Data Movement Control for the PowerPC Architecture

Silas Boyd-Wickizer and Asif Khan

1 Introduction

There is an inherent cost for applications to access off-chip DRAM. A potential solution is a single large on-chip cache that all cores access with uniform latency. This architecture makes it easy for application developers to implement efficient inter-core sharing and use the entire on-chip cache. A large shared on-chip cache, however, is still prohibitively slow. Architects ensure each core has fast access to some portion of on-chip memory by distributing on-chip memory in pieces so that every core is near some cache. In theory this provides a large amount of aggregate cache capacity and fast memory for each core. Unfortunately, it is more difficult for software to use a distributed cache effectively than a shared cache effectively. The goal of this project is to explore whether extensions to hardware can help software make better use of distributed caches on multicore processors.

Consider some of challenges faced by software trying to use a distributed cache on a multicore processor. In some architectures, an application can cache data only in the caches of the cores it's currently executing on. This provides applications with access to only a small amount of on-chip cache capacity. Even if the application is executing on all cores, it is expensive to accesses data in a remote core's cache, and it's likely that each core's individual caches would end up caching the same commonly accessed data. Duplicating data reduces the number of distinct data items cached on-chip, which essentially reduces the effective cache capacity.

Promising software solutions (*e.g.* [6, 10]) use thread migration to help manage cache contents. The basic idea behind these solutions it to assign data items to on-chip caches and migrate threads amongst the caches as they access the data items. Moving a thread closer to the data it accesses reduces access latencies and helps ensure that the same data is not duplicated many times. The implementations of these solutions, however, can have significant overheads. Migrating a thread can require as many as 20,000 cycles [11]. Distributing data items to caches requires software to track which data items it assigns to which caches, adding overhead and essentially duplicating directories maintained by hardware. Even if software is able to efficiently manage mappings from data items to caches, it can only guess if a data item is actually cached or or has been evicted by hardware.

This project explores the opportunity to extend hardware to make it easier for applications to use on-chip caches efficiently and thereby improve performance. We introduce a set of hardware extensions, which we refer to as Data Movement Control (or DMC), that are in the form of three new instructions: `cpush`, `clookup`, and `cmsg`. The instructions give software more information about and control over on-chip cache contents. `cpush` allows a thread running on one core to move cache lines into another core's cache; `clookup` returns the location of any cache line; and `cmsg` provides an efficient mechanism for software to send an active message [12]

to a remote core, which allows a thread to efficiently manipulate data in a remote core's cache. Collectively, these instructions address some of the shortcomings of previous software-only cache management solutions.

To evaluate potential performance improvements we implemented the DMC instructions in an existing cache-coherent multicore Book-E PowerPC implementation. The resulting DMC PowerPC microarchitecture is backwards compatible with the Book-E PowerPC microarchitecture. The original implementation, as well as the DMC implementation, provide cycle accurate processor timing when synthesized for on BEE3 FPGA system. Results from running synthetic benchmarks on the DMC BEE3 core indicate that using DMC instructions can improve the performance of operations that manipulate as few as two shared cache lines.

A main challenge in implementing DMC is doing so without causing deadlocks or invalid states in the cache coherence controller. The implementation of `cpush` is particularly tricky because the cache coherence controller must handle race conditions where a core requests a particular cache line in a particular mode (*e.g.* M), while another core simultaneously pushes the same cache line in a different mode (*e.g.* S). Another challenge is implementing the DMC extensions so that they are compatible with existing PowerPC applications, yet still provide high performance. For example, when a remote core begins executing an active message it must not violate the PowerPC ABI, which mandates that software restore all the execution state (*e.g.* register values) when the active message completes. If software saved and restored all execution state, however, active messages would be prohibitively expensive.

The main contributions of this project are: (1) the introduction of the DMC hardware primitives that simplify software cache management; (2) a new type of active message that is addressed by memory address instead of destination core; and (3) an implementation and evaluation of DMC hardware using synthetic benchmarks.

The rest of this report is organized as follows. Section 2 briefly discusses some related work. Section 3 describes the interface and semantics of the DMC instructions and Section 4 describes their design and the solutions to the design challenges. Section 5 discusses the implementation of the DMC instructions and our testing procedure. Section 6 describes preliminary results from apply DMC instructions to microbenchmarks, Section 7 discusses limitations of this work and directions for future research, and Section 8 concludes.

2 Related work

There is a significant amount of work related to multicore cache management and computation migration. The following section describes a few examples from each category, but it is not an exhaustive discussion.

Multicore cache management Several techniques have been proposed to improve cache management on multicore processors. O^2 [6] is a software run-time that manages cache contents using thread migration. The O^2 run-time attempts to track cache contents, assigns data to a cache when there is spare capacity, and migrates threads amongst cores as they access data items. Software data spreading [10] aims to allow single threaded applications to use the cache capacity of all

cores' caches using techniques similar to O^2. Several research operating systems, such as Corey [5], Barrelfish [3], and fos [13], try to improve the cache usage in the operating system kernel by dedicating cores to operating on particular sets of kernel data.

Computation migration The J-Machine was 1024-node parallel computer built from message-driven processors [8], which provided low-overhead messaging and context switching, similar to `cmsg`. Applications developers wrote fine grained concurrent programs for the J-Machine using J-Machine specific programming languages and tools that distributed data objects amongst the nodes and took advantage of the cheap messages to access objects efficiently. MCRL [9] and Olden [7] are software systems that migrate computation to the chip that stores the data in its local memory in order to avoid the latency of off-chip memory accesses. In MCRL the decision to migrate is made dynamically by a run-time, while in Olden the decision is make statically by the compiler.

The project differs from previous work by augmenting an existing microarchitecture with DMC instructions. The goal is for existing applications and run-times to improve performance with only a few small modifications.

3 DMC hardware interface

This section describes the high-level design of the DMC PowerPC. We present the hardware interface for each DMC instruction, describe the semantics guaranteed by hardware, and give examples of how software might use each instruction. We assume a cache architecture with per-core L1 data caches and an inclusive L2 shared by all cores. We think, however, that DMC instructions could be implemented for other architectures as well.

3.1 `cpush`

The `cpush` instruction takes two arguments: an address and a core ID. When a software thread executes `cpush address core-id` it is requesting that the hardware copy the contents of the cache line at `address` to the core identified by `core-id`. If, for some reason, hardware ignores the request, software correctness is not affected (similar to ignoring a prefetch instruction).

The outcome of executing `cpush address core-id` depends on the cache line state (modified, shared, or invalid) of `address` in the local L1 cache. The following list describes each outcome.

- If `address` is marked shared in the local L1 then the cache controller copies the cache line to the destination cache and marks it as shared.
- If `address` is marked modified in the local L1 then the cache controller invalidates the local cache line, copies the cache line to the destination cache, and marks the cache line as modified in the remote cache.
- If `address` is invalid in the local L1 then the cache controller ignores the request.

The processor pipeline doesn't wait for the cache controller to copy data between caches.

Software can use `cpush` to optimize inter-core communication of shared memory applications. If a thread running on one core knows it will need to to share recently accessed data with another core it can use `cpush` to move the data to the other core's cache. The hope is that the data will arrive in the core's cache before the core tries to access it.

One example usage of `cpush` it to optimize thread migration in multicore run-times, like MIT Cilk [1] or the Go programming language [2]. Multicore run-times migrate a thread by de-scheduling the thread off the source core, saving the values of the CPU registers in a thread context buffer, and adding the thread context buffer to the run-queue on the destination core, which will execute the thread.

The cost of migration is composed of cache miss penalties to transfer the thread context from one cache to another, and the cache miss penalties once a thread starts executing and accessing its working set. A multicore run-time could reduce both of these components using `cpush` to push the thread context and parts of the thread working set (*e.g.* the top stack frames) from the source to the destination core before the source core adds the thread context to the destination core's run-queue. The cache controller will be transferring the thread context and working set to the destination core while the source core is adding the thread context to the run-queue. The destination core can read the thread context buffer without incurring cache misses and once the thread begins executing it will be able to access parts of it's working set without incurring cache misses.

3.2 `clookup`

The `clookup` instruction takes an address as an argument and returns the closest core that caches that address.

The return value of executing `clookup address` depends on the cache line state in the local L1 cache and the L2 directory. The following list describes the return value based on the cache states.

- If `address` is marked shared or modified in the source core's L1 then hardware returns the source core's core ID.
- If `address` is invalid in the local L1 and the L2 directory indicates the cache line is shared or modified in another core's L1 then the hardware returns the remote core's core ID.
- If the cache line is invalid in the sending core's L1 and invalid in the directory then hardware returns -1 to indicate that no core caches `address`.

`clookup` was originally designed to help test the implementation of `cmsg`. We think, however, that `clookup` might be useful in its own right. One challenge to building software run-times that manage cache contents is tracking which cores cache what data. Tracking data location in software is error prone, costly, and essentially duplicates the cache line state maintained by hardware. These systems could potentially replace their software data tracking schemes with `clookup`, which would be accurate and have lower overhead.

3.3 `cmsg`

The `cmsg` instruction is an implementation of active messages. Active messages [12] are an asynchronous communication mechanism. An active message contains a destination core ID and a function pointer which the destination core executes upon arrival of the message, passing the message body as arguments to the function. Instead of requiring software to provide a core ID, `cmsg` allows software to specify a memory address, which the cache controller resolves to a core ID. Specifically, `cmsg address, pc, body` causes the nearest core that caches `address` to start executing the function at `pc`, loads the contents of `body` into Special Purpose Registers (SPRs), and loads the source core's ID in a SPR. We refer to active messages sent in this manner as *content addressable* active messages. The DMC implementation also allows an application to specify a destination core directly using the core's ID, which is often useful for replying to a content addressed active message.

Hardware handles `cmsg address, pc, argument` in several ways depending on the cache line state in the source core's L1 cache and in the L2's directory:

- If the cache line is marked shared or modified in the source core's L1 then hardware clears a "delivery" bit in the local condition register (CR) to indicate the message was not delivered.
- If the cache line is invalid in the local L1 and the L2 directory indicates the cache line is marked shared or modified in another core's L1 then the hardware interrupts the other core as described below and sets the delivery bit in the source core's CR to indicate that the message was delivered.
- If the cache line is invalid in the sending core's L1 and invalid in the L2 directory then hardware clears the delivery bit in the source core's CR.

Hardware always delivers an active message when the application passes the destination core ID to `cmsg`.

If the L2 directory holds a suitable destination core, the source core sends a message containing `pc` and `body` to the destination core. When the message arrives at the destination core, the destination core loads `body` into SPRs and generates an interrupt, setting the PC to `pc`. Similar to standard PowerPC interrupts, the destination core saves a small amount of execution state in Save/Restore Registers so that software can resume the execution before the interrupt.

`cmsg` provides a low-overhead mechanism for executing code on a remote core. If a thread running on one core needs to manipulate several cache lines in another core's cache, it can use `cmsg` to do so, instead of copying the cache lines into its local cache. A type of application where this might useful is one that creates many threads which operate on shared data structures. For example, the Linux kernel uses linked lists and other shared data structures to implement the physical page allocator, LRU page replacement, reverse page tables, and many other facilities. Adding to and removing from these linked lists often incurs several cache misses as the kernel updates linked list pointers and modifies subsystem specific shared meta-data.

Linux could reduce the number of cache misses by using `cmsg` to execute the list manipulation code on the core likely to cache the list. The address supplied

to `cmsg` could be the address of the spin lock (or some other synchronization primitive) that the kernel uses to serialize updates to the list. Since the spin lock would always be acquired before updating the list, it's likely that if a core caches the address for the spin lock it will also cache the list meta-data. Using `cmsg`, updates to the list and meta-data might avoid incurring cache misses.

4 DMC hardware design

This section discusses the microarchitecture design of `cpush`, `clookup`, and `cmsg`, and highlights important decisions for ensuring correctness and high performance.

4.1 Design overview

We were able to augment the original PowerPC pipeline with the DMC instructions without making substantial revisions to the original design. The main reason for this is that executing `cpush`, `clookup`, and `cmsg` requires performing many of the same operations (*e.g.* calculating the effective address) and state updates (*e.g.* queuing a request to the L1 cache) required to execute a store or load instruction.

The bulk of our redesign was centered on the L1 and L2 modules. The L1 and L2 modules in the original and DMC PowerPC design implement an MSI protocol and use request and response messages to communicate cache line state between the cores, L1s, and the L2. The following lists the request types:

- Core-to-L1 – Lookup, Push, M, S
- L1-to-L2 – Lookup, M, S
- L2-to-L1 – S, I

and lists the response types:

- L1-to-Core – Lookup, Push, M, S
- L2-to-L1 – Lookup, M, S
- L1-to-L2 – Push, S, I

To avoid deadlocks the original PowerPC cache coherence design enforces the invariant that requests do not block responses and that the L1 handles L2-to-L1 requests before handling pending core-to-L1 requests. In the original PowerPC cache coherence design each each L2-to-L1 response had a matching L1-to-L2 request.

While executing a `cpush` instruction, the core sends a Core-to-L1 Push request to the L1 cache. If the L1 caches the address, it sends a L1-to-L2 Push *response*, which includes the cache line contents, to the L2 and also sends a L1-to-Core response back to the core. We discuss the justification for using a response type and the implications of that decision in Section 4.2. If the L1 doesn't cache the address, it drops the request and sends a L1-to-Core response to the ALU pipeline stage. The L2 module sends an L2-to-L1 M or S message to the destination core. This entire process is illustrated by Figure 1.

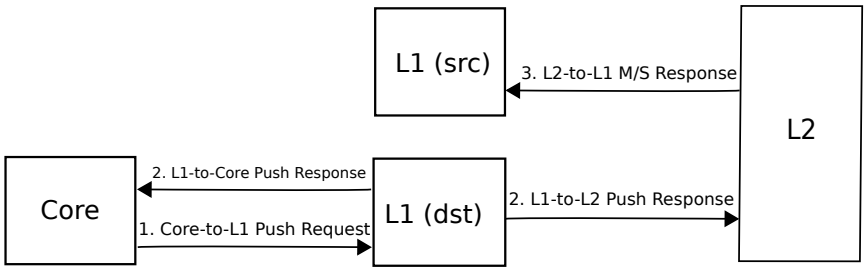

Figure 1: The sequence of messages for processing a `cpush` instruction. The numbers indicate message ordering.

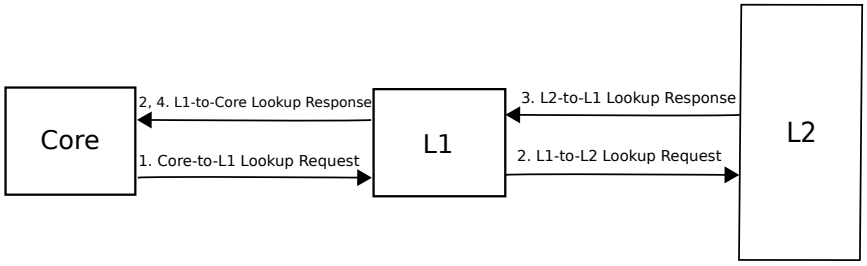

Figure 2: The sequence of messages for processing a `clookup` or a `cmsg` instruction. The numbers indicate message ordering.

To execute a `cmsg` instruction, the core sends a Core-to-L1 Lookup request, which contains the address to lookup, to the L1 cache. If the L1 caches the address, it response immediately with a L1-to-Core response that contains the local core ID. Otherwise the L1 sends a L1-to-L2 Lookup request to the L2. The L2 queries the directory, and responds to the L1 with the core ID. The L1 forwards the response to the core. Figure 2 illustrates this process.

Once the core receives a Lookup response, it either writes the core ID to a register if executing `clookup`, otherwise it sends an active message addressed by the core ID. The active message network is separate from the network used by the cache coherence messages, but is shared with other PowerPC inter-core messages, such as doorbell interrupt messages.

The rest of this section describes some of the challenges we faced when designing DMC and how we addressed them.

4.2 `cpush` correctness

`cpush` requires a careful design because it directly modifies cache state and violates invariants in the original PowerPC cache coherence design. The risk is that `cpush` could cause hardware deadlocks or put the caches in an invalid state. The following paragraphs describe how `cpush` augments the existing cache coherence protocol and how our design handles tricky corner cases that might otherwise cause deadlocks or invalid cache state.

To work well with the existing MSI implementation, the DMC PowerPC cache controller sends the cache line associated with a `cpush` in a response message. This design, however, breaks the invariant assumed in the original PowerPC implementation that L2-to-L1 response has an associated L1-to-L2 request. A potential bug

might be that a core's L1 cache receives a response due to a push request from another another core, but never processes the response. In this example the L1's cache state would differ from the directory maintained by the L2. The following list enumerates the DMC PowerPC invariants for handling request and response messages. **req** refers to a request and **resp** refers to a response.

- If L1-to-L2 **req** = M, L2 accepts if cacheline = S or I, else discard
- If L1-to-L2 **req** = S, L2 accepts if cacheline = I, else discard
- If L2-to-L1 **req** = S, L1 accepts if cacheline = M, else discard
- If L2-to-L1 **req** = I, L1 accepts if cacheline = M or S, else discard
- If L2-to-L1 **resp** = M, L1 accepts if cacheline = S or I, else discard
- If L2-to-L1 **resp** = S, L1 accepts if cacheline = I, else discard
- If L1-to-L2 **resp** = Push, L2 accepts if cacheline = M or S, else discard
- If L1-to-L2 **resp** = S, L2 accepts if cacheline = M, else discard
- If L1-to-L2 **resp** = I, L2 accepts if cacheline = M or S, else discard

A tricky class of the problems that arises with `cpush` is handling the cases where a core request a cache line that another core is simultaneously pushing. For example, if a core is waiting for a response to a modified request for a cache line and a response arrives due to a push request from another core that contains a shared copy of the cache line. If the L1 accepts the shared copy, but marks it as modified, the L1 state and L2 directory state would differ.

The following list describes the logic the DMC PowerPC implementation uses to handle `cpush` in the L2. **resp** refers to response sent from the L2 to the destination core's L1 (*i.e.* the message containing the cacheline) and **req** refers to a simultaneous request for the same cacheline from the destination core or from another core. Each main bullet describes the state of the L2 directory. Sub-bullets under each main bullet describe the logic for handling the the **req** and **resp**. We assume a system with three cores.

- Directory state: source L1 = M, destination L1 = I, other L1 = I
  - If L1-to-L2 **req** is not present, set source L1 = I and destination L1 = M, send L2-to-L1 **resp** = M
  - If L1-to-L2 **req** = M or S (from destination L1), set source L1 = I and destination L1 = M, send L2-to-L1 **resp** = M
  - If L1-to-L2 **req** = M or S (from other L1), set source L1 = I and destination L1 = M, send L2-to-L1 **resp** = M, then process L1-to-L2 **req** from other L1
- Directory state: source L1 = S, destination L1 = I, other L1 = I
  - If L1-to-L2 **req** is not present, set source L1 = S and destination L1 = S, send L2-to-L1 **resp** = S

  - If L1-to-L2 **req** = M or S (from destination L1), set source L1 = S and destination L1 = S, send L2-to-L1 **resp** = S, then process L1-to-L2 **req** from destination L1
  - If L1-to-L2 **req** = M or S (from other L1), set source L1 = S and destination L1 = S, send L2-to-L1 **resp** = S, then process L1-to-L2 **req** from other L1

- Directory state: source L1 = S, destination L1 = S, other L1 = I
  - Discard L1-to-L2 **resp** = Push (from source L1)

- Directory state: source L1 = S, destination L1 = I, other L1 = S
  - If L1-to-L2 **req** is not present, set source L1 = S and destination L1 = S, send L2-to-L1 **resp** = S
  - If L1-to-L2 **req** = M or S (from destination L1), set source L1 = S and destination L1 = S, send L2-to-L1 **resp** = S, then process L1-to-L2 **req** from destination L1
  - If L1-to-L2 **req** = M (from other L1), set source L1 = S and destination L1 = S, send L2-to-L1 **resp** = S, then process L1-to-L2 **req** from other L1

- Directory state: source L1 = S, destination L1 = S, other L1 = S
  - Discard L1-to-L2 **resp** = Push (from source L1)

We believe, based on the above enumeration and the tests described in Section 5, that our design and implementation of `cpush` is correct.

4.3 `cmsg` performance

One potential performance problem with interrupting execution to execute an active message it the cost of saving and restoring General Purpose Registers (GPRs) and setting up a PowerPC ABI compliant environment for executing C code. This process requires about 70 instructions.

To avoid saving and restoring execution state, we added a second register file. When a core receives a `cmsg`, it switches to the secondary register file, and switches back when the `cmsg` interrupt handler returns. It should be possible to ensure the secondary register file is always in an ABI compatible state when the core switches to it. This solution precludes supporting nested `cmsg` interrupts and requires an additional register file. On the BEE3, however, the extra register file fits into a BRAM partially used by the original register file.

Another shortcoming is that the source core sends the `cmsg` message after receiving a reply from the L2 directory. In a more efficient implementation the L2 would send the `cmsg` message directly after performing the lookup, instead of replying to the source core.

5 Implementation

This section describes the the hardware implementation of the DMC PowerPC, the software implementation of the DMC run-time, which we used for benchmarking and testing, and the testing suite we wrote and used to test the DMC PowerPC.

Hardware The DMC PowerPC implementation adds about 250 lines and modifies about 750 lines of code in the original 7701 line PowerPC BSV. Most of the modifications were to the `mkL2Cache` and `mkDCache` modules. The DMC PowerPC is compatible with the same boards as the unmodified PowerPC.

We feel it was import to add DMC instructions to a cycle accurate hardware simulator that is itself implemented in hardware. Without a cycle accurate simulator, it would be difficult to perform a performance evaluation of benchmarks that use DMC instructions.

Working with a simulator written in BSV and that runs on an FPGA has two advantages over software simulators. One is that adding DMC instructions actually requires modifying hardware, in contrast to software simulators. This allows us to gage the complexity of adding the instructions to a real processor implementation and forces us to respect hardware constraints, such as limited on-chip storage and short critical paths. This is in contrast to software simulators which developers often extend using C or high-level languages like Python. Without the constraints of hardware, developers can implement overly simplistic or unrealistic designs. For example, the author implemented cache coherence in an $x86$ emulator in about 40 lines of C and essentially added a 1 byte overhead for every cache line of memory. The implementation worked well for a simulator written in C, but its simplicity and storage overheads would not work well on a real processor.

A second advantage from running on an FPGA is that simulation is fast. The PowerPC model runs at 100 MHz and uses approximately 9 cycles to model 1 real PowerPC cycle. Therefore, the simulated PowerPC runs at about 11.11 MHz. This is two orders of magnitude slower than a real server PowerPC chip might run, but also two orders of magnitude faster than a cycle accurate full system simulator written in C [4].

Software The DMC run-time, which includes threads, a thread stealing scheduler, locks, a linked list implementation, and a memory allocator, is about 2000 lines of C code. DMC instructions are easy to use. Modifying code to use DMC instructions usually requires changing only a few lines of C code.

Testing We tested the DMC hardware using a series of software stress tests. Much of our testing focused on `cpush`. One reason for this is that `cpush` is tricky to implement correctly because it modifies cache state, therefore we wanted to test it thoroughly. A second reason why most of our testing focused on `cpush` is that we implemented `cpush` first and the implementation of `clookup` and `cmsg` reused much of the well tested `cpush` code.

Our tests for `cpush` try to trigger the corner cases described in Section 4.2. For example, to trigger the case where a core executes a `cpush` on a modified cache line while another core simultaneously requests a shared copy of the cacheline, the test would create threads on different cores, one thread would spin in a loop

incrementing a shared variable then calling `cpush`, while the other thread would spin and constantly read the value of the variable.

One challenge in testing `cpush` was to verify that executing `cpush` would cause a cache line to be copied into another core's cache. To verify that `cpush` was behaving as expected, we instrumented the `mkL2Cache` and `mkDCache` modules to print Push requests and responses and inspected the output.

6 Evaluation

We evaluated the DMC PowerPC implementation using a dual-core processor synthesized to a BEE3 FPGA system. The PowerPC model simulates a 100 MHz processor. Each per-core L1 data cache is 64 Kbytes and the shared L2 cache is 512 Kbytes. Cache lines are 16 bytes and the memory access latencies are fixed at 31 cycles to access the L2 and 255 cycles to access DRAM.

We use three microbenchmarks, programmed to run with and without making use of `cpush` and `cmsg`, to measure performance. We choose one microbenchmark to measure how expensive it is to execute `cmsg`, and two others on the basis that the represent operations executed in complex applications. The performance measurements should, at best, be considered encouraging preliminary results. The DMC PowerPC lacks features found in advanced processors, such as out-of-order execution, hardware prefetching, and symmetric multi-threading, which would change performance.

6.1 The cost of `cmsg`

To understand the cost of executing a `cmsg` instruction we wrote a microbenchmark that compares the cost of reading cache lines from another core's cache to the cost of executing a `cmsg` to read the cache lines. The benchmark creates two threads. One thread fills its cache with shared cache lines by modifying every cache line in a 64 Kbyte array. A second then reads the entire array in N cache line segments. The benchmark measures the average time to read one N cache line segment using `lw` and using `cmsg` to an active message to read the cache lines. After executing a `cmsg`, the thread spins in a loop until a flag variable is set to zero. The destination core sends replies using an active message, which clears the flag variable. The point at which using `cmsg` becomes cheaper than `lw` helps show when it might improve performance to use `cmsg`.

Figure 6.1 shows cost using `lw` and `cmsg`. The x-axis shows the number of cache lines in each segment and the y-axis shows the average latency to read each segment. The `cmsg` case always generates a pair of lookup request and response messages, one inter process active message to read the cache lines, and one inter process active message to signal that the read is complete. The `lw` case generates a pair of request and response cache coherence messages for each cache line. Therefore, we expect the latency of the `lw` case to increase much faster than the `cmsg` case.

With 1 cache line, using `lw` and `cmsg` generate the same number of cache coherence request and replies, and perform about the same. For two cache lines, the `cmsg` reduces the latency to read the cache lines by about 17%. As the benchmark manipulates more cache lines, `cmsg` provides more benefit. Using `cmsg` to access 8

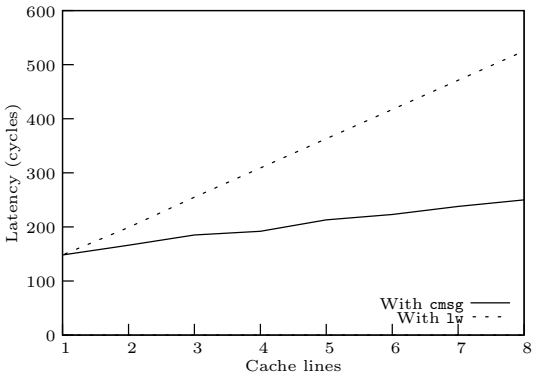

Figure 3: Results for the memory scan benchmark. The x-axis shows the number of cache lines in a segment and the y-axis shows the average latency to the read segment from another core's L1 cache.

cache lines is 52% faster than using `lw`. The cost of `cmsg` increases slightly as the set size increases because the benchmark must execute more instructions to read all cache lines.

6.2 Thread migration

To evaluate potential software performance improvement from using `cpush` we wrote a microbenchmark that ping-pongs a thread between two cores and measures the average round-trip time. The benchmark uses two cores, one is executing the migrating thread while the other spins in its scheduling idle loop, continuously checking for threads on its run queue. To migrate the thread, the source core de-schedules the thread, switches to another thread (the idle thread in this benchmark), which saves the core's registers in a context buffer and adds the thread context buffer to the run-queue of the remote core. The idle core will notice that the new thread context on its run queue, dequeue the context, and start executing the thread by reloading the thread register values from the context buffer.

When the remote core loads the values of a thread's registers it usually incurs several cache misses, which increases the round-trip time. We use `cpush` to reduce the round-trip time by pushing the contents of the context buffer to the destination core before writing to the shared variable. This means that transferring the context buffer from one core to another will be overlapped with the operation of adding the context to the remote run queue.

Figure 6.2 presents the results of the thread ping-pong microbenchmark. The x-axis shows the number of cache lines in the thread context that the benchmark uses `cpush` to move from the source to the destination core. The y-axis measures the round-trip time in cycles to migrate a thread from the source to destination and back to the source.

Without using `cpush`, the round-trip time is about 2202 cycles. As the benchmark uses `cpush` to move more cache lines the round-trip time reduces steadily until 6 cache lines, where the round-trip time is 831 cycles. Pushing more than 6 cache lines does not decrease the round-trip time because the FIFOs connecting the pushing core's L1 cache, the shared L2, and the destination cores L1 cache become full. In the current implementation the destination core stalls in this case.

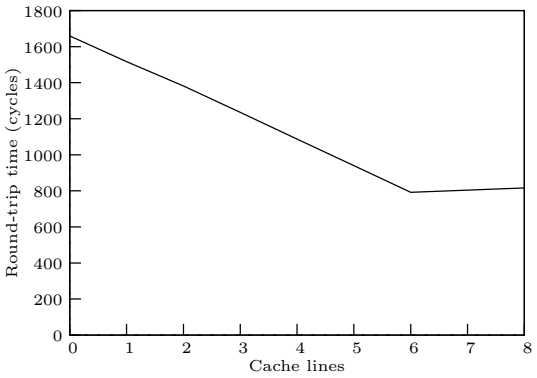

Figure 4: Results for the thread migration microbenchmark. The x-axis shows the number of cache lines the source core pushes to destination core using `cpush`.

It would be correct, however, to simply drop the push request in the source L1 if the FIFO to the destination L2 is full.

6.3 Linked lists

Linked lists are commonly used to build more complex data structures. For example, the Linux kernel uses linked lists to implement the physical page allocator, LRU page replacement, reverse page tables, and many other facilities. The kernel usually maintains invariants, implemented with shared memory, associated with complex data structure. When the kernel updates the underlying linked list, it also updates the other invariants. For example, when adding a virtual page to a reverse page table, the kernel acquires a lock, inserts the page into a list, and increments a per-page reference count.

We wrote a list microbenchmark to measure potential performance improvements from using `cmsg`. The list microbenchmark initializes a list by inserting 20 elements into the list, then creates two threads that insert into or remove from the list. To operate on the list, a thread acquires a spin lock protecting the list, performs the insertion or removal with equal probability, and releases the lock. Performing an operation on can incur as many as 5 cache misses: acquiring the lock, setting a list entries next and previous pointers, setting the previous elements next pointer, setting the next elements previous pointer, and releasing the lock. To model the situations where software might update additional shared memory (*e.g.* the reverse page table described above), the benchmark modifies a variable number of extra cache lines while holding the spin lock.

The list microbenchmark uses `cmsg` to perform the list operation. The microbenchmark uses the the address of the spin lock to address the message, uses the address of the function performing the list operation as the PC, and uses the list element to insert or delete as the argument. After executing `cmsg` the thread spins until a flag variable to set to zero. The destination core replies using an active message, which clears the flag variable.

Figure 5 presents the results for the linked list microbenchmark. The x-axis shows the number of extra cache lines the benchmark modifies while holding the spin lock. The y-axis shows the average latency for executing a list operation.

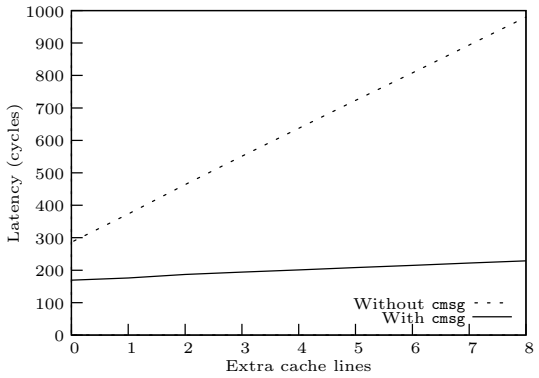

Figure 5: Results for the list microbenchmark.

The results indicate that, even for modifying on extra cache line, using `cmsg` decreases latency by about 22%. As the number of extra cache lines increases, the cost of performing a list operation increases with and without `cmsg`. Both increase because of the additional instructions the CPU must execute to modify the extra cache lines. However, the cost without `cmsg` increases much faster than the cost with `cmsg`, because every additional cache line modification incurs a cache miss. When using `cmsg`, on the other hand, the extra cache line are likely to held in the destination core's L1 cache.

7 Future work

One factor limiting the extent of the workloads we can evaluate is the low core and cache count of our dual core system. This system makes it hard to evaluate how well DMC works for workloads that could take advantage of a large aggregate on-chip cache capacity by actively managing cache contents. Adding support for more cores would allow us to explore using DMC instructions in for some of these workloads.

The implementation and evaluation of `cmsg` and active messaging has a number of loose ends. Our current usages of `cmsg` assume that the destination core is always running in the same virtual address space as the thread executing `cmsg`. This assumption works for executing kernel functions in a kernel with a global virtual address space, like Linux, but doesn't allow user-level threads to execute `cmsg`, because the destination core might be running in a different virtual address space. One potential solution is for the processor to include value of the Process ID Register (PID), which serves as the core's TLB tag, in the active message. Upon reception, the destination core load the PID value.

Our evaluation of `cmsg` doesn't address how the operating system kernel can guarantee fairness. A core could spend all its time executing active messages from other cores, starving the threads on that core's run-queue. It might be possible to detect this situation and either migrate all the threads to other cores, or mask active message interrupts for a short while.

A second issue we should evaluate is the patterns of memory accesses applications make for which `cmsg` might hurt performance. For example, if an application reads some set of data objects very often, it might not be beneficial to use `cmsg`

to accesses them, and instead allow the MSI cache coherence protocol copy the objects into all the L1 caches.

8 Conclusion

This report introduces the DMC instructions for managing on-chip caches and describes the design and implementation of the instructions for a PowerPC processor. Results from mircobenchmarks indicate that using DMC instructions improves the performance of certain operations by reducing the number of cache misses. The results suggest that DMC instructions might be useful for a larger class of workloads.

References

[1] The Cilk Project, May 2011. `http://supertech.csail.mit.edu/cilk/`.

[2] The Go Programming Language, May 2011. `http://golang.org/`.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Haris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: a new OS architecture for scalable multicore systems. In *Proc of the 22nd SOSP*, Big Sky, MT, USA, Oct 2009.

[4] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. Van Hensbergen, and L. Zhang. Mambo: a full system simulator for the powerpc architecture. *SIGMETRICS Perform. Eval. Rev.*, 31:8–12, March 2004.

[5] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. D. Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proc. of the 8th OSDI*, December 2008.

[6] S. Boyd-Wickizer, R. Morris, and M. F. Kaashoek. Reinventing scheduling for multicore systems. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS-XII)*, May 2009.

[7] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[8] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills. Architecture of a message-driven processor. In *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98, pages 337–344, New York, NY, USA, 1998. ACM.

[9] W. C. Hsieh, M. F. Kaashoek, and W. E. Weihl. Dynamic computation migration in DSM systems. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, Washington, DC, USA, 1996.

[10] 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*, pages 460–470, New York, NY, USA, 2010.

[11] R. Strong, J. Mudigonda, J. Mogul, N. Binkert, and D. Tullsen. Fast switching of threads between cores. *ACM SIGOPS Operating Systems Review*, 32(2), April 2009.

[12] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th annual international symposium on Computer architecture*, pages 256–266, New York, NY, USA, 1992.

[13] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, 2009.