UNIVERSITY OF CALIFORNIA, SAN DIEGO

Efficient Cache-Coherent Migration for Heterogeneous Coprocessors in Dark Silicon Limited Technology

A Thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

Scott Ricketts

Committee in charge:

Professor Michael Bedford Taylor, Co-Chair
Professor Steven Swanson, Co-Chair
Professor Chung-Kuan Cheng

2011

The Thesis of Scott Ricketts is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____
Co-Chair

_____
Co-Chair

University of California, San Diego

2011

DEDICATION

To my mom and dad.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS AND ABBREVIATIONS

btl beetle

btl-cc beetle with cache coherence

c-core conservation core

CMNI Communications Memory Network Incoming

CMNO Communications Memory Network Outgoing

CPU Central Processing Unit

d-cache data cache

DMA Direct Memory Access

DMNI D-cache Memory Network Incoming

DRAM Dynamic Random-Access Memory

GDN General-purpose Dynamic Network

i-cache instruction cache

IMNI I-cache Memory Network Incoming

ISA Instruction Set Architecture

LLSC Load-Linked Store-Conditional

MDN Memory Dynamic Network

MESI Modified/Exclusive/Shared/Invalid

MLI Multi-Level Inclusion

MPSoC Multiprocessor System-on-Chip

MRU Most Recently Used

NUCA Non-uniform Cache Access

PC Program Counter

PRF Pending Request File

RTL Register Transfer Level

UMNI User Memory Network Incoming

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my endlessly supportive mother and father, to whom this thesis is dedicated.

And of course, I thank Michael Taylor, my adviser, who taught me the science of hacking, among many other things, both technical and non-technical. I think it would be impossible to enumerate in paragraph form all of the ways he has impacted me since I moved to San Diego, so instead, I hope to repay him with a role in a future rap video. I would also like to thank my other committee members, Steve Swanson and CK Cheng.

Jack Sampson provided a great deal of feedback about the migration system, the results, and the writeup and was my liason to the GreenDroid toolchain. Thanks to Jack for counseling me through the final stages of this project with clarity and enthusiasm.

Ikkjin Ahn was hugely helpful during the early stages of this project as I acclimated to the Raw infrastructure. Thank you to Anshuman Gupta for discussions about the directory hardware design.

I also thank Houman Ghajari, my friend and manager at Maxentric. His flexibility and support throughout this project showed me that his first priority is the growth and well-being of the people in his team. Brandon Beresini, another colleague at Maxentric, has been my hacking partner-in-crime on a number of projects and is always eager to discuss an interesting technical problem, help debug a difficult software issue, or provide an excuse to go to Don Carlos and indulge.

Finally, I thank my brother, Daniel Ricketts, now a UCSD graduate student himself and truly my longest and most loyal friend.

ABSTRACT OF THE THESIS

Efficient Cache-Coherent Migration for Heterogeneous Coprocessors in Dark Silicon Limited Technology

by

Scott Ricketts

Master of Science in Computer Science

University of California, San Diego, 2011

Professor Michael Bedford Taylor, Co-Chair
Professor Steven Swanson, Co-Chair

Current trends in processor manufacturing indicate that in order to meet power budgets, chips will have to power-gate an increasing fraction of transistors. This so called *dark silicon* will play an important role in motivating future architectures. Recent research proposes to allocate these large dark regions to power-efficient specialized coprocessors. GreenDroid is a tiled architecture where each tile includes a set of coprocessors, generated at design time, that target the anticipated workload. A central challenge to retaining the efficiency of this system is providing low overhead migration mechanisms so that tasks can move around the chip to exploit different coprocessors. This thesis presents a prototype migration

system for GreenDroid and other tiled coprocessor-based architectures. The results show that for single-threaded irregular workloads, migration between c-cores spread across a four-tile configuration can provide power savings from 4.8× to 6.2× compared to an all software approach on a single tile. When compared against an ideal case where all c-cores fit on a single tile, the energy overhead of migrating amongst tiles ranges from 2.6% to 24.1%.

Chapter 1

Introduction

Current mobile application processors resemble their desktop and mainframe counterparts from past generations, thanks to sustained advancement in fabrication technologies. In the past decade, multicore has emerged as the paradigm for scaling energy efficiency and performance with process technologies. Current multicore designs add processing elements in the hope of scaling performance. However, as the number of cores increase, so does the footprint of the architectural supporting cast – for example, the on-chip caches, directories for cache coherence, and on-chip networks. These elements are designed to manage a challenging point of tension in the design of general purpose processors: retaining programmability while improving performance. For instance, modern operating systems require cache-coherent shared memory, but directories are typically power hungry, area hungry, or both, and coherence protocols apply pressure on the on-chip networks and caches, again increasing the demand for silicon and energy.

While increasing transistor counts makes chip area increasingly cheap, the chip power budget remains constant. In the past, processors have been able to continue to scale at full utilization while remaining at the same power level. However, we are entering a new regime of scaling where leakage limits our ability to reduce operating voltage and thus hinders our ability to reign in power consumption. In fact, recent research indicates that in order to remain within power budgets, future systems will have to effectively turn off an increasing portion of the chip. This is the *utilization wall*, and the portion of the chip that is not switching is often called

dark silicon.

Consequently, chip designers will have to re-examine the multicore paradigm for the next generation of low power processors under this new set of constraints. Our research group at UCSD has approached this problem inspired by two insights. First, the consequence of future feature shrinking is that area will be cheap, while energy will be expensive. Second, custom logic is known to be significantly more energy efficient than general purpose processors. Thus, a proposed approach is to allocate the dark silicon for specialized logic. When the workload can leverage the specialized logic, it does so and leaves the general purpose logic dark, and when it cannot, it power-gates the specialized logic and uses the general purpose resources.

For broad applicability, such systems must employ a diverse toolset of such specialized, power efficient coprocessors. A given application is likely to require a subset of coprocessors that is spread across the chip. Thus, execution contexts must migrate throughout the chip, seeking out appropriate coprocessors. As the number of coprocessors on chip increases, the overhead of migration will become an increasing energy and delay burden and may negate the gains expected from using specialized logic in the first place. My research focuses on addressing this challenge.

Our group is designing and implementing a coprocessor-based research prototype, *GreenDroid* [1]. GreenDroid is a tiled multicore processor where each tile includes a general purpose processing core, L1 caches, and a set of coprocessors called *conservation cores*, or c-cores [2]. The GreenDroid toolchain can automatically generate c-cores based on the anticipated workload. The current prototype targets mobile phone applications, using the Android codebase as the source for c-core generation.

This thesis presents several contributions to the GreenDroid design. First, it details *Submarine*, a system for migrating execution between tiles to exploit arbitrary sets of c-cores with low overhead. Migration includes two key pieces: (1) the dispatch, where the current tile sends its program counter and register file state to a remote tile and (2) data transfer, where the state of data in the task's address space becomes visible to the remote tile. Submarine handles the first piece

using message passing over GreenDroid's on-chip network. The second piece, data transfer, occures implicitly through a cache-coherent shared memory system that I designed for GreenDroid.

Next, the thesis evaluates a set of optimizations for mitigating Submarine's migration overhead. The optimizations center around power-gating memory subsystem components as execution migrates. These optimizations are only effective if the frequency of migration is sufficiently low. However, my analysis of migration overhead indicates that using specialized hardware for key migration mechanisms can further reduce overhead and increase the threshold of migration frequency for which power gating optimizations can provide net savings. The results show that for single-threaded irregular workloads, migration between c-cores spread across a four-tile configuration can provide power savings from 4.8× to 6.2× compared to an all software approach on a single tile. When compared against an ideal case where all c-cores fit on a single tile, the energy overhead of migrating amongst tiles ranges from 2.6% to 24.1%.

To support this study, I made significant modifications to the cycle accurate Raw simulator, beetle (btl) [3]. Raw is a tiled multicore processor that is the basis for the general purpose elements of GreenDroid. However, Raw does not provide cache coherence, which is a central mechanism for migrating data along with execution contexts. The new cache-coherent simulator is called btl-cc. I also implemented a body of new software infrastructure for evaluating migration and shared memory systems on btl-cc, including migration and synchronization primitives, thread-safe standard libraries, and a set of coherence-friendly memory instructions. Moreover, while the focus of this research was for migration of irregular applications on GreenDroid and similar architectures, I also evaluated the coherence system for data parallel shared memory applications using the SPLASH2 benchmark suite [4].

The remainder of this document is organized as follows. Chapter 2 establishes the architectural model under study and describes the migration system. Chapter 3 details the cache coherence system and evaluates the protocol on the SPLASH2 workload. Chapter 4, specifies the power-gating policies in the memory

subsystem during migration. Chapter 5 reports and analyzes migration results. Chapter 6 gives an overview of related work, and Chapter 7 concludes. Finally, for those interested in implementation details, Appendix A discusses btl-cc and other toolchain upgrades that support cache-coherent shared memory in the evolved Raw system.

Chapter 2

Migrating Contexts to Coprocessors

The efficacy of specialized logic for increasingly diverse workloads depends on the ability to migrate processes efficiently across tiles to exploit various sets of coprocessors. The problem is fundamentally one of a cross-tile context switch. Logically, there are two parts to this context switch: (1) the transfer of register state and program counter, or the *dispatch* and (2) the transfer of data. In this chapter I present this migration system. First, I overview the migration system and the chip architecture, GreenDroid, that I used as a case study for my research. Then I discuss the two parts of context migration and their implementation details for GreenDroid. Finally, I specify the prototype Submarine primitives for providing task migration for GreenDroid.

2.1 System Overview

Each GreenDroid tile includes a host processor, L1 data and instruction caches, c-cores, and an interface to the on chip networks. Figure 2.1 shows how GreenDroid replicates these tiles in a grid. The architectural resources on the grid communicate over a set of dynamic on chip networks: the General-purpose Dynamic Network (GDN) and the Memory Dynamic Network (MDN). These networks are fast – words hop between neighboring tiles in one cycle. They are also

latency insensitive – that is, tiles "fire and forget" packets onto the network, and dynamic routers use packet header bits to get the payload to the correct destination. Both user- and system-level entities have access to these networks. The memory system, for example, sends cache miss requests over the MDN. Meanwhile, user-level programs are free to send DMA or I/O messages over the MDN as well. On the edge of the chip, directories sit between the MDN and the DRAMs, serializing all shared memory requests that map to their respective DRAMs.

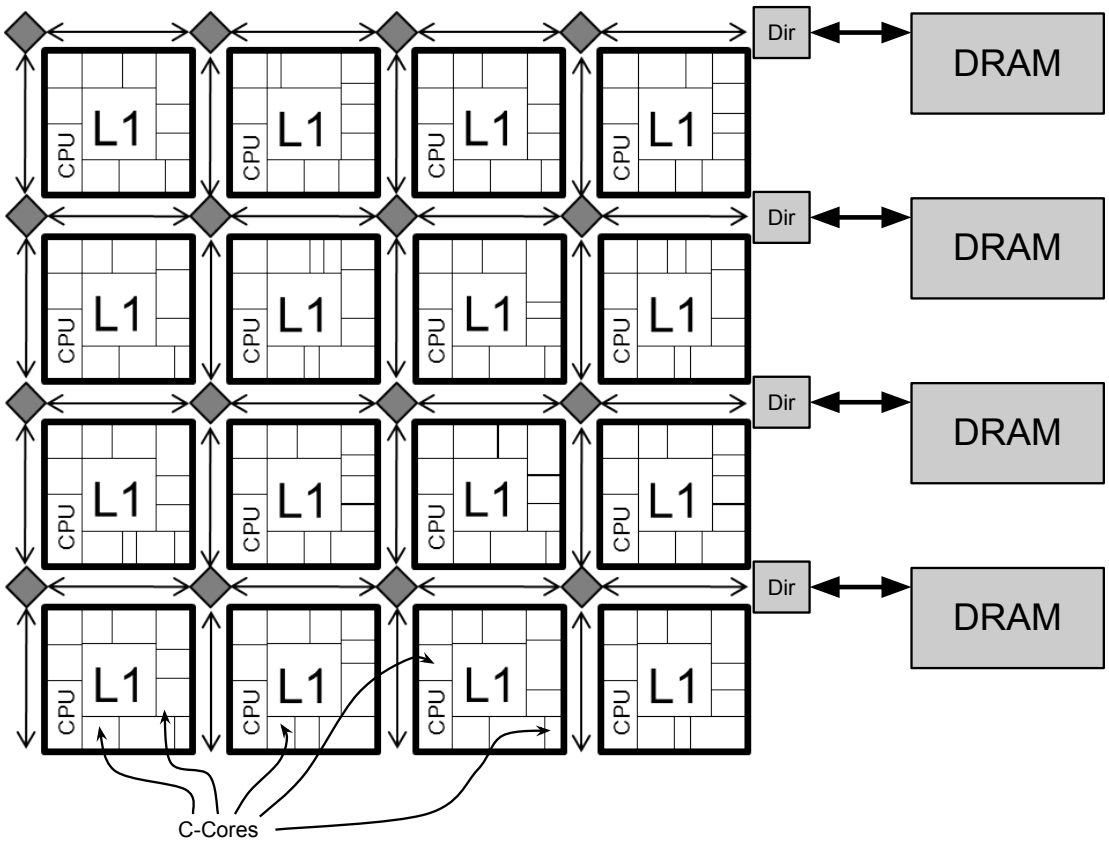

Figure 2.1: **A Floorplan for GreenDroid with Directory Coherence** In the cache-coherent GreenDroid prototype, directories sit between the MDN and DRAMs. Each tile includes a host CPU, L1 data and instruction caches, and a set of c-cores. The set of c-cores on each tile may be different, necessitating a mechanism for migrating execution between tiles.

The set of c-cores on each tile is not identical. GreenDroid targets mobile

application workloads – that is, it is meant to be the central processor for smart phones. The toolchain takes the Android codebase and automatically designs a set of c-cores that can cover 95% of execution of an example workload with about 7 mm^2 of chip area [1]. The motivation for c-cores is energy savings: a typical mobile application processor in a 45 nm process running at 1.5 GHz consumes 91 $^{pJ}/_{instr}$, while a GreenDroid c-core would consume just 8 $^{pJ}/_{instr}$. Accounting for code not covered by any c-core in the system, this would translate to about 12 $^{pJ}/_{s}$ – about a 7.5× improvement.

The c-cores and host processor share the L1 caches on the tile. At most one processor per tile is active: when execution reaches a c-core covered region, a c-core becomes active, and when execution returns to a software-covered region, the general purpose host processor takes over. It is the job of the toolchain to make c-cores out of the *hot code*, or code that consumes a large portion of the workload's executed instructions.

A task running on a GreenDroid tile may reach a region of code that maps well to a c-core on some remote tile. In this case, the task can migrate to the remote tile to exploit the c-core. The task need not migrate, however. Instead, it could execute locally in software on the host processor. Therefore, the system should be able to weigh the cost of migration against the potential gains from c-core execution. The cost of migration is largely dependent on the dispatch latency and the data transfer overhead. These factors affect energy and delay on the critical path of execution, but also affect energy consumption on the on-chip networks and in the memory system. Moreover, during migration, it may be desirable to power-gate subsystems on-chip for additional energy savings. These policies introduce additional overhead costs. The following sections describe the dispatch and data transfer pieces in greater detail.

2.2 Migration Dispatch and Migrating the Register File State

The toolchain annotates program code with region-to-tile mapping information. When execution encounters a new region that requires a migration, the host processor begins migrating the context. The first part of the context is the register file state and PC, which in this system is called the dispatch address. Thus, the host processor sends a 27-word *migration packet* to the destination tile that includes the dispatch address and register file state, omitting some unnecessary registers. Migration packets travel over the GDN. On the receive side, the destination tile copies in the register state and jumps to the dispatch address.

Table 2.1 provides the details of the migration packet. The migration packet does not include the tile-specific network registers, so migration calls should make sure to synchronize with respect to incoming or outgoing packets; that is, a migration should not occur when an incoming packet is on its way to the current tile or an outgoing packet is not completely sent from the current tile. Because memory instructions block on misses, the hardware already takes care of synchronization around memory-system packets. However, the user is responsible for such synchronization around user-generated packets. Also, the packet does not include the global pointer because in the prototype we only test one process at a time. In future systems, however, the packet may include the global pointer.

The link register is an interesting case. Because of the design of the migration dispatch, the packet does not include the link register. Consider the code snippet in Listing 2.1, which shows the send of the migration packet to the new tile. After sending the migration packet, the sender will jump into the runtime, to `migrate_log_exit_and_wait_for_dispatch`. Notice that the last word of the migration packet is the address at label `0`, which the receiving tile will interpret as the dispatch address, jumping to label `0`. The code at label `0` is actually the return code from the migration runtime function. Therefore, the new tile will put the correct return address from the call stack into the link register (`$31`) and then jump to this address. Therefore, there is no need to transfer the link register value

Table 2.1: **The Migration Packet** The migration packet includes a subset of the register file. Word 0 is the packet header. The Raw specification [5] has a full register file specification for the Raw system, which closely resembles the specification for the register file of the GreenDroid prototype.

Location in Packet	Registers	Use
0	–	GDN Header
1	$1	Reserved for assembler ($at)
2..3	$2..$3	Expression evaluation and return values
4..7	$4..$7	Procedure arguments
8..15	$8..$15	Temporaries
16..23	$16..$23	Callee saved registers
24	$29	The stack pointer
25	$30	A callee saved register
26	–	Dispatch address

to the new tile.

Note that there is still some unnecessary state included in the migration packet – e.g. the callee saved registers. These remain in the packet conservatively to be robust against changes to the ways that the toolchain inserts migration primitive calls.

By default, the c-core holds its state after a migration. This is important for nested c-core calls. When execution returns to a c-core, it can begin executing without any need to restore state. However, if a separate call requires the c-core, the state will be overwritten. Therefore, on migration, if the c-core state is still live, the runtime sets a *full-bit*. When another call needs the c-core, if the full-bit is set, it saves the state through a call into the runtime. In this sense, c-core state is considered callee-saved – but conditionally based on the full-bit. When execution returns to a c-core, it can restore state through the runtime. The toolchain c-core compiler limits the amount of state that can be claimed callee-saved. In this thesis, I assume that this limit is less than the size of a migration packet. This allows me to model c-core state migration in the same way as a register-file state migration.

Another interesting case is the `setjmp`/`longjmp` functionality in the POSIX

Listing 2.1: **Migrate Push Example** A snippet from the migration runtime system. In this case, `$3` happens to hold the packet header. The tile sends the migration packet over the GDN. The `$cgno` register is the output port to the GDN. As noted, the listing omits a chunk of code that moves registers 1-23 to `$cgno`.

```
migrate_to_tile:
        move $cgno,$3
        # code ommitted (move $1..$23 to $cgno)
        move $cgno, $sp
        move $cgno, $30
        move $4, $16
        move $5, $17
        jal init_for_new_tile
        la $8, 0f
        move $cgno, $8
        move $4, $18
        j migrate_log_exit_and_wait_for_dispatch
0:
        lw      $31,36($sp)
        lw      $18,32($sp)
        lw      $17,28($sp)
        lw      $16,24($sp)
        #nop
        addu    $sp,$sp,40
        j       $31
```

standard. On a call to `setjmp`, the C library saves program state, including the position in the call stack. On a subsequent `longjmp` call, execution returns to the state saved by `setjmp`. This is problematic for migration, because `longjmp` may cause the execution to leave the current region before reaching a migration primitive call into the runtime. Submarine solves this as follows. On a `setjmp` call, the runtime system saves the tile stack state in the same way that it saves the call stack state. On a `longjmp` call, the runtime restores the tile stack state and execution migrates to the tile where the `setjmp` call occurred. This functionality is useful for POSIX programming.

2.3 Migrating Data

The data transfer piece is more challenging. When execution leaves a tile, data in the L1 cache may be dirty. For regularly-structured applications, like streaming signal processing, it may be possible to use compiler analysis to insert data transfer functions that explicitly move data between tiles upon encountering a migration. However, for irregular applications, this is not practical. A naive solution would explicitly send all dirty data from the current tile to the next tile just before migration. However, this could negate any potential cache expansion benefits from migration, because the next tile is unlikely to need all of the dirty data in the current tile's cache. Instead, I designed a cache coherence system for GreenDroid so that data can move on demand through the shared memory hierarchy. Level 1 directories sit at the DRAM ports and implement a MESI protocol [6]. Chapter 3 discusses the cache coherence system in greater detail.

2.4 Migration Runtime Primitives

Currently, Submarine is designed for single-program execution, where a single task migrates throughout the chip. This allows me to isolate the overhead factors of migration for a single workload. A task can access the system through a set of primitives, which I overview in this section.

Regions can map to software execution on the host or to specialized execution on a particular c-core. At run time, when encountering the beginning of a new region, the system checks first to see if the region requires a c-core, and if so, whether it is available on the local tile. If the c-core is not available locally, the system migrates execution to the appropriate tile. At the end of a region, the system migrates execution back to the return tile. To track the migration directions, Submarine stores a *migration stack* of return tiles, in a similar manner to the way that the call stack tracks return addresses. Thus, the migration primitive at the beginning of a region is `migrate_push`(tile) and the primitive at the end is `migrate_pop`().

On a `migrate_push`, the current tile pushes its coordinates onto the tile

stack, sends a migration packet to the next tile, and then calls into the runtime. The runtime logs the exit and jumps to wait code that waits for the next dispatch from the GDN. In the prototype system, the runtime may power-gate certain subsystems on exit, depending on what policies are enabled – this is discussed in greater detail in Chapter 4. On a `migrate_pop`, the current tile pops the next tile's coordinates from the migration stack and sends over a migration packet in the same way as in `migrate_push`.

Future work could extend the Submarine design to support multiprogramming. The major necessary modifications would be the following. Firstly, the tile stack would be program-specific; therefore, Submarine would store a tile stack for each program. Also, there could be multiple incoming dispatch packets at a tile. To prevent GDN deadlock, dispatch packets would need to cause interrupts, and the associated interrupt handlers would enqueue the dispatch requests in some Submarine-managed work queue that the tile would service in order. Finally, the current program launch system assumes a k-tile configuration rooted at tile 0, with the other available tiles arranged contiguously from tile 1 to tile $k-1$ in the mesh. However, programs in multiprogrammed workloads would require arbitrary launch locations, which would require some minor modifications to the Submarine initialization code.

Chapter 3

The Cache Coherence System

As processes migrate across the chip, the cache coherence system migrates data on demand through the shared memory hierarchy. There are a number of elements at play in a coherence system. The protocol defines the allowed cacheline states and the message transactions that can change the state of a cacheline and move data through the hierarchy.

The architectural implementation is quasi-independent from the protocol. Typically, for small numbers of caches, implementations are bus based – the caches snoop the bus to avoid illegal states. This approach does not scale well, however, because as the number of caches increases, it becomes impractical to broadcast every transaction over a single bus. Instead, scalable manycore architectures typically employ point-to-point interconnects for memory traffic. Indeed, Raw and GreenDroid use their MDN for this purpose. Still, the system needs to serialize the MDN transactions to maintain coherence. A common architectural approach is the *directory*. Directories track the states of memory blocks across the chip, receive requests from caches, and implement the protocol accordingly.

Moreover, implementing coherence protocols over an on chip network creates the challenge of deadlock avoidance. There are a number of common approaches to this problem, all of which have tradeoffs in delay and on chip area and energy consumption. Typically, if there are seperate physical or virtual networks for requests and replies, deadlock avoidance becomes an easier problem. However, this option is costly from an energy perspective, and thus it is undesirable un-

der the constraints for GreenDroid. Instead, I choose to use a single network for all coherence traffic and ensure deadlock avoidance by guaranteeing that for any message sent, there will be sufficient buffering space on the receiver to sink the message, freeing the network.

In this chapter I discuss the design decisions for the prototype cache coherence system for GreenDroid. First, I detail the coherence protocol. Next, I describe the directory design. Finally, I explain the deadlock avoidance approach.

3.1 The Submarine Coherence Protocol

This section details the Submarine cache coherence protocol for the GreenDroid prototype. The target workload for this research is irregular applications. The approach was to reduce single-threaded coherence overhead while retaining correctness and reasonable performance scalability for multi-threaded workloads. As a result, the chosen protocol is based on the conventional MESI protocol. In some cases there are opportunities for optimization of the implementation when many caches share the same block of data – however, the design decision was to avoid these in favor of simplicity.

3.1.1 Protocol Overview

At the cache, each line includes the normal dirty and valid bits along with a new bit for coherence: the readonly bit. These bits encode the coherence state of the cacheline – one of the MESI states (Modified, Exclusive, Shared, or Invalid) – as enumerated in Table 3.1. The valid and readonly bits are part of the tag array. Therefore, during the tag stage of the pipeline, a memory instruction operating on a particular block can determine whether the block is valid in the cache and if so, whether the tile has write access to the appropriate line. If the instruction misses or if there is insufficient write permission for a store instruction, the cache controller starts the cache state machine, which sends a coherence request onto the MDN.

Directories at the DRAM ports receive the coherence requests. Each direc-

Table 3.1: **The Cacheline Coherence States** Submarine implements a MESI protocol using valid and readonly bits in the tag arrays and dirty bits in the status arrays. Note that a line cannot be valid, dirty, and readonly, because rendering a line dirty requires write access.

Valid	Dirty	Readonly	Coherence State
0	*	*	Invalid
1	0	1	Shared
1	0	0	Exclusive
1	1	0	Modified
1	1	1	Not allowed

tory entry stores the state of a memory block. The state can be Owned, Shared, Unowned, or Pending. The Pending state indicates that an incomplete request on the block is still pending and that additional state regarding the block is located in the Pending Request File (PRF). There is no Modified state in the directory, because the directory does not know when an L1 cache modifies a line. Thus, it assumes conservatively that any block marked as Owned in the directory entry could be a Modified line in the owner cache.

3.1.2 Coherent Instruction Semantics and Protocol Details

The Raw ISA is MIPS-like. The GreenDroid prototype will have a similar ISA. For this study, I modified the Raw ISA to have coherent semantics. The coherence system supports the previous Raw instructions, with the exception of `tagsw`, which causes issues because it allows the user to modify the tags – and therefore the coherence state – arbitrarily. I added load-linked (`ll`) and store-conditional (`sc`) for synchronization, similar to the primitives of the same names in the MIPS R4000 architecture [7]. The coherence system considers the Raw cache administrative instructions – i.e. instructions for flushing and invalidating cachelines – to have global semantics by default; that is, the coherence system applies them to all caches. I provide more details about these ISA changes in Appendix A.

For reasoning about coherence, it is helpful to group the memory instructions into three families:

1. *Sh* - instructions which require the shared state
2. *Ex* - instructions which require the exclusive state
3. *Flinv* - global and non-global invalidations, flushes, and flush-invalidations

With the exception of load-linked, all loads require the shared state. Load-linked goes straight to an exclusive state, because it is assumed that a store-conditional to the same address will follow shortly. Load-linked and all stores require the exclusive state.

Figure 3.1 is the protocol diagram for Sh instructions, that is, instructions which require the shared state. Coherent loads require that no cache client in the system has write access to the block. The directory ensures this property. It then either passes data along to the requesting client or sends a cacheline read request to the DRAM on behalf of the requesting client. If the incoming request is on a block that is unowned, the directory will grant exclusive access to the requester, based on the heuristic that loads are typically followed by stores to the same address, and forward the request to the DRAM on behalf of the requester. If the block is owned, the directory sends a remote flush request to the owner, who downgrades to shared access and responds with an acknowledgment with or without data. An acknowledgment without data simply means that the data is already clean in the owner's cache. If the incoming request is on a block that already has a pending state, the directory will NACK the request. Notice that if the block is in a shared state, the directory unnecessarily sends a read to the DRAM instead of getting the data from an on-chip sharer. The challenge of asking a sharer for the data is that the line may be evicted before the remote request for data arrives. This adds complexity to the protocol, but is not a major obstacle. Other protocols, like MOESI, explicitly choose an owner of every block that is responsible for keeping valid data on chip, even when it is clean.

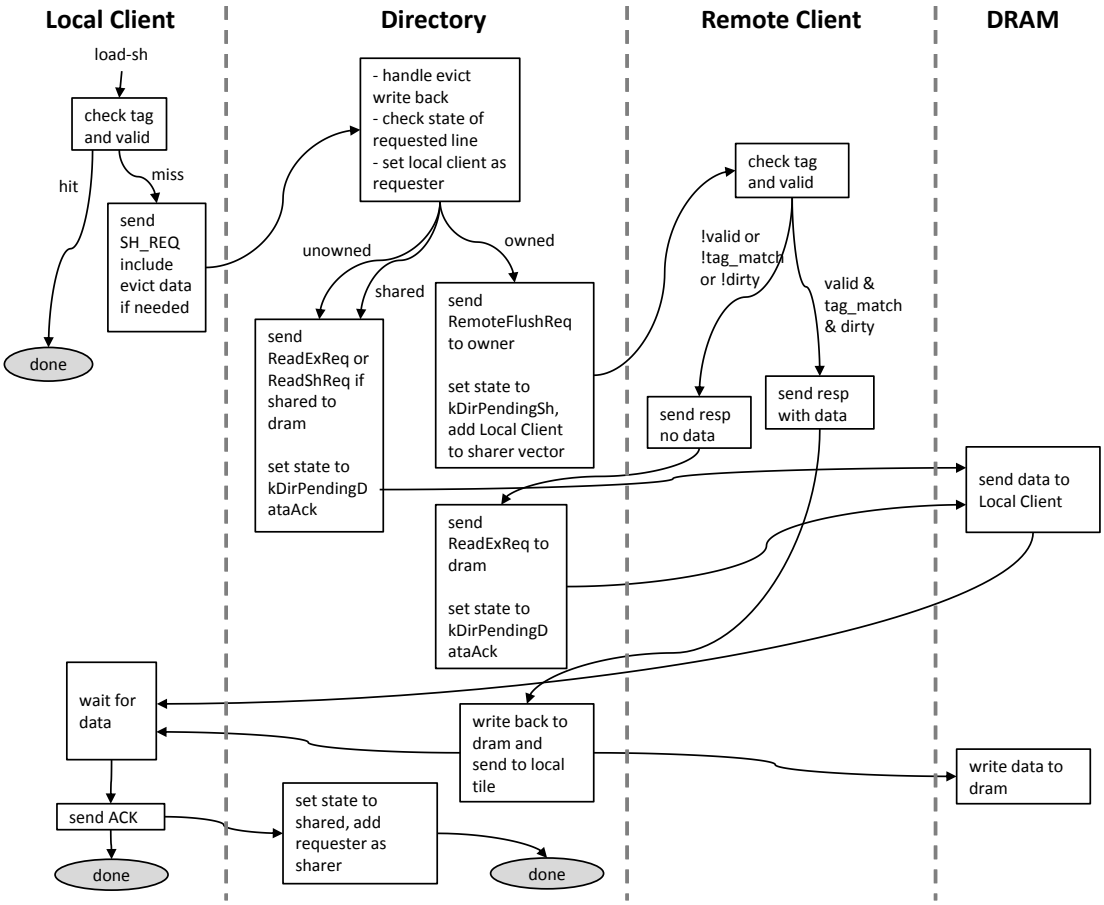

Figure 3.1: **Coherent Load Protocol Diagram** The diagram shows the protocol activity in response to a request for shared access. The directory ensures that before granting this access, no other cache in the system has write access. If the directory entry state is owned, the directory communicates with the remote owner client to downgrade the write access to shared access.

Figure 3.2 is the protocol diagram for Ex instructions, or instructions which require exclusive access. Stores and exclusive loads (e.g. load-linked) require that no other cache client has a valid copy of the block. If the requesting client already has shared access, it will send an upgrade request to the directory. If it does not have a valid copy of the line, it will send an exclusive request. The directory handles both of these requests similarly. If the block is not owned anywhere in the system, the directory forwards the request to the DRAM. If the block is owned, the

directory sends a remote flush-invalidate request to the owner. This transaction is similar to the remote flush, except that the remote client must invalidate the line. If the incoming request is on a block that already has a pending state, the directory will NACK the request. Notice that if the block is in a shared state, the directory invalidates all of the sharers, losing the ability to get the data from on chip and requiring a costly DRAM request. This is a similar issue to that of the unnecessary DRAM read for shared requests on blocks already in the shared state. Similarly, a potential solution would be a MOESI-based protocol.

Finally, Figure 3.3 is the protocol diagram for the Flinv family of memory instructions. As previously mentioned, Flinv instructions have global semantics by default. That is, when a single tile issues the instruction, the coherence system applies the request to all caches. Therefore, even if the block is invalid in the local client cache, the request must go to the directory in order to determine if any other caches have the block. Submarine also supports local versions of the Flinv instructions, which are not shown in this diagram. There is no local invalidate currently, but the protocol could support it if necessary. Also, Figure 3.3 does not show the protocol for handling these requests when there is already a pending request on the line. The race conditions of pending requests relative to Flinv instructions present some complexity to the system. As it is currently designed, the directory cannot NACK requests with data. Removing this requirement could simplify these issues at the directory, but might also add complexity at the cache.

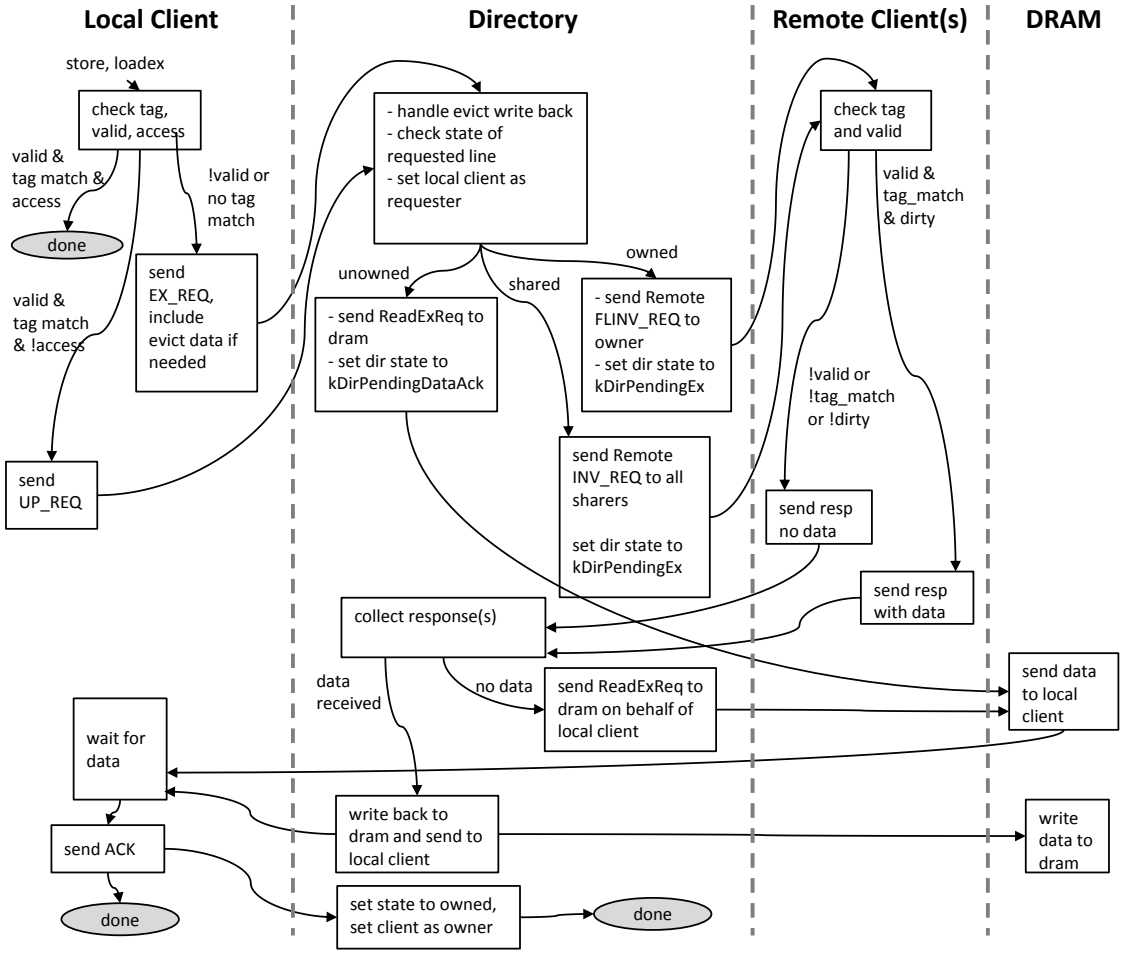

Figure 3.2: **Coherent Store/LoadEx Protocol Diagram** The diagram shows the protocol activity in response to a request for exclusive access, either through an upgrade request or exclusive request. Before granting exclusive access, the directory ensures that no other valid copies of the block are in any cache in the system. It establishes this state by sending remote requests to remote clients as pictured.

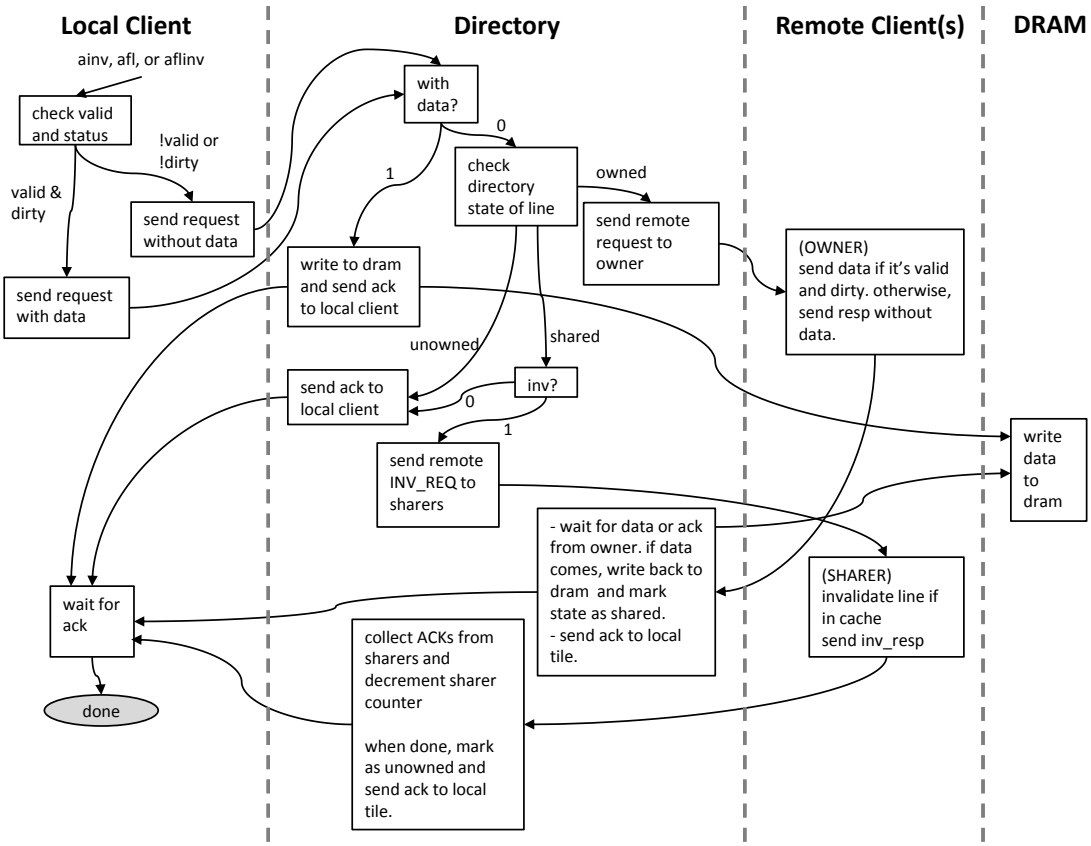

Figure 3.3: **Coherent Invalidate/Flush/Flush-Invalidate Protocol Diagram** The diagram shows the protocol activity in response to an invalidation, flush, or flush-invalidation. The local client blocks waiting for acknowledgment that the transaction is complete. The directory sends the acknowledgment once it has applied the request globally across the chip. For example, to service a flush request, if some remote client has a dirty copy of the line, the directory sends a remote request to the remote client to flush the line. Once the remote flush is complete, the directory acknowledges the requester.

3.2 The Directory

The directories manage coherence for a given migration group of tiles. Directories are challenging to design scalably in both area and energy. Duplicate-tags

directories have associativity that scales linearly in the number of tiles, while common alternatives like vector- or pointer-based sparse directories require large area to prevent directory entry evictions. While there have been a number of proposed novel approaches to directory design for scalable manycore architectures [8], I choose to mitigate scalability problems by assuming that in the common case, an application will use a small set of k tiles. The toolchain determines this set of tiles at design time. The directory then allocates k directory entry ways to support coherence across k tiles. For each tile, the directory duplicates its tags in a directory way. Alongside each tag is the directory entry, which records the state. At run time, if more tiles are needed, a second level of directories or a multiplexed alternative general-purpose directory can provide correctness. This falls in line with the c-core paradigm: an energy-efficient design for the common case and a less efficient but general design as a catch-all.

Figure 3.4 shows the directory design for a k-tile migration group. During a pending request, the directory needs to hold extra state information while messages are in flight across the chip. Thus, each directory module includes a small PRF, indexed by client ID and also content addressable by memory address. The directory also includes an MDN Arbitrator that arbitrates between the directory and DRAM for memory access. Not pictured are the FIFOs between modules. Some of the FIFOs are employed for performance reasons, while others are used as sinks to prevent deadlock. The next section discusses the FIFO requirements in greater detail.

3.3 Deadlock Avoidance

Consider a network model where we reason about clients as having two autonomous modules: the network interface, which sends and receives words onto and from the network, and the controller, which processes those words. The key component to deadlock avoidance on the MDN is the guarantee that the receive-side network interface can always sink messages, even if the receive-side controller is stalled. For arbitrary network protocols, of course, this would be quite challenging,

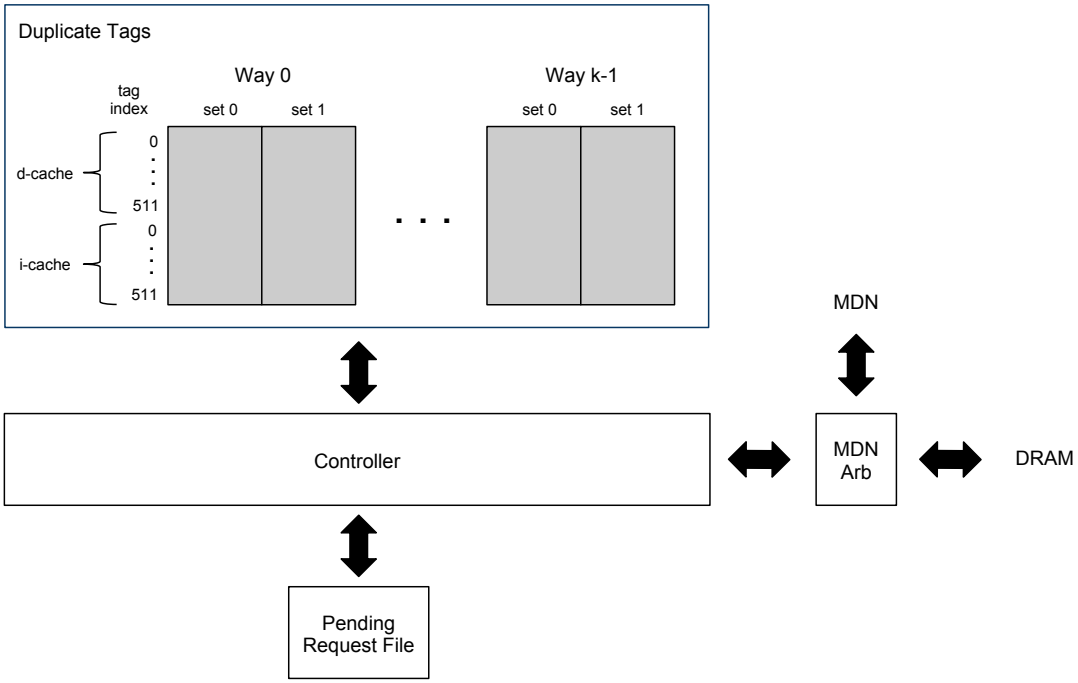

Figure 3.4: **A Directory for a k-Tile Migration Group** The directory manages coherence for a fixed number of tiles. The design partitions the storage into per-tile ways.

but the memory system model adds sufficient structure to the messaging traffic over the MDN such that proving upper bounds for buffering is possible. This section establishes these bounds for the three types of MDN client: the directory, the data cache (d-cache), and the instruction cache (i-cache).

3.3.1 Buffering at the directory

The directory will never block an incoming network word as long as there is always enough buffering space to sink it. Thus, it is necessary to calculate the minimal storage size S_{dir}, measured in words, that will satisfy the property that at any time there will be no more than S_{dir} words destined for the directory on the network.

To calculate this upper bound, we make the requirement that before initiating a request on the MDN, a cache must guarantee that enough buffering exists

at the directory to sink all traffic related to the request. Because of the structure of the coherence protocol, we can pre-allocate this buffering. Let B_D and B_I be the necessary buffering allocation for outstanding d-cache and i-cache requests, respectively.

Moreover, each cache has a fixed upper bound on the number of outstanding memory requests that it can make. Let this bound be R. For example, Raw does not allow the cache to have more than one memory request in flight, and thus for Raw, $R = 1$.

If there are d d-caches and i i-caches in the system, we can then bound S as follows:

$$S_{dir} \leq dRB_D + iRB_I \tag{3.1}$$

There are three phases in the fulfillment of a memory request from a requesting cache, referred to as the *local cache*:

1. Request Phase - The cache sends the request to the directory. At this point the directory may NACK the request, which would logically end the request.

2. Remote Phase - If necessary, the directory uses messaging with other clients in the system to establish a state that meets the coherence requirements for granting the request. These messages will lead to responses from remote clients back to the directory. For example, the directory may need to need to multicast an invalidation to a number of clients, and those clients would then send responses to indicate that the invalidation has taken place.

3. Ack Phase - The directory sends a response to the local cache, which may require that the local cache send an acknowledgment back to the directory.

Let $b_p^{(d)}$ and $b_p^{(i)}$ be the storage required at the directory for phase p of d-cache and i-cache requests, respectively. Since a request is always in exactly one of those three phases, we then have the following:

$$B_D = \max_{p=1}^{3} b_p^{(d)} \tag{3.2}$$

$$B_I = \max_{p=1}^{3} b_p^{(i)} \tag{3.3}$$

Table 3.2: **Coherence Packet Formats** The largest in flight messages are the packet pairs that result from data cache misses with evicts: the cache piggy backs an eviction packet behind the miss request packet, which can be either a shared or exclusive request. Note that the table omits flush and flush-invalidate formats when they do not include data – these are the same formats as their "with data" counterparts, less the 8 data words.

Phase	Operation	Packet Format	Size
Request	Sh	Hdr, Addr	2
	Up	Hdr, SubHdr, Addr	3
	Ex	Hdr, Addr	2
	Sh+Evict	Sh packet; Hdr, Addr, Data	12
	Ex+Evict	Ex packet; Hdr, Addr, Data	12
	Inv	Hdr, SubHdr, Addr	3
	Fl, FlInv	Hdr, SubHdr, Addr, Data	11
Remote	FlResp, FlinvResp	Hdr, SubHdr, Data	10
	InvResp	Hdr, SubHdr	2
Ack	DataAck	Hdr, SubHdr	2

Table 3.2 lists the packet formats in each of the phases. For d-caches, the largest sized request is one that includes a miss and an eviction – for example, a shared request with eviction or an exclusive request with eviction. Such as message includes a header and address for the miss request, another header and address for the eviction, and then a cacheline of data. Thus, $b_1^{(d)} = 12$. Since i-caches do not evict data, their max request size is smaller. An invalidation note is the largest such message, including a header, subheader, and address, and so $b_1^{(i)} = 3$.

In the Remote Phase, an exclusive request can cause the directory to invalidate some number of lines at remote sharer clients. In the current implementation, the directory multicasts to all sharers. Since invalidation responses have length 2, if there are k sharers, the directory would need to allocate $2k$ words of buffering. We can reduce this requirement by sending smaller groups of multicasts. In practice, the number of sharers on a given memory block is typically small. The focus of this study is small migration groups of four tiles or less, and so for the purposes

Table 3.3: **Max Words in Flight Per Outstanding Cache Request** At any time, the fulfillment of a cache request is in one of three phases. This table lists the max words in flight destined for the directory for each phase and then shows the max over all phases. The numbers are different between d-cache and i-cache requests because i-caches never evict.

Phase	D-Cache	I-Cache
Request	12	3
Remote	10	10
Ack	2	2
Max	12	10

of this analysis we assume that the remote phase will be limited by FlResp and FlinvResp packets, not InvResp packets. Therefore, $b_2^{(d)} = 10$. In any case, a general-purpose implementation of this system would likely limit multicast groups to a small number of tiles.

Since i-caches never require exclusive access, their requests never require that a remote cache invalidate a line. The only type of remote response possible as part of satisfying an i-cache request is a flush response, which includes a header, subheader, and cacheline. Therefore, $b_2^{(i)} = 10$, whether or not the directory supports smaller invalidation multicasts.

Finally, in the Ack Phase, the only type of message that is sent to the directory is an acknowledgment, which includes a header and subheader. Thus, $b_3^{(d)} = 2$ and $b_3^{(i)} = 2$.

Table 3.3 summarizes these calculations. From Equations 3.2 and 3.3, we have

$$B_D = 12 \tag{3.4}$$

and

$$B_I = 10 \tag{3.5}$$

Finally, this allows us to calculate the upper bound S from Equation 3.1. An n-tile GreenDroid or Raw system has one d-cache and one i-cache per tile. For

this study we assume that the GreenDroid design will limit tiles to one outstanding request per cache – i.e. $R = 1$, as is the case with Raw. Therefore we have:

$$S_{dir} \leq 22n \tag{3.6}$$

To see that this is a tight bound, consider the following example. Assume each d-cache request is in the Request Phase, and is a miss with an eviction. This creates $12n$ words in flight destined for the directory. Meanwhile, each i-cache request is a miss on a block that is exclusive in another cache. In the Remote Phase of these requests, the directory will be waiting on flush responses, creating $10n$ words in flight. This gives us $22n$ words in flight. For a directory supporting a 4-tile migration group, this translates to 88 words or 352 bytes of buffering. A general purpose directory supporting a full 4-by-4 mesh of GreenDroid tiles would require 352 words or 1408 bytes of buffering. For reference, a tag RAM for a GreenDroid data cache is 2560 bytes.

This sink buffering at the directory is implemented with the CMNI FIFO. The *CMNx* terminology is a relic of the Raw network interface register naming conventions. Thus, the outgoing FIFO is called the CMNO FIFO.

The directory also requires a CMNO FIFO for outgoing packets onto the MDN. The CMNO FIFO must have enough storage to hold the maximum size packet, because the MDN will not begin sending a CMNO packet until it is completely ready. This is to prevent deadlock between the directory, DRAM, and MDN.

3.3.2 Buffering at the caches

The tile multiplexes MDN input to three different buffers: the UMNI FIFO, the DMNI FIFO, and the IMNI FIFO. The UMNI buffering requirements are defined by the legacy user-level use of the MDN, which is not relevant to this study. The DMNI and IMNI buffering requirements are upper bounded by the maximum size of a response and the maximal allocation of in flight remote requests.

A response to a cache is either an acknowledgment – 2 or 3 words – or a cacheline response with a header and 8 words of data. At the same time, the cache

must be able to sink any incoming remote request. There can be at most one incoming remote request for each outstanding request on chip. Since we assume $R = 1$, there can be at most $2n$ incoming remote requests: n on behalf of data caches and n on behalf of instruction caches. Each remote request is a header, subheader, and address. Therefore, the necessary buffering at the caches is given by:

$$S_{DMNI} = S_{IMNI} = 9 + 6n \tag{3.7}$$

The buffering requirements at the caches and directory grow with the number of tiles in the system. However, the directory storage will dominate, assuming the duplicate-tags model. As the number of tiles grows large, we will have to design more clever protocol and storage methods for reducing RAM requirements for the general-purpose directories.

3.4 Evaluation of the Coherence System for Parallel Workloads

While the focus of this thesis is on migration of irregular, sequential programs, cache coherence designs typically optimize for parallel performance. My intention is for this protocol to be lightweight for sequential execution and provide reasonably efficient and correct execution of parallel workloads. This section presents an evaluation of the cache coherence system that btl-cc models in order to show that the system has low overhead for sequential programs and scales well for parallel programs.

3.4.1 Single-Tile Performance Overhead

Table 3.4 lists the overheads measured for two workloads: data parallel SPLASH2 benchmarks [4] and irregular benchmarks from SPEC2000 [9], EEMBC [10], and libjpeg [11]. The results indicate that on average, the coherence system adds low overhead: 1.4% for the SPLASH2 benchmarks and 3.3% for the irregular benchmarks. The highest observed overhead is 8.2%.

Table 3.4: **Single-Tile Runtime Overhead of Cache Coherence** The results below show the performance overhead of running a single-tile with cache coherence on btl-cc (With CC) vs without cache coherence on btl (No CC). The overhead for these single-tile runs is at most 8.2% and on average is low – 1.4% for the SPLASH2 workload and 3.3% for the irregular workload.

Category	Benchmark	No CC (cycles)	With CC (cycles)	Overhead (%)
SPLASH2	radix	25.1M	26.0M	3.3
	cholesky	1.33B	1.36B	1.8
	barnes	291M	295M	1.6
	fft	74.9M	76.3M	1.8
	water-spatial	2.50B	2.51B	0.0
	water-nsquared	1.30B	1.30B	0.1
	Mean	921M	927M	1.4
Irregular	bzip2	1.75B	1.89B	8.2
	cjpeg	1.98B	2.00B	1.0
	gzip	2.36B	2.43B	2.8
	mcf	586M	621M	6.0
	djpeg	701M	714M	1.9
	viterbi	818M	818M	0.0
	Mean	1.37B	1.41B	3.3

The outliers on the high end are bzip2 and mcf, which also have much higher data cache stalls as a proportion of run time on the non-coherent system than do the other irregular benchmarks. Chapter 5 takes a closer look at the irregular benchmarks.

3.4.2 Parallel Performance

Figure 3.5 presents scalability results for the SPLASH2 benchmarks running on btl-cc. As a point of comparison, I also examined the MIT Alewife Machine [12] and the Stanford DASH [13]. These architectures were early prototype distributed shared memory multi-processors with directory-based coherence systems.

The published evaluations of these processors provide scaling results for the original SPLASH benchmarks, allowing for some points of comparison with my SPLASH2 results. The SPLASH2 water-nsquared benchmark is similar to the original SPLASH water benchmark. Therefore, it is possible to plot the Alewife and DASH scalability numbers for the water benchmark alongside the water-nsquared scaling on btl-cc. The Alewife results included an fft kernel, and though not the SPLASH2 fft, I use it as an approximation of expected scaling as well.

The water-nsquared results are quite similar for all three architectures, showing speedups around 7× for 8 processors. The remaining three benchmarks exhibit reasonable scaling, although there are some concerns about the performance beyond four processors. The general indication, though, is that the btl-cc coherence system meets the requirement of reasonable efficiency for parallel workloads, given that the current focus is irregular sequential applications.

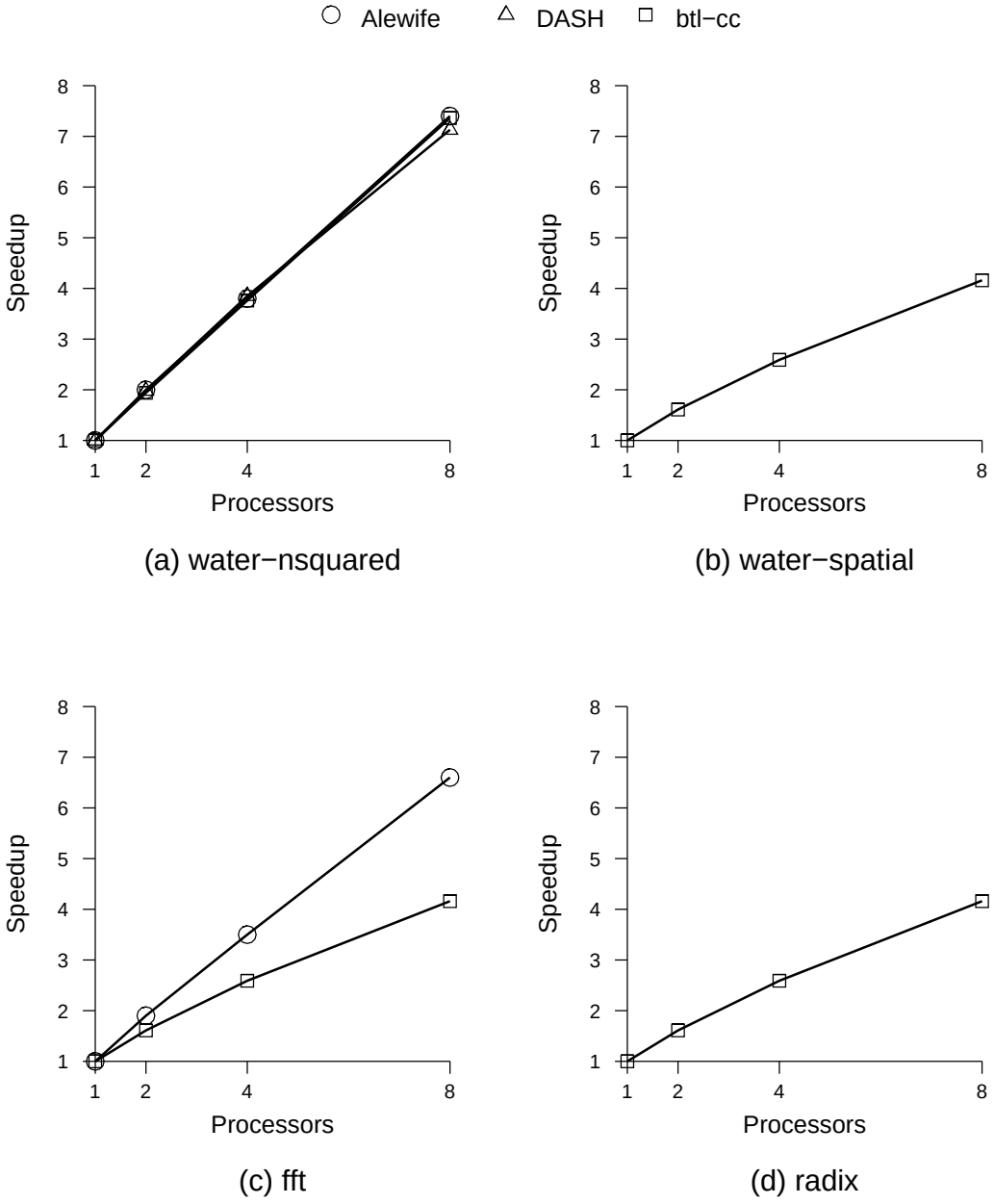

Figure 3.5: **Scaling SPLASH2 Benchmarks on btl-cc** The plots show scaling of parallel benchmarks on btl-cc. For water-nsquared, the plot includes data points for Alewife and DASH from published results. The fft plot also includes published results for Alewife.

Chapter 4

Memory System Napping

When a process leaves a tile, the host and coprocessors can move to a power-gated state. However, the data cache will likely contain dirty data, thus preventing the immediate power gating of the data, tag, and status RAMs. I consider three policies for optimizing the data cache energy consumption, shown in Table 4.1 and called *cache nap policies*. In this chapter I describe the policies, specify the algorithm that decides whether or not to nap after execution leaves the tile, and discuss the overhead elements of napping.

4.1 The Cache Nap Policies

In *awake caches*, all cache components remain active. In *drowsy caches*, the tile flushes the cache before napping so that it can put the data into a low leakage mode [14], while the tags remain active so that the tile can service coherence requests. Note that since all lines are clean, the tile can power-gate the status array, as long as we can tolerate losing the Most Recently Used (MRU) bits. In *sleepy caches*, the tile flush-invalidates the cache so that it can power-gate all cache components. Moreover, since all cachelines are invalid, the directory no longer needs to track this cache and can therefore power-gate the corresponding way in its duplicate tag memory. In the other policies, this way must remain active.

Table 4.1: **Cache Napping Policies** In an awake cache, the cache is fully active. In a drowsy cache, we flush all dirty lines, power-gate the status array, and put the data array into a low leakage state, leaving the tag array active for servicing coherence requests. In sleepy caches, we flush-invalidate, then power-gate the data, tags, and status, and moreover, since the cache no longer has valid data, we gate the directory way corresponding to the tile. In the table, *P-Gated* means power-gated.

Policy	Before Napping	Data	Tags	Status	Directory Way
Awake	No nap	Active	Active	Active	Active
Drowsy	Flush	Low-leakage	Active	P-Gated	Active
Sleepy	Flush-Invalidate	P-Gated	P-Gated	P-Gated	P-Gated

4.2 The Nap Decision

Both `migrate_pop` and `migrate_push` have the same nap-related behavior and more generally, are *exit points*. When a tile exits, the nap controller decides whether or not to nap. If the subsequent idle time will be long enough to justify the napping overhead, the nap controller should decide in the affirmative. Thus, at an exit point the nap controller must make a prediction about idle time. To this end, the nap controller stores a history of idle times at each exit point in the program, indexed by a hash of the instruction address. If the history indicates that the next idle time will surpass a given threshold, the nap controller sets a nap signal. The current prototype models the nap controller as a hardware module on each tile. Figure 4.1 provides a schematic of the *idleness predictor*, a component of the nap controller that predicts the next amount of idle time.

4.3 Nap Overhead

In the critical path of a migrating execution, the process must dispatch to the new tile and the new tile must wakeup and begin executing. These are the *dispatch* and *wakeup* latencies, respectively. Once application instructions start

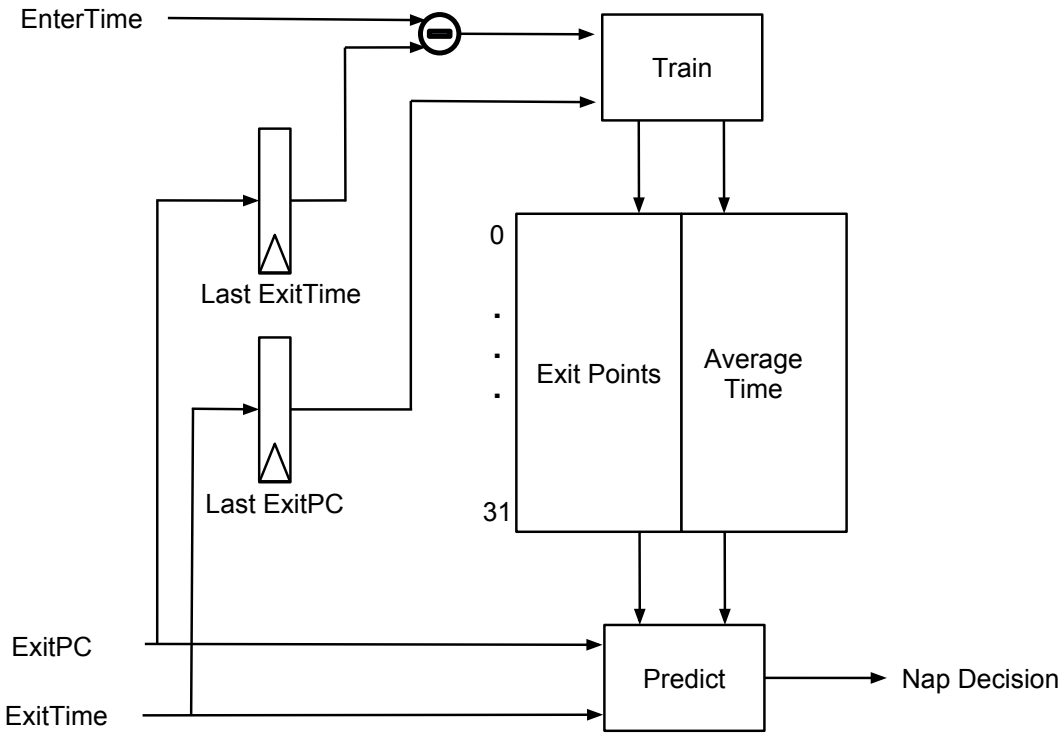

Figure 4.1: **The Idleness Predictor** When a tile exits – i.e. execution migrates to another tile – the local nap controller predicts whether the tile will be idle for long enough to merit napping.

executing on the new tile, the process is in the *compute* phase.

The compute latency for a migrating application may differ from that of the same application executing on a single tile because of cache effects. On the one hand, well designed migration can create cache expansion: if the migration points correspond to changes in the working set, each subset of the overall working set gets its own cache. On the other hand, if there are too many data conflicts between the different migration phases, coherence latencies may create undesirable delay overhead as the process migrates. For this reason, the reader will notice that in Chapter 5, when analyzing migration results I distinguish between compute cycles and cycles where the data cache is stalling the pipeline.

The dispatch latency is fairly fixed. Assuming that the GDN is free, the message passing latency will be on the order of 30 cycles for the dispatch packet.

Sometimes the instruction cache will miss when calling the dispatch routine, but this at worst pushes the dispatch latency to about 100 cycles.

Similarly, the wakeup latency is not dependent on application behavior. Instead, wakeup depends on the nap policies for the processors and cache. Awake caches do not require wakeup. For sleepy and drowsy caches, I assume a fixed 300 cycle wakeup time.

Of course, napping also creates more work outside of the critical path. This does not contribute to overhead in delay, but instead to overhead in energy. After dispatch, a tile must flush or flush-invalidate the cache before entering a drowsy- or sleepy-cache state, respectively. This is the *flinv* phase, and it encapsulates either of the two cache management routines.

In the next chapter, I evaluate the impact of these overhead components, looking at both energy and delay.

Chapter 5

Migration Results

The value of exploiting c-cores for increasingly diverse workloads depends on the ability to migrate processes efficiently across tiles to exploit various sets of coprocessors. I have presented a system for migrating execution using the GDN for copying the register file state and sending the dispatch and the coherence system for transferring data. I have also detailed three cache nap policies for additional energy saving during migration. In this chapter I evaluate these approaches and provide insight about effective optimizations for future migration systems.

5.1 Methodology

I model GreenDroid using a modified version of btl, the Raw cycle-accurate simulator. Raw [15] is a tiled processor from which GreenDroid derives a number of its features, including the general purpose processing core and on chip networks. While Raw does not have c-cores, c-core execution closely follows the original control- and data-flow of the program, so c-core execution time and memory access patterns are similar to running all in software on a Raw processor. We can adjust btl results to resemble GreenDroid results with c-cores by applying an energy model that distinguishes between execution on the host processor and on c-cores.

Table 5.1 details the power model parameters for a tile. I assume a 45 nm technology node and derive processor power model parameters from previous research [1]. In simulation results assume a 1500 MHz processor clock and 667

Table 5.1: **The Tile Power Model** The tile power model comes from previous research [1] and queries to CACTI [16]. The table gives dynamic energy for processors in pJ per cycle and for RAMs in pJ per access.

Structure	Leakage (mW)	Dynamic Energy (pJ per unit)
Host Processor	1.25	91
C-Core Processor	1.25	8
D-Cache Data	0.221	6.85
D-Cache Tags	0.017	1.63
D-Cache Status	0.002	0.30
Directory Way	0.017	1.63

MHz DRAM clock. Power models for data caches and directory ways come from CACTI [16], following the Raw data cache parameters (32 kB, 2-way, 1024-line) along with the added readonly bit in the tag for coherence support. I assume that each directory way will have the same properties as a data cache tag RAM.

In a drowsy cache, I assume a low leakage state that is 25% of normal leakage, based on previous research on drowsy caches [14]. For power-gated components, I assume no leakage and ignore the leakage of the power-gate itself, because I assume it will be small relative to other energy components.

Since Raw does not support cache coherence, I made some fundamental changes to the Raw toolchain and simulator to run experiments for this thesis. The modified simulator is called btl-cc (beetle with cache coherence). In Appendix A, I detail the modifications to the Raw toolchain as a case study for evolving a non-cache-coherent system into a cache-coherent system.

5.2 The Workloads

To isolate the energy and delay impacts of this migration system, I consider a set of single-process workloads, listed in Table 5.2, that migrate within a migration group of at most four tiles. The workloads come from SPEC 2000 [9], EEMBC [10], bzip2, and libjpeg [11]. The libjpeg benchmarks – cjpeg and djpeg

Table 5.2: **The Migration Workloads** The GreenDroid toolchain annotates each program with migration calls that predetermine the number of tiles in its migration group. This table lists the number of migration tiles and the execution time in the profiled phase of the workload running on a single tile configuration with migration disabled. It also shows the average migration interval – the number of cycles between exit points.

Workload Name	Suite	Migration Tiles	Single-Tile Exec Time (cycles)	Avg Migration Interval (cycles)
bzip2	bzip2	4	1.9B	1.1M
cjpeg	libjpeg	4	2.0B	290K
djpeg	libjpeg	3	710M	112K
gzip	SPEC 2000	3	2.4B	121M
mcf	SPEC 2000	3	640M	78M
viterbi	EEMBC	2	820M	136K

– are similar to the corresponding jpeg benchmarks in EEMBC. The bzip2 benchmark is actually not the SPEC 2000 version but is the real bzip2 code. Note that since each benchmark requires at most 4 tiles, GreenDroid can fit each benchmark within a single row of a 4-by-4 configuration with a directory and DRAM on the east port of the row.

5.3 Energy

The motivation for GreenDroid is energy reduction. In this section I present energy measurements for the migration workloads running on btl-cc. The energy model accounts for both leakage and dynamic costs at the host processors, c-cores, L1 caches, and directory.

Figure 5.1 paints a promising picture for GreenDroid energy efficiency. The plot shows overall energy efficiency relative to an all-software solution running on a single tile host processor. Energy measurements are based on the model specified in Section 5.1, except that the plot varies the c-core efficiency – relative to the host processor – along the x-axis. Benchmark runs have migration enabled and

explore the space of all three cache nap policies – the plot only shows the nap policy with the best savings in each case, because the difference between best and worse was generally too small to display clearly. I assume that for the sleepy and drowsy policies, the flush and flush-invalidate routines will be implemented in hardware with the same energy model as a c-core. Results from the Android codebase indicate that c-core efficiency will be just over 11× for GreenDroid. At this level, we see energy savings between about 4× and 6×. Table 5.3 provides the energy savings figures at the expected GreenDroid c-core efficiency.

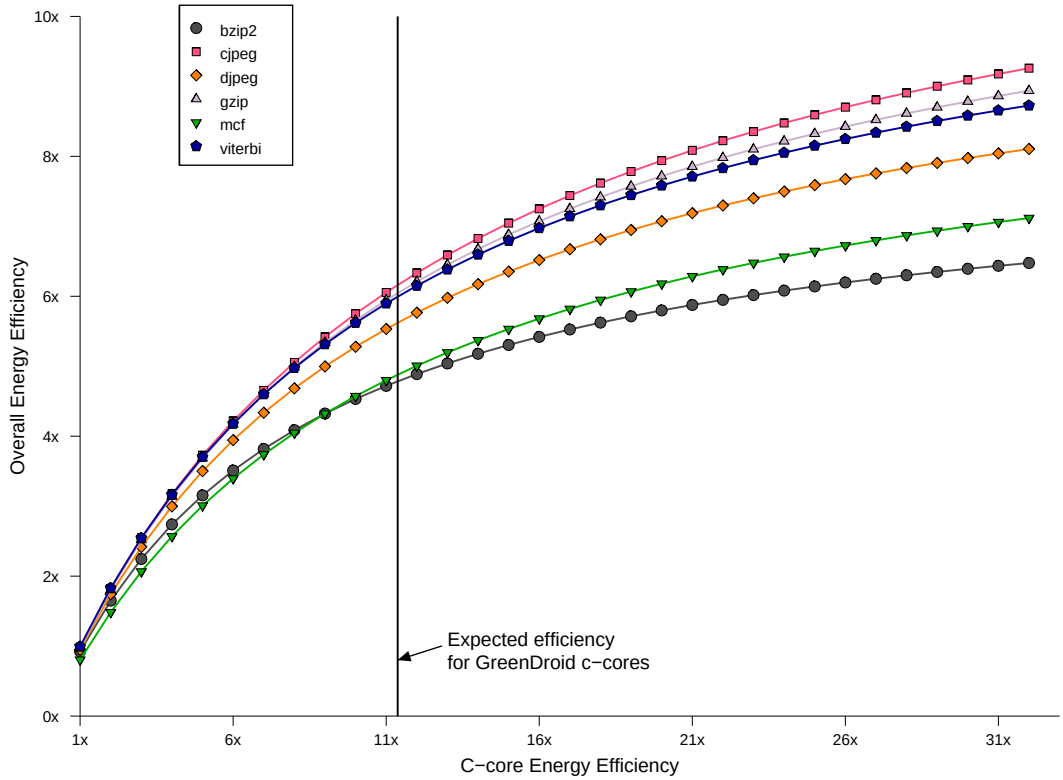

Figure 5.1: **Energy Savings from Migration** The plot shows, for each benchmark, the energy savings across the three cache nap policies relative to running on a single-tile host processor in software. Each data point gives the best energy savings across the nap policies. The plot varies the c-core efficiency relative to the host processor – otherwise the energy model is as specified in Section 5.1. The expected c-core efficiency for GreenDroid is just over 11×.

Table 5.3: **Measured Energy Savings for Migrating Irregular Workloads** The savings listed here are from the GreenDroid c-core efficiency point as marked in Figure 5.1. For the sleepy policy, savings range from 4.8× to 6.2×, and there is little variance across policies.

bench	awake	drowsy	sleepy
bzip2	4.7×	4.7×	4.8×
cjpeg	6.1×	6.1×	6.2×
djpeg	5.6×	5.3×	5.1×
gzip	5.9×	6.0×	6.0×
mcf	4.8×	4.8×	4.9×
viterbi	6.0×	5.9×	5.9×

The thin bar ranges are small, indicating that the cache nap policy choice has little impact. Also, we see similar savings across all six of the benchmarks. The benchmarks with the poorest energy savings are bzip2, mcf, and djpeg. As it turns out, these benchmarks demonstrate three different pathologies that increase migration overhead in this system.

In the following analysis, I will break down the contributing factors to migration overhead. Table 5.4 shows the complete list of energy components in the analysis. I consider the static and dynamic components of memory system energy separately for the directories and the L1 data caches. I also delineate the energy in the flush/flush-invalidate hardware, combining static and dynamic components. I then break down the host processor and c-core energy components by phase: energy consumed during wakeup, data cache stall, and other compute cycles. Each of these host/c-core components include both static and dynamic energy.

5.3.1 Overhead Analysis

Figure 5.2 shows the energy breakdowns for the expected GreenDroid c-core efficiency point. For each benchmark, the baseline case is the idealistic single-tile with all necessary c-cores, so that there is no migration overhead, pictured as the

Table 5.4: **Component Legend for Energy Measurement Results** Energy results in this chapter contain the the components listed here. Relevant plots use the abbreviated component names in their legends.

Component	Description
dirLeak	directory leakage
L1Leak	L1 data cache leakage
dirAccess	dynamic energy from reads and writes to the directory
L1Access	dynamic energy from reads and writes to the L1 cache
flinv	energy in the flush/flush-invalidate controller
hostDcacheStall	host proc leakage and dynamic energy during dcache stalls
ccoreDcacheStall	ccore leakage and dynamic energy during dcache stalls
wakeup	host proc energy during wakeup
hostCompute	all other host leakage and dynamic energy
ccoreCompute	all other ccore leakage and dynamic energy

left-most bar in each cluster. The other bars in the cluster represent configurations that spread the c-cores across the migration group and employ one of the three cache nap policies. The bar annotations show the overhead as a percentage of the baseline energy. Again, in this plot I assume that the flush/flush-invalidate routines run on hardware.

Notice first that in all cases, the compute components on the host and ccore have constant contribution across all four bars in each cluster. This is an intuitive invariance: the system is performing the same computation whether it allows migration or not.

There are no clear conclusions from this plot about cache nap policy choice. We would expect that the sleepy or drowsy policies would reduce overhead, and they do for five of the size benchmarks. However, the reductions are small and for djpeg, the drowsy and sleepy policies increase the overhead significantly.

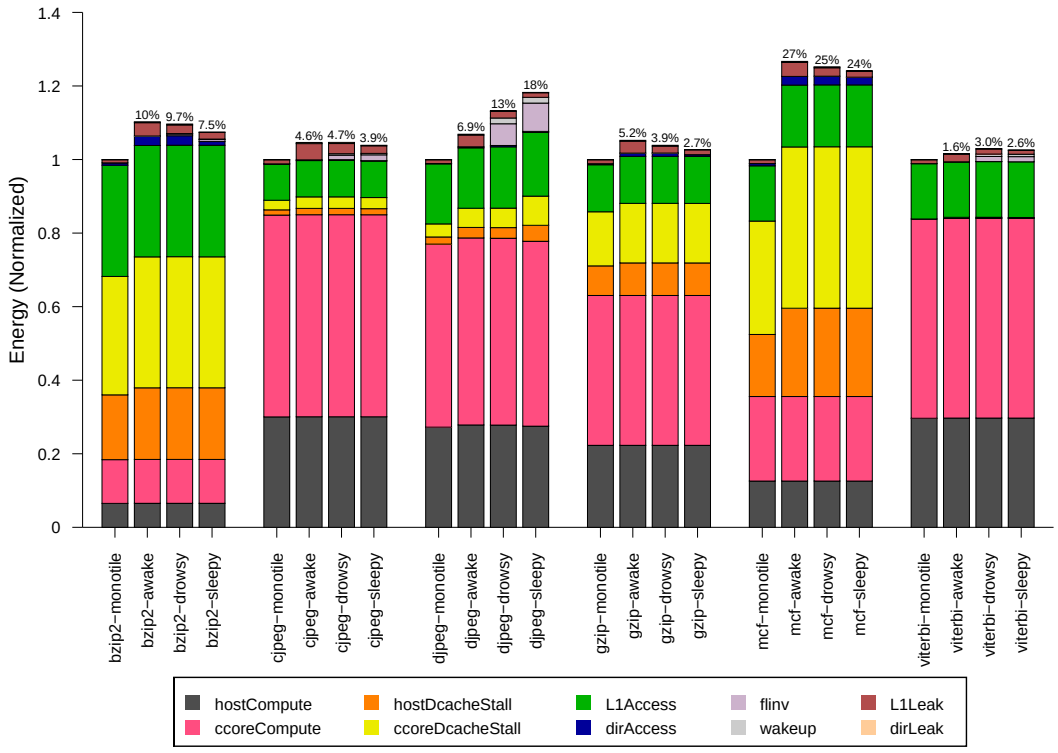

Figure 5.2: **Migration System Energy** This plot shows the energy in the migration system for each benchmark across the three cache policies. The baseline is a single-tile system where all c-cores are available so that migration is not necessary – the left-most bar in each cluster shows these data points. The annotations at the top of the other bars gives the overhead percentage relative to the baseline.

The previously mentioned poorly performing benchmarks – bzip2, djpeg, and mcf – have the highest overheads for any cache nap policy. Notice, interestingly, that while bzip and mcf have similar energy savings overall, the mcf overhead for the GreenDroid c-core efficiency point is more than double the bzip overhead for any of the cache nap policies. The reason for this is that on a single tile, mcf spends a larger proportion of its energy in the host processors and c-cores than bzip2. In other words, the compute cycles to L1 access ratio is higher for mcf, so it stands to benefit more from the energy savings provided by c-cores, mitigating

its high overhead relative to bzip2. The same is true for djpeg – despite having overheads between 6.9% and 18.2%, compared to 7.5% to 10.3% for bzip2, djpeg achieves greater energy savings because it has a higher ratio of processor energy to memory system energy. The other three "well behaved" benchmarks – cjpeg, gzip, and viterbi, exhibit the same behavior, but the effect is less pronounced because their overheads are relatively smaller.

That said, we would like to understand what some of the major contributing factors are to migration overhead. To look more closely at the overhead breakdown, I subtract the baseline from each component, leaving just the overhead residual. Figure 5.3 plots these residuals as a percentage of overhead for each (benchmark, policy) pair.

For mcf, the energy consumed in the host processors and c-cores during data cache stalls dominates the overhead. This is a result of a high cache miss rate that has a magnified energy penalty during migration, because misses have to go through costly coherence transactions. We know from Figure 5.2 that the sleepy and drowsy nap policies mitigate the overhead for mcf by reducing L1 leakage energy. Figure 5.3 shows that as this component decreases, the overhead from waiting for data cache stalls becomes increasingly important.

The bzip2 benchmark exhibits somewhat different energy behavior. Like in mcf, the sleepy and drowsy policies reduce the L1 leakage energy, but additionally, the bzip2 run demonstrates energy savings in the directory under the sleepy policy. The bzip2 benchmark migrates over four tiles, so under the other policies, the directory must access all four directory ways when looking for an entry. However, the sleepy policy is able to shut down up to three of the four ways. Again, as the sleepy and drowsy policies mitigate overhead, the data cache stalls in the processors become increasingly dominant in the overhead.

The djpeg benchmark is a largely different case. The overhead under the awake policy is actually not that bad relative to the other benchmarks, just 6.9%. Also, the sleepy and drowsy policies reduce the L1 leakage energy as expected. However, the wakeup and flush/flush-invalidate routine wipes out the savings from napping, and we see high overhead: 13.3% for the drowsy policy, which requires

flushing the cache, and 18.2% for the sleepy policy, which requires flush-invalidating the cache.

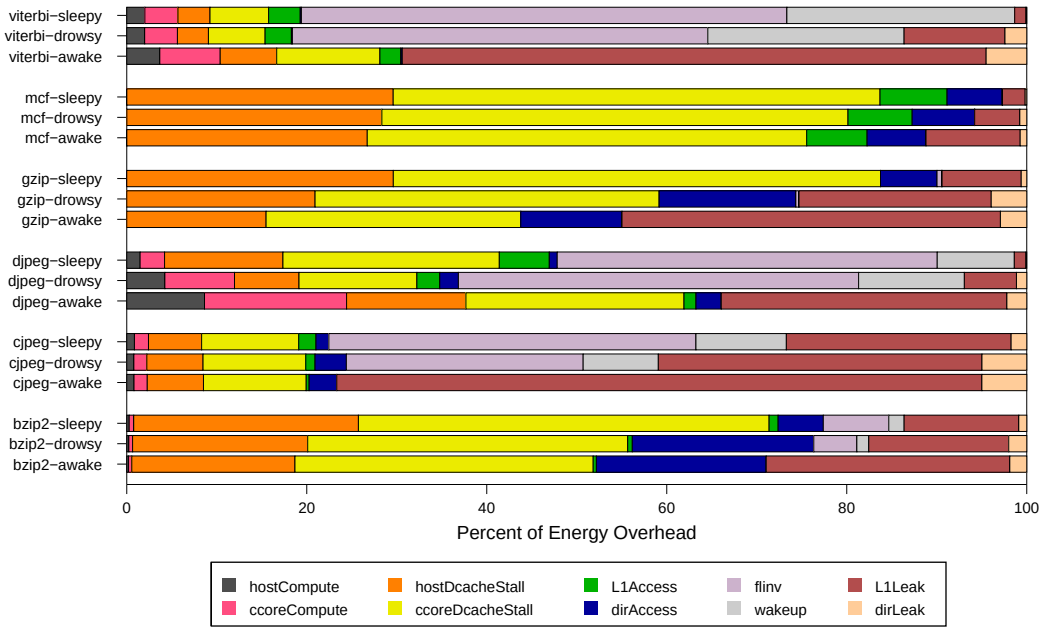

Figure 5.3: **Migration System Energy Overhead Breakdown** This plot breaks down the migration energy overhead components to picture the key factors. For each component data point, I subtract the baseline and plot the residuals as a percentage of the total overhead.

The gzip results show a reduction from the cache nap policies, going from 5.2% overhead with the awake policy, to 3.9% with drowsy, and then 2.7% with sleepy. Figure 5.3 shows that this reduction is largely a result of reductions in L1 leakage energy, with the sleepy policy providing additional savings in directory access energy.

While viterbi overall has low overhead – between 1.6% and 3.0% – it does not demonstrate improvement from the nap policies. Again, the wakeup and flush/flush-invalidate routines eliminate the savings. The sleepy policy actually offers a small improvement for cjpeg – from 4.6% down to 3.9%, but in general cjpeg suffers from the same issue.

Notice that the wakeup and flush/flush-invalidate penalty prevented nap

savings for cjpeg, viterbi, and djpeg. These benchmarks also have the shortest migration interval, migrating on average every 290K, 136K, and 112K cycles, respectively. Compare this to the other three benchmarks, which on average have between 1.1M and 121M cycles between exit points (see Table 5.2).

The analysis presented in this subsection leads to two insights. Firstly, poor coherence behavior can be problematic for migration if the host and/or c-core processors have to burn energy during misses. Secondly, cache nap policies can be effictive, but for high migration frequencies, even when there is an average migration interval of as much as 290K cycles, we should carefully consider the wakeup and flush/flush-invalidate penalties.

5.3.2 Implications of Increasing the C-Core Efficiency

In the previous subsection, the energy model assumed a c-core efficiency from estimations for the Android codebase. Moving forward, however, we would like to know what might be the implications of improved c-core efficiency in the context of the migration system. Figure 5.1 shows diminishing returns, as expected. Figure 5.4 shows these savings broken down by energy component for the sleepy policy.

We see that as the c-core efficiency increases, the L1 access energy behavior becomes increasingly important. At the same time, because the c-cores are more efficient, energy consumed during data cache stalls is less penalizing. Still, it remains a major energy factor for bzip2 and mcf, even out to a c-core efficiency of 20×. Thus, reducing data cache stalls through coherence optimizations has the potential to provide savings even as c-core efficiency increases. Also, notice that as expected, as the c-core efficiency increases, the host energy increases relative to the total energy. The plots show more interestingly, though, that for all of the benchmarks, the host energy approaches almost half of the total energy shortly after passing the GreenDroid c-core efficiency point. The results then indicate that moving forward, improving c-core coverage will provide larger energy reductions than improving c-core efficiency.

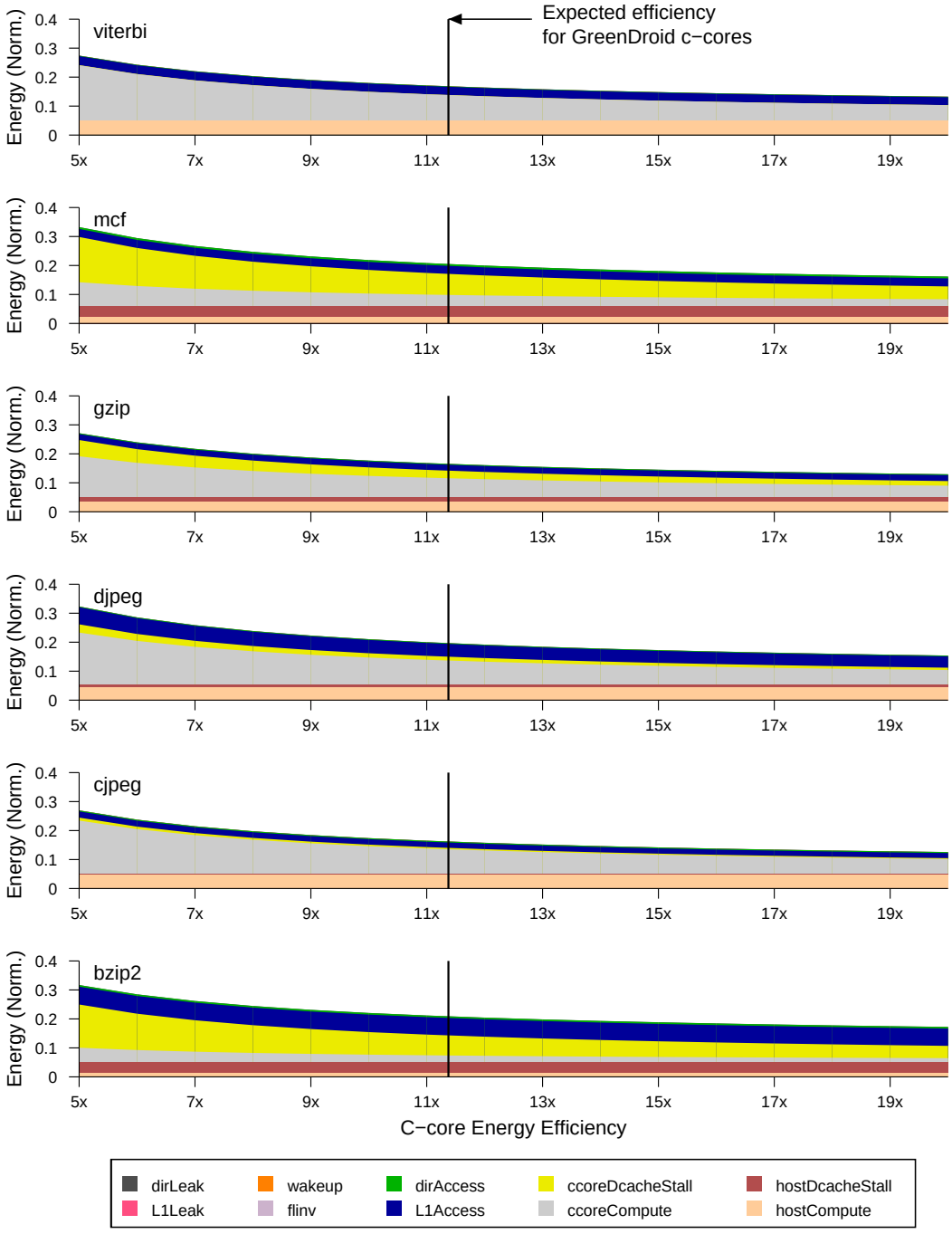

Figure 5.4: Migration System Energy Breakdown vs C-Core Efficiency For each benchmark, the plot shows the energy in the migration system relative to a single-tile system with no c-cores, broken down into components. The results here are for the sleepy nap policy, which tends to be the most efficient.

5.3.3 Justification for a Hardware Flush/Flush-invalidate

In the previous analyses, I assumed that the flush/flush-invalidate routines would run on specialized hardware. In this subsection I compare this configuration to the alternative software approach and quantify the degree to which the hardware-based approach improves the effectiveness of the cache nap policies. More specifically, for each approach I calculate the break-even points for idle time where napping becomes beneficial.

In the nap controller, the tile keeps a "last latency" for each exit point address. When an exit point is reached, if one of the napping policies is enabled and if the last latency is greater than a threshold, the tile moves to a nap state. The current threshold is 20K cycles, based on the typical delay overheads of migration. However, to optimize for energy, the tile should consider a model of the energy overhead of napping relative to staying awake. First, consider the cost of staying awake, E_a. Let *cache* and *dir* be the cache and directory structures, respectively. Let $D_s(a)$ be the dynamic energy of structure s for a accesses, and let $L_s(t)$ be the leakage of structure s over time interval t. Then we have:

$$E_a(t, a_{cache}, a_{dir}) = D_{cache}(a_{cache}) + L_{cache}(t) + D_{dir}(a_{dir}) + L_{dir}(t) \tag{5.1}$$

In sleepy napping, the energy E_s is the cost of the flinv and wakeup components:

$$E_s = E_{flinv} + E_{wakeup} \tag{5.2}$$

In drowsy napping, the energy E_d are these components plus the leakage of the data RAM in the cache in a low-leakage state, plus the dynamic energy and leakage of the tag RAM and directory RAM:

$$E_d(t, a_{tag}, a_{dir}) = E_{flinv}+E_{wakeup}+kL_{data}(t)+D_{tag}(a_{tag})+L_{tag}(t)+D_{dir}(a_{dir})+L_{dir}(t) \tag{5.3}$$

where k is the low-leakage constant, which we conservatively assume to be about 0.25 based on published research on drowsy caches [14].

We would like to know the break-even point, defined by the idle time t, given some assumptions about the number of accesses over that time interval.

Table 5.5: **The Napping Break Even Table** This table shows break-even points for napping based on an analytical model for energy, specified in Equations 5.1, 5.2, and 5.3. Since this model includes parameters that depend on application characteristics, the table displays results for each benchmark based on experimental results. It shows two approaches: one in which the flinv component is implemented in software, and the other in which it is implemented in hardware. There are two insights here: (1) that a hardware flinv mechanism can greatly reduce the threshold of useful napping and (2) that sleepy caches reduce this threshold further compared to drowsy caches.

		Threshold Cycles		
bench	policy	w/ sw flinv	w/ hw flinv	hw threshold reduction
bzip2	drowsy	856440	22221	39×
bzip2	sleepy	767581	16988	45×
cjpeg	drowsy	2298006	68113	34×
cjpeg	sleepy	2225532	51533	43×
djpeg	drowsy	1425745	35350	40×
djpeg	sleepy	1432017	27411	52×
gzip	drowsy	4684531	42385	111×
gzip	sleepy	4060557	32711	124×
mcf	drowsy	918713	37215	25×
mcf	sleepy	2721416	28304	96×
viterbi	drowsy	598025	26003	23×
viterbi	sleepy	477635	20630	23×

Accesses depend on the characteristics of the application; however, we can get a good indication of these characterisitcs from automatic profiling in our toolchain. Using the data from the benchmark runs, then, I derive the implied break even points for the two flush/flush-invalidate approaches: software- and hardware-based. Table 5.5 presents these results.

Notice that firstly, under our energy model, a hardware flinv implementation reduces the breakeven point for napping from 23× - 124×. This expands the set of workloads for which napping can be applied. Moreover, regardless of the flavor of flinv implementation, the sleepy policy generally provides a lower break even

point than the drowsy policy, although by less than an order of magnitude. This is of course expected, given that the sleepy policy power-gates more structures than the drowsy policy.

5.4 Delay

Thus far I have focused my analysis on energy. However, migration obviously presents a delay overhead as well. Figure 5.5 plots the delay impact of migration across the different cache nap policies. The plot breaks down delay into similar components as we saw for energy. I do not include the flush/flush-invalidate routine because it is not part of the critical path. What is clear is that data cache stalls distinguish the delays across the nap policies. The dispatch and wakeup latencies have negligible effect in the current model.

As I discussed in the previous section, the benchmarks which have the highest migration overheads – bzip2, djpeg, and mcf – also have the largest relative increase in data cache stalls when we enable migration. This increase is for the most part invariant across cache nap policies. The djpeg results provide an exception. Recall that the flush/flush-invalidate routines were problematic for energy for djpeg, reducing the savings from the cache nap policies. These routines are similarly troublesome for delay because they can interfere with coherence traffic. When they constitute a larger relative portion of memory traffic, their delay impact increases. Thus, in the djpeg results we see higher delay contribution from data cache stalls under the drowsy and sleepy policies than for the awake policy.

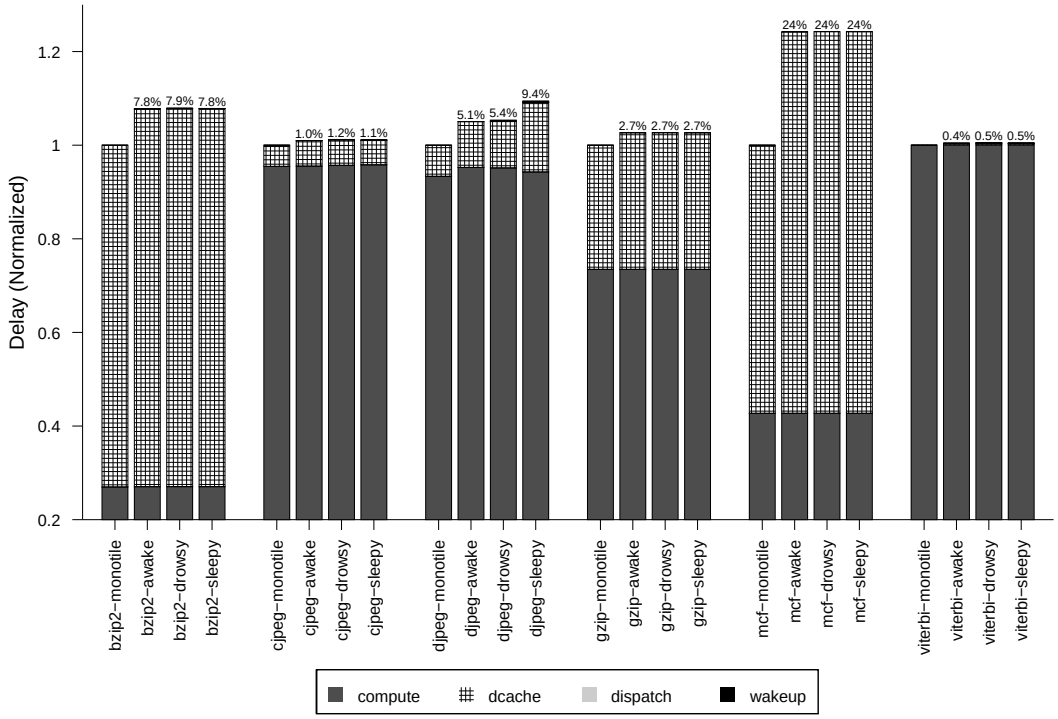

Figure 5.5: **Delay Impact of Migration** This plot has the same configuration as Figure 5.2 – its energy counterpart – except that the vertical axis represents delay. Again, the baseline is the ideal, single-tile case with all c-cores so that there is no need to migrate. The delay break-down includes time during wakeup, dispatch, data cache stalls, and all other computation.

Chapter 6

Related Work

This thesis draws on a body of architecture research across a number of topics. Submarine's design is based on a number of well-studied architectural mechanisms, including task migration, memory system napping, and directory cache coherence. The design targets a new problem, however: migration for exploiting c-cores. In this chapter, I organize related work into three categories: migration, memory system napping, and cache coherence, and explain how previous research both influenced the current Submarine design and may help to implrove it in a future prototype.

6.1 Migration

In this thesis, I studied migration as required for exploiting special-purpose c-cores. A large body of previous work has explored migration for a number of purposes.

Heo et al studied the use of activity migration for reducing peak junction temperature [17]. Their system duplicates architectural components and migrates execution between them. However, their study was limited to two units. The system uses a shared L2 cache to move lines between cores with a constant 2-cycle latency. They propose that a cache-coherence protocol could move lines on demand to the active cache and note that the inactive cache would need to participate in the coherence protocol. The pre-nap flush and flush-invalidate routines in the

Submarine system mitigate and remove this requirement, respectively.

Chakraborty et al proposed computation spreading [18] to partition dissimilar program phases across multiple physical hardware resources. In particular, they studied the separation of a program into its user-level and system-level pieces, so that architectural resources such as caches and branch predictors can target more uniform workloads, improving performance.

Similarly, Kamruzzaman et al recently proposed the use of migration for *software data spreading* [19]. In this technique, threads migrate to utilize cache capacity across cores and consequently improve single-threaded performance. In the GreenDroid toolchain, the aim is to exploit the power efficiency of migrating to remote c-cores, but we also expect to see the cache expansion effects provided by data spreading. As in the GreenDroid system, in software data spreading, migration is compiler-directed. The software data spreading results target a set of modern cache-coherent multiprocessor systems and thus use the coherence subsystem for on demand data transfer. They evaluate both OS-based and user-level-based context switch mechanisms and find that overheads range from 9-14 µs through the OS and 1-3 µs through the user-level. Since the btl-cc prototype system does not run an OS, my context switching system is significantly more lightweight – typically less than 0.1 µs to transfer state over the GDN. However, the future prototype will run an operating system. Kamruzzaman's work thus sheds some interesting light on a possible future challenge, and it remains an open question whether the lightweight GDN-based context migration will be as efficient with an operating system running.

Brown et al propose hardware-based optimizations for thread migration in their Shared-Thread Multiprocessor (STMP) [20]. In STMP, groups of four cores share storage for inactive thread state and a control unit for scheduling and migrating threads. The intended benefits are to increase the amount of thread storage available to each core, to reduce the overhead of migrating or switching between threads, and to allow for thread redistribution through the shared thread controller for higher frequency load balancing and maximizing symbiotic behavior. In this thesis, I considered single-threaded workloads, and therefore the STMP

architecture would not provide direct benefit. However, on a future GreenDroid prototype that requires multi-programming, we might observe prohibitive migration overheads. In this case, it would be interesting to explore a STMP-inspired hardware approach for reducing these overheads.

More broadly, a large body of research investigates the use of inactive compute resources for speculatively improving single-threaded performance. For example, speculative multi-threading [21, 22, 23] divides a running thread into multiple threads in a speculative manner, determining later whether the parallel execution was correct. Similarly, research has shown that helper threads [24, 25] can improve single-threaded performance by speculatively prefetching data in an execution stream outside of the critical path. While these techniques are performance positive, they can be power inefficient because of their speculative approaches. That is, speculation, when wrong, does not hurt performance but consumes unnecessary energy. Still, it would be interesting to see if a similar concept could apply to migration for c-cores.

In this thesis I found that a significant contributor to migration overhead can be the coherence traffic supporting data transfer. Prefetching, either in hardware or via a helper thread, could potentially mitigate this penalty. In fact, Brown et al have studied prefetching for migration performance in detail [26]. Their method predicts a thread's working set and prefetches it into the new cache. Interestingly, they find that prefetching the instruction working set is more important than the data working set. They look at both single-threaded and speculative multithreading workloads. Their single-threaded migration is meant to cover a general class of migrating workloads, migrating at intervals of $1, 10, 100, ..., 10^6$ instructions. This is in contrast to the study in this thesis, where the toolchain inserts migration points based on anticipated program behavior. Also, their primary metric is performance in terms of execution time, while this thesis focuses on energy. Still, their results are useful for motivating future optmizations to the GreenDroid migration system.

Kumar et al introduced using single-ISA heterogeneous processors as a power optimization [27]. Their study modelled migration overhead, but did not

directly explore the underlying migration mechanisms. Shen and Petro proposed a task migration framework for heterogeneous Multiprocessor System-on-Chip (MPSoC) architectures [28]. Their problem is different, however, because they must address the challenge of differing ISAs across processing units. GreenDroid could be considered a heterogeneous MPSoC, given that each tile has a different set of c-cores. However, the ISA is the same on every tile, and the toolchain handles execution on the c-cores. Moreover, the focus of my work is not the programming model or the compiler, but rather the migration mechanisms, especially in the context of data movement. The work of Shen and Petro, on the other hand, focuses on novel ISA extensions and task scheduling.

6.2 Memory System Napping

Concerns about the growing impact of leakage energy has driven research that focuses on leakage savings in the memory hierarchy. Kaxiras et al proposed policies for turning off cache lines that the processor is unlikely to use, a technique called *cache decay* [29]. Their results show that because L1 cachelines often sit unused for some time before being evicted, cache decay can provide significant leakage reduction. While they do explain how cache decay would integrate with a coherence system, they do not study these implications directly.

The *drowsy* cache policy in this thesis is inspired by the work of Flautner et al [14]. Their work focused on reducing cache leakage from so called *cold* lines, or lines with relatively low access rates. Their system puts these lines in a drowsy state, accepting a performance penalty but saving energy.

Ghosh et al proposed *virtual exclusion* [30]. Their observation was that while the trend toward CMP systems with shared memories has made Multi-Level Inclusion (MLI) a common feature in memory hierarchies, MLI is power inefficient because the same memory block is leaking in multiple levels. Cache decay does not work well at higher memory levels like L2 because such systems would not have the MLI property. Drowsy caches allow the system to retain the MLI property but do not achieve the same level of savings as cache decay systems. Virtual exclusion

seeks to get close to the leakage savings of cache decay without losing the MLI property. They use gating to save leakage at L2 and a MOESI coherence protocol to keep track of the location of the valid copies of a block on chip, retaining MLI.

In this study, while I implemented coherent instruction caches, I focused primarily on the energy and performance impact of the data caches. Yang et al presented methods for gating in the instruction cache [31] that are integrated with architectural techniques for adapting to the required instruction cache size.

6.3 Cache Coherence for Tiled Processors

A number of researchers have proposed novel designs of shared memory systems for tiled processors. The canonical problem is the management of Non-uniform Cache Access (NUCA) systems. Typically, each tile is assumed to have a private L1 cache and some portion of a distributed L2 cache. The L2 cache may be private, shared, or some hybrid approach. A common observation is that while private L2 policies offer good hit latency, they fall short of optimal capacity miss rates for the L2, because a tile cannot access unused, remote cachelines. On the other hand, shared L2 policies mitigate the capacity problem, but increase the hit latency when the cacheline resides in a remote portion of the L2.

A number of clever schemes have been proposed to address this tradeoff [32, 33, 34, 35, 36, 37, 38, 39, 40]. The standard approach is to use the coherence system, usually managed by directories for scalability, to allow tiles to share the distributed L2 cache. Moreover, the schemes provide a set of techniques for keeping data in the L2 without sacrificing hit latency for unshared data. Examples include block migration, victim replication, capacity stealing, cache-to-cache transfers, global replacement strategies, and repartitioning. Some systems require OS support to provide hints about the sharing patterns of each page of memory.

Directory storage is an important design point for cache-coherent multicore processors. If two directory entries conflict in some given storage structure, then tiles that have cached lines must invalidate them and write back any dirty data. This can lead to unpredictable performance issues if the directory entry conflicts

happen to occur for heavily used data. To guarantee that this never happens, the directories must have enough storage to track every cacheline in the system. The class of directory storage schemes that take this approach [41] is often called *duplicate-tags*. Usually, the tag indices are divided amongst the directories, and each directory is responsible for tracking the state of the cachelines at those indices. To avoid conflicts, space is allocated for a directory entry for each cache at each tag index. This structure must have an associativity equal to the number of caches. For this reason, energy scalability is a concern. An alternative approach, sometimes called *sparse directories*, uses a sharer vector for each directory entry. However, to avoid conflicts, this structure must have a high storage allocation. Ferdman et al proposed Cuckoo directories [42] to reduce associativity while minimizing directory entry conflicts with multi-stage hashing. Moreover, a number of approaches have been proposed for reducing the storage impact of sharer vectors. It has been observed that in the common case, only some small constant number of caches are actually sharing a particular block and therefore, a more space-efficient solution is possible [43, 44].

Chapter 7

Conclusion

As feature sizes shrink, processor designers will face new challenges to improve performance while meeting fixed power budgets. New solutions will have to emerge to meet these challenges. Innovations could come at the materials, circuit, or architecture levels – or more likely, all of the above. In this thesis I studied a promising architectural approach: migrating tasks to specialized coprocessors to reduce power consumption. I focused on the design of this migration system, with the goal of keeping its energy footprint low. The first-order takeaway from my research is that efficient migration is feasible and practical for tiled coprocessor-based architectures like GreenDroid. My cycle-accurate simulation results show low overheads and large savings as c-core efficiencies increase.

I also found that the key element in such a migration system is the shared memory hierarchy; however, it is not the power of the memory subsystem itself that is most important. My results indicate that in fact, a key factor is the performance of the coherence system during migration. Slow data transfer forces the processors and c-cores to stall and waste power. Still, the memory system contributes to the power consumption as well, and because of this I have introduced and analyzed techniques for reducing this consumption under a migrating workload by power-gating memory system components. Results indicate that drowsy and sleepy cache nap policies may be able to mitigate migration overhead to the extent that the energy footprint of the pre-nap flush and flush-invalidate routines can be minimized.

Moving forward with the development of the GreenDroid prototype, we will need to consider the components of migration overhead. Given that in the era of dark silicon area is cheap but power is expensive, it is likely that hardware-based solutions for efficient migration will be attractive. My proposed hardware-based flush-invalidate state machine is one example. As we move to support multi-programmed workloads, there will be additional challenges. A more sophisticated migration system will need to handle sharing of c-cores between tasks, which will increase the resources needed to save and migrate task state. Again, it is likely that a future prototype would use specialized hardware to meet these requirements with acceptable power efficiency. Moreover, in the current prototype I study the common case where the set of migration group tiles is fixed at compile time. For general purpose execution, I assume that the coherence system will switch to a less efficient but more general directory controller, following the c-core/software paradigm for exploiting dark silicon. A future prototype will need to explore this idea further.

Finally, the work presented in this thesis will help our group design and implement the GreenDroid prototype memory and migration subsystems. My cycle-accurate simulation infrastructure specifies the redesigned L1 data- and instruction-cache controllers. We can use these designs in the next FPGA-emulated prototype. Moreover, my bC directory model can run alongside the emulation system. I have designed it to be general purpose in order to consider various directory storage schemes. Also, my migration system, Submarine, can contribute to the future GreenDroid runtime. It is my hope that my work will both motivate and contribute to the fabrication of a future GreenDroid chip.

Appendix A

Implementation Details

The implementation of cache coherence systems is a challenging engineering problem. In this research, I model GreenDroid and similar tiled architectures using the Raw cycle-accurate simulator, btl. However, the Raw system does not support cache coherence. Moreover, the programming model is very much "bare metal" – the intention is to expose the underlying architectural resources of the chip to the programmer. Therefore, as a research prototype, system support for shared memory programming constructs is not a priority. As a result, as part of this research I evolved the Raw toolchain at several levels, which can serve as a case study for upgrading systems for shared memory multi-threading.

In this appendix chapter, I detail the engineering problems that I encountered during the course of my research. Firstly, I discuss the redesign of the memory subsystem, including new data- and instruction-cache controllers, directories, and some minor changes to the DRAM controller. I then specify additions and revisions to the Raw ISA for the purpose of providing cache coherence. I also describe the new library support for synchronization and threading mechanisms, as well modifications to the C Standard Libraries for thread-safety. Finally, I note the implementation details for the migration and napping systems that I modelled in btl-cc for this research.

A.1 Redesigning the Raw Memory Subsystem for Cache Coherence

Raw [15] is a tiled processor developed at MIT that was fabricated into a 16-tile chip in 2002. The Raw simulator, btl, is the basis for btl-cc, the simulator used in this research. The btl simulator was validated against the Raw RTL. The chip model is written in C++. Btl also provides a user interface for prototyping new devices using a multithreaded, bytecode compiled extension language called bC.

Implementation of architectural designs in a simulation environment provides an infrastructure for both specifying and evaluating designs. In this section I discuss the major components of the memory subsystem: the L1 caches, the directories, and the DRAMs, and explain how I designed and implemented each piece for btl-cc.

A.1.1 The L1 Caches

An early challenge introduced in the new system is that caches receive and must service unsolicited requests, even while in the middle of a cache miss. This requirement is rooted in protocol deadlock avoidance. In the coherence system, forward progress of a transaction may depend on forward progress of remote requests. The protocol could deadlock if misses can block incoming remote requests. Consider an example: suppose a miss in cache x depends on a remote request at cache y, while a miss at cache y depends on a remote request at cache x. In this case, if the misses block the remote requests, the protocol will deadlock.

To allow the caches to service remote requests, I made two major changes to the tile design. First, I implemented a more flexible scheme for arbitrating the MDN between the data cache, instruction cache, and user program. Second, I redesigned the cache state machines. I will now take a moment to detail these changes.

In the previous Raw system, caches only receive solicited requests, making arbitration for the MDN somewhat straight forward. A cache can lock the MDN

while waiting for a miss. Similarly, the user can lock the MDN using the mlk instruction, which also pulls in some number of instruction cache lines to make sure that the instruction cache will not miss while the MDN is locked. There will be no need to use the MDN in the mean time, and the miss will not block any other local request, because the MDN is locked. For the new system, I implemented a more sophisticated arbitration scheme, shown in Figure A.1. The caches can only hold the lock when forward progress in the state machine is guaranteed. The mlk and munlk instructions have no effect. Instead, the tile buffers user packets in the UMNO. When a complete packet is ready, the tile locks the MDN and sends the packet. This design allows me to run legacy code that uses the MDN at the user level.

Similarly, I modified the tile to arbitrate incoming messages from the MDN. This does not require any locking; instead, a simple module inspects the opcodes of incoming headers and routes the packets to the correct input buffers: either the DMNI, IMNI, or UMNI, for the data cache, instruction cache and user, respectively.

Figure A.2 shows the data cache state machine. The major modification necessary was the ability to service remote requests. Therefore, I added a `FreeWait` state for waiting during misses. The state machine is design so that each state has exactly one of two properties:

1. Active - guaranteed to make progress to the next state in some finite period of time

2. Free - available to accept remote requests

The `START` and `FREE_WAIT` states have the Free property, while all others have the Active property. While Active, the state machine may be waiting on one of two things:

1. the MDN output port

2. the MDN arb to grant the lock to write to the MDN output port

The MDN output port becomes free when there is space in the input buffer to the local MDN router. As long as the network is not deadlocked, the router is

guaranteed to make progress, and therefore the MDN output port will eventually become free.

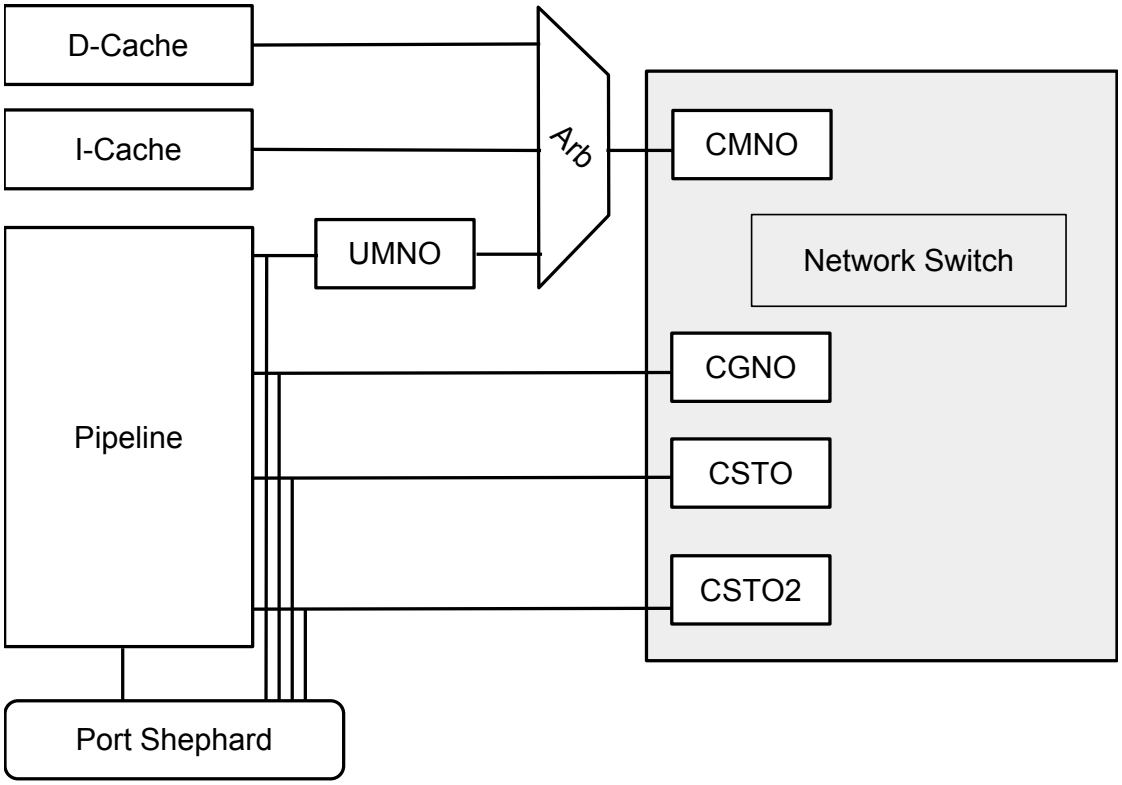

Figure A.1: **Arbitration of Outgoing MDN Packets** The MDN is shared by three clients on the tile: the d-cache, i-cache, and user system. The user system packets are buffered for store-and-forward atomicity. All three MDN clients contend for an MDN lock through the arb. The Port Shephard manages flow control between the user system (i.e. the pipeline) and the user-level network ports. The UMNO port is actually invisible to the user. The user sees the $cmno register as specified by the Raw ISA. The figure shows the static network ports, CSTO and CSTO2, but experiments in this study do not use the static networks.

The clients that compete for the lock to write to the MDN output port are the data cache, the instruction cache, and the user program. If a client holds the lock, the only condition where it will be blocked is if the MDN port is not free. Again, assuming no deadlock, this port will eventually become free and thus the

client will eventually give up the lock.

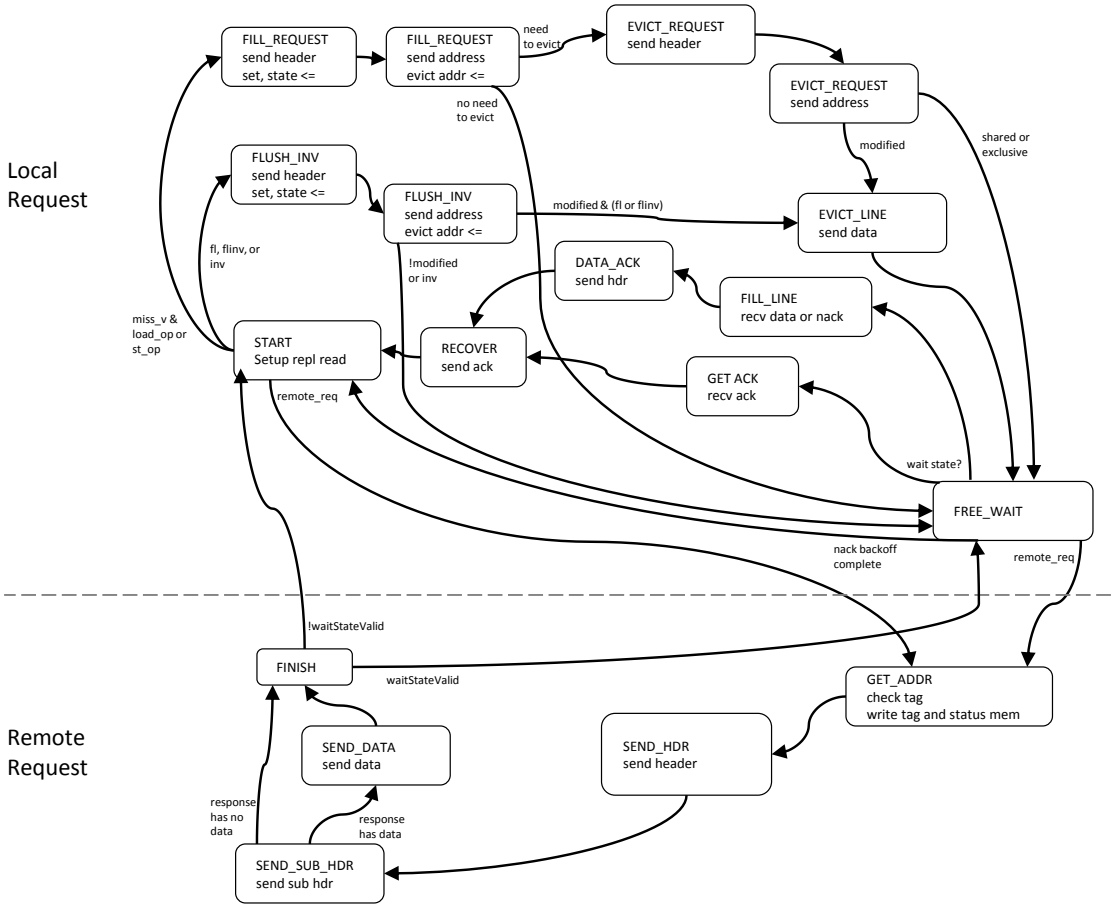

Figure A.2: **The Coherent Data Cache State Machine** The data cache is able to accept both local and remote requests. When in the `START` and `FREE_WAIT` states the state machine is able to begin servicing a remote request. At the end of the remote request, the state machine returns to one of those two states. In this state diagram, if there are multiple outgoing edges from a state, edge annotations describe the conditions for the respective state transition. The `FINISH` state in the Remote Request partition is actually absorbed into both the `SEND_SUB_HDR` and `SEND_DATA` states, and does not consume an extra cycle. In general, however, all state transitions consume at least one cycle.

Therefore, we see that assuming no deadlock on the MDN, the Active states will always make forward progress. The FreeWait states, on the other hand, stall

until the cache receives either a local request or a remote request. This means that as long as there is no deadlock on the MDN, the cache will always be able to service requests in some finite amount of time. In Chapter 3 I showed how buffering at the MDN clients prevents MDN deadlock by allowing the clients to sink any possible incoming messages, regardless of the state of their controllers. Therefore, the MDN is guaranteed to become clear in a finite amount of time, and allow Active states to make forward progress. Also recall that a requirement for holding the MDN lock is that the client is guaranteed to make forward progress. Clearly, then, the caches can only hold the lock when in Active states and must release the lock before moving to Free states.

The instruction caches are also coherent. This, in theory, allows the system to support writing over instruction memory or even self-modifying code, although neither of these use cases were focuses of my research. I implemented coherent instruction caches because a future GreenDroid prototype will need this feature to support a modern operating system, and I wanted to model the delay impact of coherent instruction cache missing and to verify that my protocol and directory scheme were amenable to instruction cache coherence. The instruction cache state machine is similar to the data cache state machine, except the instruction cache does not need to flush or service remote flush requests.

I modelled all L1 designs with a hardware implementation in mind and followed btl conventions for timing. For example, I used btl FIFO models when necessary and used the existing infrastructure for the state machines, tags, data, etc.

A.1.2 The Directory Model

The directory is modelled in bC. For purposes of simulation, it sits off chip and interfaces with the chip I/O ports. In a real implementation, the directories would be on chip. Implementing the directory in bC has two primary advantages. First, it allows for rapid prototyping. Second, the GreenDroid group is currently building an emulation infrastructure that allows for bC devices to run concurrently with either FPGA emulation or a hardware implementation of the chip. A bC

model thus allows us to continue research on the directory design, even if the chip design is in a later stage of development.

In my implementation, I logically divide the directory model into a number of components:

- Storage model
- Controller
- CMNI FIFO
- CMNO FIFO
- Pending request file
- MDN arbiter

The storage model is generalized. Currently it assumes that there are never conflicts, which is the case for the current duplicate-tags directory design. Instead of directly modeling the one-way-per-tile system, however, the simulator code uses sharer vectors for ease of implementation. This still simulates the intended behavior of the system.

The directory controller is implemented as a state machine in a similar structure to the cache state machines. It consumes incoming words from the CMNI FIFO no faster than one word per cycle and places outgoing words on the CMNO as long as there is space. Otherwise, the controller stalls.

The CMNI FIFO is not flow controlled because it has the provably necessary allocation as described in Section 3.3. The CMNO FIFO is flow controlled and exists to decouple the controller and MDN arbiter.

The MDN arbiter is implemented as a simple two-state state machine that can consume one word per cycle. There are two producer clients that interface with the MDN arbiter: the CMNO FIFO and the DRAM Virtual Port. In the first state, `kMDNArbGetHeader`, the MDN arbiter peeks the two producer clients, picks the next input based on a round-robin policy, pulls the length from the header, and places the header on the MDN port if there is room. If the candidate

producer is the CMNO FIFO, it makes sure that the complete packet is ready. The next state is `kMDNArbGetData`. In this state the payload of the packet moves through the MDN arbiter to the MDN port at a rate of one word per cycle. When the MDN arbiter has finished moving the packet, it returns to `kMDNArbGetHeader`.

A.1.3 The DRAM Controller

The DRAM controller was largely unmodified, with two exceptions. First, in the Raw system the DRAM interfaces with an I/O port of the chip. In my model, the directory interfaces with a physical I/O port and then provides a virtual port to the DRAM. Second, the DRAM must recognize an extended set of opcodes. In the Raw system, there is a single cacheline read opcode, but in btl-cc, there are three: cacheline read from an i-cache, cacheline read for shared access from a d-cache, and cacheline read for exclusive access from a d-cache. The DRAM treats all of these opcodes in the same way, but copies them to the header of the response so that the tile can interpret the packet correctly.

A.2 The Cache-Coherent Raw Instruction Set Architecture

From an engineering perspective, one of the biggest challenges of supporting multi-threading is providing locking mechanisms and cache-coherent I/O. The former requires atomic memory access. The latter, cache-coherent I/O, requires well-defined flushing and invalidation mechanisms. For efficiency, both of these upgrades are best addressed at the ISA-level. In this section I discuss these design decisions and implementation details.

A.2.1 Load-linked and Store-Conditional

Locking is a key requirement to support standard shared memory programming models. For efficiency, shared memory architectures typically provide some atomic memory primitives from which the system can provide locking mechanisms.

To this end, I added load-linked (`ll`) and store-conditional (`sc`) instructions to the ISA. My implementation is based on the design specified for the MIPS R4000 architecture [7]. The semantics of these instructions are as follows. A store-conditional is a special store instruction that only commits its store if no tile in the system has modified the block at the store address since the most recent call to load-linked on the local tile. The store conditional places its success code (1 on success, 0 on failure), in the destination register. A load-linked operates like a normal load, with two exceptions: (1) the pipeline stores its cacheline-aligned address in a special register for comparing against remote modifications to the block and (2) on a miss, a load-linked requests exclusive access instead of just shared access. A load-linked call, in practice, is followed in some short period of time by a store-conditional call on the same tile to the same address. Requesting exclusive access is an optimization based on this common case.

To support the load-linked and store-conditional instructions, I designed a Load-Linked Store-Conditional (LLSC) Unit for the tile. The LLSC Unit includes the special register for the last load-linked address and a register for flagging that some tile has modified the block. After a load-linked instruction, the local cache will have exclusive access to the line and therefore will receive an invalidation message from the directory if any other cache attempts to modify the block. The cache signals the LLSC Unit upon receiving invalidation messages for comparison against the current load-linked address. On a match, the modified flag is set. When a store-conditional instruction is in the memory stage of the pipeline, it checks the modified flag – if the block is unmodified and the store is a hit in the local cache, the store-conditional succeeds. If the line is modified, it fails. If the line is unmodified but the store misses in the cache, the cache will issue a normal exclusive request, continuing to watch for invalidations. This is an atypical case though, because it means that the some time between the load-linked and the store-conditional the local tile issued a memory instruction that caused the eviction of the load-linked line – placing memory instructions between `ll` and `sc` is discouraged.

A.2.2 Invalidate, Flush, and Flush-Invalidate

Standard I/O libraries typically make extensive use of flushing and invalidation, sometimes categorized as *cache administrative instructions*. While the focus of this research was not the I/O system, there are two important reasons to provide real support for I/O in the cache coherence system. Firstly, most standard benchmarks use standard I/O functions. Removing I/O from these benchmarks is not just laborious and impractical, it also detracts from the comparability of my results to other experiments using standard benchmark suites. Secondly, cache-coherent invalidations, flushes, and flush-invalidates introduce a number of complications to the coherence system, that, if overlooked, could lead to an over-simplified design and inaccurate modeling of a future, "real" system.

The Raw system supports a significant subset of the C standard library through a port of newlib [45]. The newlib port makes calls to both address-based and tag-based cache administrative instructions. These calls introduce a number of engineering issues. In the Raw system, the address-based instructions `ainv`, `afl`, and `aflinv`, invalidate, flush, and flush-invalidate a cacheline based on the given address. Since Raw is not cache-coherent, these instructions only deal with the local cache – on a miss, no further action is necessary. However in a cache coherence system, the semantics become more challenging. First of all, the directory needs to track the location of memory blocks throughout the system, so a tile must notify the directory whenever it invalidates a line. Moreover, to support execution migration, all cache administration needs to be global by default, because the migration is hidden from the programmer. For example, consider a program that writes to a buffer and then flushes that buffer to some output device. If execution migrates between the buffer write and flush, the dirty data will likely sit in a remote cache at the time of the flush. Therefore, the flush needs to be global for correct execution. For this reason, in btl-cc, the `ainv`, `afl`, and `aflinv` instructions are global by default.

Still, I quickly found use for local cache administration for cache-napping purposes. Before sleeping a cache, a tile must flush-invalidate all lines, but would not want the request to apply to all caches. For this reason, I added `afll` and

`aflinvl` to the ISA. There is currently no support for a local invalidation instruction, because it was not necessary for this research; however, it would fit naturally into the existing support infrastructure for `afll` and `aflinvl` and is scheduled for future work.

The tag-based cache administrative instruction `tagsw` in the Raw system allows the user to modify the tags arbitrarily, including the dirty and valid bits. This has an undefined impact on the coherence state of the system; therefore, btl-cc does not support `tagsw`. In order to support legacy code, I replaced any use of `tagsw` in the Raw libraries with an alternate implementation. Typically, programs use `tagsw` to invalidate cachelines after flushing. The instruction does not need the full memory address – only the tag index – and thus it is efficient when invalidating large cache ranges. In most cases I was able to replace the call with either `ainv` or `aflinv`. However, since these instructions require the full memory address, I had to insert a call to `tagla`, which loads the cacheline memory address based on the index into the tag array. This call to `tagla` sacrifices a cycle of performance in the best case. A future system would likely include coherent tag-based cache administrative instructions to mitigate this unnecessary latency.

Additionally, as described in Section A.1.1, the mlk and mulk instructions have no effect and are essentially nops in btl-cc.

A.3 Library Design for Cache Coherence and Shared Memory Multi-threading

For the purposes of this research, I encountered some of the major issues at the OS/library level for shared memory systems. I present my solutions in this section. These design decisions were motivated by the focus of the research project, and thus do not provide the full set of services that would be necessary for a complete system. However, my sense is that these approaches address, conceptually, the main challenges.

A.3.1 Synchronization

Most shared memory multi-threaded programs make use of synchronization mechanisms, the most common of which are locking and barriers. I implemented lock and unlock in assembly using load-linked and store-conditional. Listing A.1 shows the lock routine. The use of these instructions ensures that the increment of the lock variable is atomic. If another tile modifies the lock between the load-linked and store-conditional, the store-conditional will fail. The routine is surrounded conservatively by memory barriers to protect against compiler optimizations that might violate assumptions about synchronization. The unlock function sets the lock value to zero, again surrounded by memory barriers.

Listing A.1: **Acquire Lock** The lock routine load-linkeds the lock until it is non-zero. Then it tries to store a one in the lock using store-conditional. If the `sc` fails, it iterates; otherwise, the lock is acquired. Note that the current Raw assembler does not support `ll` and `sc`. Instead, I use macros that explicitly place the machine code for the instructions as data words in the assembly. In this listing I changed the macros to assembler instructions for readability.

```
acquire_lock:
        la $8,lockaddr
loop:
        ll $9, $8
        bne $9, $0, loop
        addiu $9, $9, 1
        sc $10, $9, $8
        beq $10, $0, loop
```

From the locking routines, then, I implemented a simple shared memory barrier routine, provided in Listing A.2. The barrier data structure includes a lock and a counter. When a tile encounters the barrier, it acquires the lock and increments the counter. When the counter value indicates that all tiles have reached the barrier, the tile multicasts a wakeup message on the GDN to all waiting tiles.

A.3.2 The C Standard Library

The Raw toolchain uses a port of newlib [45] for C standard library functions. While this port is not thread safe, newlib provides a nice mechanism for adding locking and for building a thread safe version of the library. I defined the newlib locking macros with calls to the new locking routines. I also modified some of the global data structures to allow each tile to have an individual errno value.

A limitation of the current malloc implementation is that all threads that share the same malloc code pull from the same pool of available memory without any attention to smart mappings based on the relative DRAM location. Since much of my results are based on 4- and 8-tile configurations, this was not an issue. However, for scalable implementations, one would likely want to map malloc pools to optimal physical DRAM locations. This would require modification of the sbrk and malloc modules in newlib.

A.3.3 Multi-Threading and PARMACS Support

The SPLASH2 benchmark suite uses PARMACS [46] as a portable abstraction layer for multi-threading. For example, a linux system can define the PARMACS macros in terms of pthreads calls. However, the Raw system does not have a standard set of multi-threading functions. To get SPLASH2 running, then, I built a simple thread dispatching library.

The thread dispatching for PARMACS works as follows. At initialization time, the root tile broadcasts its global pointer over the GDN to all worker tiles. The worker tiles then overwrite their global pointers with this new address. Thus, all tiles in this multi-threading group share the same address space, and can be thought of as a group of tiles running the same process. The worker tiles then block waiting for a thread dispatch on the GDN. On thread creation, the root tile sends a function pointer, also known as a dispatch address, to the worker tile. The worker tile receives this packet and jumps to the dispatch address. The workers send acknowledgments to the root tile when they are done, emulating a "join" functionality as may be familiar to pthreads programmers.

A.3.4 HWIC Trampoline

The Raw chip has hardware instruction caching for both the host processor and the static switch processor. However, in my research I did not consider the static networks, as it is anticipated that GreenDroid will not have static on-chip networks. However, the existing codebase for activating hardware instruction caching, the *HWIC Trampoline*, also activates the switch instruction cache. I made some minor modifications to this code so that there would be no unsupported cache miss messages from the static switch instruction caches.

A.4 Migration

Chapter 2 gives an overview of the migration primitives. I implemented the `migrate_push` and `migrate_pop` routines using a mix of C and assembly. I also wrote a simple preprocessor that takes code that the toolchain has annotated with region mappings and inserts migration calls. I also provide an assembly routine for inactive tiles that are waiting for dispatch packets on the GDN.

Moreover, the previous newlib port does not support process migration. To date I have discovered and fixed one issue. Because the implementation assumes that processes will not migrate, the library precomputes network headers destined for the host interface (in this case *host* refers to the platform host, not the tile host processor). After a migration, the source coordinates in these network headers will be wrong. To solve this, I added a routine `init_for_new_tile` that updates the `hosthdr` global variable with the new source coordinates after every migration. In fact, you can see the jump to this routine in the migrate example code in Listing 2.1.

A.5 Napping

As described in Chapter 4, caches can nap to save power. The tile does two things before napping the cache. First, it notifies the nap controller. In my implementation, the host processor interfaces with the nap controller through a

set of magic instructions. The instructions, listed below, allow the nap controller to train its predictor and update event counters for analysis.

- kMagicMigrateLogExit - notifies the nap controller of an exit
- kMagicMigrateNapAsleep - notifies the nap controller of a sleepy nap
- kMagicMigrateNapDrowsy - notifies the nap controller of a drowsy nap
- kMagicMigrateWakeup - notifies the nap controller of a wakeup
- kMagicMigrateRecvDispatch - notifies the nap controller of a new dispatch

The btl simulator supports the addition of magic instructions which can hook to bC code for flexible functionality without fundamentally altering the instruction pipeline or tile architecture. For a description of the functionality of Raw magic instructions, see the Raw specification [5].

Second, the tile must flush or flush-invalidate the entire cache before putting it in a drowsy or sleepy state, respectively. These routines make use of the local flush (`afll`) and flush-invalidate (`aflinvl`) instructions.

For the same reasons as the directory, I implemented the Nap controller in bC. Also, it is conceivable that the nap controller could run as a software library, and I wanted to retain the flexibility to model either a software or hardware implementation. I described the functionality of the nap controller in Chapter 4.

Listing A.2: **Shared Memory Barrier** Tiles in the barrier group increment a shared counter. A lock protects against race conditions during the increment. After incrementing the counter, the thread waits for a wakeup message on the GDN. The last thread to hit the barrier broadcasts the wakeup message.

```
void
barrier(barrier_t* b, unsigned int threads)
{
  int i, myID;
  int count;

  lock_t* lock_ptr = &(b->lock);

  acquire_lock(lock_ptr);
  count = ++(b->counter);

  if (count < threads) {
    release_lock(lock_ptr);
    // wait for barrier wakeup signal
    wait_gdn_msg(BARRIER_WAKEUP);
  } else {
    b->counter = 0;
    release_lock(lock_ptr);
    // wake everyone up
    myID = raw_get_tile_num();
    for (i = 0; i < threads; i++) {
      if (i != myID) {
        gdn_send_1w(BARRIER_WAKEUP, i);
      }
    }
  }
}
```

Bibliography

[1] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor, "The greendroid mobile application processor: An architecture for silicon's dark future," *Micro, IEEE*, vol. 31, pp. 86 –95, march-april 2011.

[2] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: reducing the energy of mature computations," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, (New York, NY, USA), pp. 205–218, ACM, 2010.

[3] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams," in *ISCA '04: Proceedings of the 31st annual International Symposium on Computer Architecture*, p. 2, IEEE Computer Society, 2004.

[4] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, (New York, NY, USA), pp. 24–36, ACM, 1995.

[5] M. B. Taylor, "The raw processor specification," 2003. `http://groups.csail.mit.edu/cag/raw/documents/`.

[6] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *Proceedings of the 11th annual international symposium on Computer architecture*, ISCA '84, (New York, NY, USA), pp. 348–354, ACM, 1984.

[7] J. Heinrich, *MIPS R4000 Microprocessor User's Manual*. MIPS Technologies, 1994.

[8] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 169 –180, feb. 2011.

[9] SPEC, "SPEC CPU 2000 benchmark specifications," 2000. SPEC2000 Benchmark Release.

[10] Embedded Microprocessor Benchmark Consortium, "Eembc benchmark suite." http://www.eembc.org.

[11] Independent JPEG Group, "Library for jpeg image compression." http://www.ijg.org/.

[12] A. Agarwal, R. Bianchini, D. Chaiken, F. Chong, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The mit alewife machine," *Proceedings of the IEEE*, vol. 87, pp. 430 –444, mar 1999.

[13] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy, "The dash prototype: Logic overhead and performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 41–61, 1993.

[14] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: simple techniques for reducing leakage power," in *In Proceedings of the 29th Annual International Symposium on Computer Architecture*, pp. 148–157, 2002.

[15] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, pp. 25–35, March 2002.

[16] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti 5.1," Tech. Rep. HPL-2008-20, HP Labs, Palo Alto, 2008.

[17] S. Heo, K. Barr, and K. Asanovic, "Reducing power density through activity migration," in *In Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 217–222, 2003.

[18] K. Chakraborty, P. M. Wells, and G. S. Sohi, "Computation spreading: employing hardware migration to specialize cmp cores on-the-fly," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, (New York, NY, USA), pp. 283–292, ACM, 2006.

[19] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, "Software data spreading: leveraging distributed caches to improve single thread performance," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, (New York, NY, USA), pp. 460–470, ACM, 2010.

[20] J. A. Brown and D. M. Tullsen, "The shared-thread multiprocessor," in *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, (New York, NY, USA), pp. 73–82, ACM, 2008.

[21] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, (New York, NY, USA), pp. 414–425, ACM, 1995.

[22] P. Marcuello, A. González, and J. Tubella, "Speculative multithreaded processors," in *Proceedings of the 12th international conference on Supercomputing*, ICS '98, (New York, NY, USA), pp. 77–84, ACM, 1998.

[23] V. Krishnan and J. Torrellas, "A chip-multiprocessor architecture with speculative multithreading," *Computers, IEEE Transactions on*, vol. 48, pp. 866 –880, sep 1999.

[24] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "Simultaneous subordinate microthreading (ssmt)," in *Proceedings of the 26th annual international symposium on Computer architecture*, ISCA '99, (Washington, DC, USA), pp. 186–195, IEEE Computer Society, 1999.

[25] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: long-range prefetching of delinquent loads," in *Proceedings of the 28th annual international symposium on Computer architecture*, ISCA '01, (New York, NY, USA), pp. 14–25, ACM, 2001.

[26] J. Brown, L. Porter, and D. Tullsen, "Fast thread migration via cache working set prediction," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 193 –204, feb. 2011.

[27] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen, "Single-isa heterogeneous multi-core architectures: The potential for processor power reduction," 2003.

[28] H. Shen and F. Petrot, "Novel task migration framework on configurable heterogeneous mpsoc platforms," in *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, pp. 733 –738, jan. 2009.

[29] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: exploiting generational behavior to reduce cache leakage power," in *Proceedings of the 28th annual international symposium on Computer architecture*, ISCA '01, (New York, NY, USA), pp. 240–251, ACM, 2001.

[30] M. Ghosh and H.-H. Lee, "Virtual exclusion: An architectural approach to reducing leakage energy in caches for multiprocessor systems," in *Parallel and Distributed Systems, 2007 International Conference on*, vol. 2, pp. 1 –8, dec. 2007.

[31] S.-H. Yang, M. D. Powell, B. Falsafi, K. Roy, and T. N. Vijaykumar, "An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches," pp. 147–157, 2001.

[32] B. M. Beckmann and D. A. Wood, "Managing wire delay in large chip-multiprocessor caches," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, (Washington, DC, USA), pp. 319–330, IEEE Computer Society, 2004.

[33] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *In Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pp. 336–345, 2005.

[34] J. Chang and G. S. Sohi, "Cooperative caching for chip multiprocessors," in *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, (Washington, DC, USA), pp. 264–276, IEEE Computer Society, 2006.

[35] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive nuca: near-optimal block placement and replication in distributed caches," in *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 184–195, ACM, 2009.

[36] J. Lira, C. Molina, and A. González, "The auction: optimizing banks usage in non-uniform cache architectures," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, (New York, NY, USA), pp. 37–47, ACM, 2010.

[37] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian, "Swel: hardware cache coherence protocols to map shared data onto shared caches," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, (New York, NY, USA), pp. 465–476, ACM, 2010.

[38] E. Herrero, J. González, and R. Canal, "Elastic cooperative caching: an autonomous dynamically adaptive memory hierarchy for chip multiprocessors," in *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, (New York, NY, USA), pp. 419–428, ACM, 2010.

[39] M. Hammoud, S. Cho, and R. G. Melhem, "Cache equalizer: a placement mechanism for chip multiprocessor distributed shared caches," in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, (New York, NY, USA), pp. 177–186, ACM, 2011.

[40] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Optimizing replication, communication, and capacity allocation in cmps," in *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, (Washington, DC, USA), pp. 357–368, IEEE Computer Society, 2005.

[41] B. W. O'Krafka and A. R. Newton, "An empirical evaluation of two memory-efficient directory methods," in *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, (New York, NY, USA), pp. 138–147, ACM, 1990.

[42] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 169 –180, feb. 2011.

[43] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A tagless coherence directory," in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, (New York, NY, USA), pp. 423–434, ACM, 2009.

[44] H. Zhao, A. Shriraman, and S. Dwarkadas, "Space: sharing pattern-based directory coherence for multicore scalability," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, (New York, NY, USA), pp. 135–146, ACM, 2010.

[45] J. Johnston, "Newlib." http://sourceware.org/newlib/.

[46] E. Lusk, J. Boyle, R. Butler, T. Disz, B. Glickfeld, R. Overbeek, J. Patterson, and R. Stevens, *Portable programs for parallel processors*. Austin, TX, USA: Holt, Rinehart & Winston, 1988.