/ground pins

Table 7. Test chip area breakdown

Unit	mm^2	%
Tile	10.0	16.5
CC	7.2	11.9
All Tiles	40.0	66.1
Routing channels	4.5	7.5
Pad ring	8.8	14.6
Core	51.7	85.4
Chip	60.5	100.0



	Baseline	CC	STR	TCC
Flip-flops	2.01	1.07	1.19	1.12
Combinational	2.18	0.87	1.10	0.99

Figure 10. Protocol controller area after synthesis

overhead, the percentage of the area consumed by the protocol controller is relatively small: around 12% of the test chip area and less than 14% of the core area (Table 7). Thus, protocol programability area overhead is less than 7% of the total system area.

7. CONCLUSIONS

As we move towards CMPs with many replicated cores, designing the memory system and the associated communication interfaces and protocols becomes one of the most important and difficult microarchitecture tasks. We provide a framework that helps address this problem. By creating a standard set of hardware units with simple operations, we convert this hardware design problem to a software programming problem. By defining the messages that each hardware unit must handle and the sequence of steps, the "subroutine," that needs to be run for each message, one completely defines the protocol's operation down to the RTL level. If we are correct in that this framework allows one to create any memory model, its greatest strength will be moving memory design and verification conceptually up a level. Instead of worrying about gates, a designer would only need to worry about the state that needs to be stored, and the operations that need to be executed. The overhead of using this method appears modest. Compared with a customized protocol controller with no internal delay the performance difference for most applications was less than 20%. Somewhat surprisingly, our results indicate it is feasible to directly implement the programmable framework on silicon. While that doubles the controller area, the controller consumes only 14% of the core area.

ACKNOWLEDGMENTS

This work would have not been possible without support from DARPA, Tensilica and ST Microelectronics. The authors also would like to thank Megan Wachs, Han Chen, Wajahat Qadeer, Rehan Hameed, Kyle Kelley, Francois Labonte, Jacob Chang and Don Stark for their help and support.

8. REFERENCES

[1] P. Kongetira, K. Aingaran, K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," IEEE Micro Magazine, Vol. 25, No. 2, pp. 21-29, March/April 2005.

[2] G. Grohoski, "Niagara-2: A Highly Threaded Server-on-a-Chip," 18th Hot Chips Symposium, August 2006.

[3] B. Khailany, W.J. Dally, U.J. Kapasi, P. Mattson, J. Namkoong, J.D. Owens, B. Towels, A. Chang, S. Rixner, "Imagine: Media Processing with Streams," IEEE Micro Magazine, Vol. 21, No. 2, pp. 35-46, April/March 2001.

[4] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, K. Yazawa, "The Design and Implementation of a First-Generation CELL Processor," Digest of Technical Papers, ISSCC, Vol. 1, pp. 184-185, February 2005.

[5] L. Hammond, B. Hubbert , M. Siu, M. Prabhu , M. Chen , and K. Olukotun, "The Stanford Hydra CMP," IEEE Micro Magazine, Vol. 20, Issue 2., pp. 71-84, March/April 2000.

[6] JG Steffan, C Colohan, A Zhai, TC Mowry, "The STAMPede Approach to Thread-Level Speculation," ACM Transactions on Computer Systems (TOCS), Vol. 23, Issue 3, pp. 253-300, August 2005.

[7] M. Herlihy and J.E.B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," ISCA-20, pp. 289-300, 1993.

[8] K. Mai, T. Paaske, N. Jayasena, R. Ho, W.J. Dally, M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," ISCA-27, pp. 161- 171, 2000.

[9] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, C.R. Moore, "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture," ISCA-30, pp. 422-433, June 2003.

[10] L. Hammond et al., "Transactional Memory Coherence and Consistency," ISCA-31, p. 102, June 2004.

[11] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, D.A. Wood, ‘‘LogTM: Log-Based Transactional Memory,’’ HPCA-12, pp. 254-265, 2006.

[12] S. K. Reinhardt, J. R. Larus, D. A. Wood, "Tempest and typhoon: user-level shared memory," ISCA-21, pp. 325-336, 1994.

[13] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chaplin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, J. Hennessy, "The Stanford FLASH multiprocessor," ISCA-21, pp. 302-313, 1994.

[14] J.R. Larus, R. Rajwar, "Transactional Memory," Synthesis Lectures On Computer Architecture, Morgan & Claypool Publishers, 2007.

[15] R.E. Gonzalez, "Xtensa: a configurable and extensible processor," Micro, IEEE Magazine, Vol.20, Issue 2., pp. 60-70, Mar/Apr 2000.

[16] Tensilica, Webpage: http://www.tensilica.com/

[17] D. Culler, J.P. Singh, A. Gupta, "Parallel Computer Architecture, A Hardware/Software Approach," Morgan-Kaufman Publishers Inc, 1999.

[18] J.B. Carter, W.C. Hsieh, L.B. Stoller, M.R. Swanson, L. Zhang, E.L. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M.A. Parker, L. Schaelicke, and T. Tateyama, "Impulse: Building a Smarter Memory Controller," HPCA-5, pp 70-79, 1999.

[19] F. Pong, M. Browne, A. Nowatzyk, M., Dubois, "Design and Verification of the S3.mp Cache-Coherent Shared-Memory System," IEEE Transactions On Computers, Vol. 47, No. 1, pp. 135-140, January 1998.

[20] A. Agarwal, R., Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B-H., Lim, K., Mackenzie, D. Yeung, “The MIT Alewife Machine: Architecture and Performance,” ISCA-22, pp 2-13, June 1995.

[21] S. Narayanasamy, B. Carneal, B. Calder, “Patching Processor Design Errors,” ICCD, pp. 491-498, October 2006.

[22] S.R. Sarangi, A. Tiwari, J., Torrellas, “Phoenix: Detecting and Recovering from Permanent Processor Design Bugs with Programmable Hardware,” MICRO-39, pp. 26-37, 2006.

[23] I. Wagner, V. Bertacco, T. Austin, “Using Field-Repairable Control Logic to Correct Design Errors in Microprocessors”, IEEE Transactions on Computer-Aided Design (TCAD), Vol. 27, Issue 2, pp. 380-393, February 2008.

[24] A.K. Nanda, A.-T. Nguyen, M.M. Michael, D.J. Joseph, “High-Throughput Coherence Controllers,” HPCA-6, pp. 145-155, 2000.

[25] A.-T. Nguyen, J. Torrellas, “Design Trade-Offs in High-Throughput Coherence Controllers,” PACT-12, p. 194, 2003.

[26] M.M. Michael, A.K. Nanda, B.-H. Lim, M.L. Scott, “Coherence Controller Architectures for SMP-Based CC-NUMA Multiprocessors,” ISCA-24, pp. 219-228, 1997.

[27] IBM tutorial, "Cell Broadband Engine solution, Software Development Kit v3.1: SPE configuration." http://publib.boulder.ibm.com/infocenter/systems/scope/syssw/index.jsp?topic=/eiccj/tutorial/cbet_3memfc.html

[28] P. Conway, B. Hughes, "The AMD Opteron Northbridge Architecture," IEEE Micro, vol.27, no.2, pp.10-21, March-April 2007. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4287392&isnumber=4287384

[29] Sun Microsystems, Inc., "OpenSPARC(tm) T1 Microarchitecture Specification," Part No. 819-6650-10, August 2006, Revision A. http://opensparc-t1.sunsource.net/ specs/OpenSPARCT1_Micro_Arch.pdf