Range-based ~~CMO's~~ CBOs considered harmful.

Derek Williams (IBM)

Sept 20th, 2020

v1.0

First a disclaimer

- This represents personal opinions only and should not be construed as any sort of official position endorsed by IBM.
- Further, no information in this presentation is considered IBM Confidential.

Acknowledgments.

- I am grateful to several people listed below (in no particular order and missing some that wished to remain anonymous) who commented on several early drafts in ways that corrected errors and helped clarify the exposition. Being listed below does NOT imply the person agrees with anything said in this presentation (some don't). It only implies that a conversation I had with them affected the ultimate contents of the presentation. As usual, any errors belong to me alone.
  - Palmer Dabbelt
  - Bruce Hoult
  - Bill Huffman
  - Grigorios Magklis
  - David Kruckemyer
  - Andy Glew
  - Andrew Waterman
  - John Ingalls

Part I : Introduction.

Introduction

- The current (as of Sept 3rd, 2020) RISC-V proposal for Cache Maintenance Op (CMO) instructions includes the notion of what I'll call a "range based" CMO -- i.e. a CMO that takes two parameters: a start and an end address.
- I think this is a bad idea and should not be done for the baseline cache ops. At the end we'll discuss a set of instructions/operations that are range based, but these are NOT the base cache block instructions.
- Making CMOs range-based is a choice with significant consequences, and so far they seem to be this unquestioned fait accompli (one person I know called it "RISC-V's 11th commandment: Thou shalt do CMOs as ranges"). I believe we need a robust discussion on this subject before that choice is made. If RISC-V is going to do range-based CMOs, we should really know why we're doing it, what we get for it, and what that costs.

Frist things first….

- We’re going to quit calling these CMOs (Cache Management Operations).
- Instead we’ll call them CBOs (Cache Block Operations).
- It will be important to think of these things as what they actually are: Cache Block Oprerations (with heavy emphasis on the word “**Block**”).
- I’ll use the words “Block” and “Line” interchangeably in this presentation.

Scope of this opinion piece -- issues covered.

- CBOs are, sadly, a HUGE subject with many, many potential points of controversy.

- The main issue covered in this presentation is the “shape” of the CBO: how many parameters a CBO has and what those parameters mean (i.e. is it a range-based CBO -- 2 address parameters, or a cache line-based CBO -- 1 address parameter).

- So this is mainly section 2 (Fixed Block Size PREFETCHes) and 3 (Variable Address Range CMOs) in the current proposal (2020-08-12 22:03:41 timestamp).

Scope of this opinion piece -- issues not covered.

- At least in this initial presentation, I will scrupulously avoid the discussion of so called "Microarchitecture Structure Range CMOs" (aka CBOs that address microarchitecture structures directly, for example, by set/way -- section 4 of the current RISC-V proposal). Getting entangled in that discussion is a mistake, to quote the famous movie*, right up there with "getting involved in a land war in Asia" ☺. Thanks, I'll pass (at least for now).

- This presentation also avoids the lesser controversy of how and if CBOs specify the cache levels or Points of Unification/Coherence they occur at. Those issues are mostly orthogonal to the issues we need to discuss here. For this discussion, we assume the CBO is global in effect.

- Finally, there is nothing in this presentation about attempting to reduce code size. That's a different subject for a different time and mostly orthogonal to CBO shape.

*Princess Bride, 1987. Google Search: "land war in Asia quote".

Mixed-line sizes - the dreaded issue #8.

- Bruce Hoult has raised the concern that the CBO design should not assume that the cache line sizes are fixed during the program execution (issue #8 in GitHub).

- For example, a thread of execution gets moved from one HART with cache line size X to a HART with a differing cache line size Y.

- It's a legitimate concern to be sure, but it would needlessly complicate the presentation below to address it immediately. There are also strict limits to what one can do about transparently handling time-variant mixed line sizes.

- I will address this later in the presentation in quite a bit of detail, but for now, please briefly and tentatively assume the line size is NOT time variant in the system.

Types of cache block instructions.

- It will help to define a tentative set of CBOs when discussing these issues (these are just examples; not necessarily what RISC-V will have). I also give Power ISA mnemonics for these for those that know them:

Name	Description.
FLUSH (dcbf)	Any modified copy is written to memory and then all cached copies are invalidated.
CLEAN (dcbst)	Any modified copy is written to memory and downgraded to an unmodified state. All unmodified cache lines are unaltered.
ZERO (dcbz)	All other cached copies of the line are invalidated, and a new copy of the cache line containing all zeros is established (without reading data) in the cache of the processor executing the instruction.
INVALIDATE (dcbi)	All cached copies (including copies modified relative to main memory) are invalidated. No data is written back to memory.
TOUCH_RD (dcbt)	A copy of the line is established in the processor executing the command as if the location were read by a Load instruction.
TOUCH_WR (dcbtst)	A copy of the line is established in the processor executing the command as if the location were read by a Store instruction.
ALLOCATE (dcba)	All other cached copies of the line are invalidated, and a copy of the cache line is established (without reading data) in the processor executing the instruction. The line contains whatever values were present in the cache before the DCBA (a known security hole).

A “touch” vs a “prefetch”.

- The touch instructions listed above are called “touch” because they are intended to “touch” (request) only one cache line.

- There is a whole class of “prefetch” CBOs one can consider implementing that control and manipulate prefetch engines (Power calls these the “stream variants” of the touch instructions).

- Depending on the nature of these prefetch engines, there could be a large amount of parameters needed: stride length, end address, stream ID, and on and on. So, a “prefetch” instruction as I’m defining it here will likely need additional plumbing/parameters beyond the baseline one cache line touch variants. We leave this debate for a later day and concentrate solely on the single cache line touch instructions.

DCBZ has two personalities.

- DCBZ can be thought of in two ways: 1) A cache line sized operation that zeros a cache line or 2) A wide fast cache line-sized store of zeros to clear a region of memory that you might think of extending to handle possibly zeroing arbitrary ranges of memory.

- The cache line-sized DCBZ (option 1) is useful to clear RAS errors from cachelines within caches and several other uses. There's no easy alternative way to pull this off. This option will have the same "shape" as all the other CBOs, and for these we'll be suggesting a non-range-based shape.

- The possibly extensible notion of DCBZ that can zero unaligned regions down to the byte (option 2) is optional (you could always use stores for that), but desirable. In this presentation we first concentrate on the first option which is called out as ZERO (dcbz). It's always a cache-line sized operation. The second option will be discussed later in the presentation but will be called ZERO_MEM. This option will have a range-based shape.

A diversion: kiss your memory model goodbye.

- If RISC-V provides INVALIDATE (dcbi) and/or ALLOCATE (dcba), we can kiss our nice, precisely defined memory model goodbye where these instructions get used.

- INVALIDATE destroys any writes in the caches that haven't made it to main memory and reverts to some earlier write for the location. ALLOCATE is worse… you get the values in whatever line the DCBA randomly chooses out of the cache. This turns the memory model into a wet, steaming pile of implementation specific goo you can't easily reason with.

- These instructions are dangerous (but may be required for some implementations) and are painful work for the formal model, verification, and compliance folks. Power (server) dropped these as quickly as we could (general purpose machines didn't need them).

- The arch should make it legal for implementations to implement DCBI as a DCBF and DCBA as a DCBZ. Even if it doesn't, implementations should have chicken switches for this.

How much opcode space is this going to cost?

- This presentation is utterly unconcerned about how many opcodes it might cost to implement any of the suggestions contained herein.
- I'm not an expert on the allocation of opcodes in RISC-V and I intend to get through this lifetime without being burdened with that knowledge ☺.
- Here, we will discuss concepts and suggested ways of managing CBOs and range-based operations. If at the end of the day, one or more of the suggestions isn't possible due to lack of opcode space, you can deal with that problem then.

This should be non-controversial, but…..

- This presentation assumes the following as (hopefully) uncontroversial facts:
  - Cache line sizes must be a power of 2.
  - The minimum line size should be the size of the largest single-copy-atomic access in the ISA (you don't want these spanning two cache lines) with some room for growth. Someday, double-wide CAS might happen, so the minimum size should be at least 16.
  - The maximum theoretical cache line size is 4096 bytes (the smallest page size). Cache lines should not span pages.
  - To explicitly state the obvious: that gives us line sizes of 16, 32, 64,128, 256, 512, 1024, 2048, 4096.
- If there is some need to contest these facts, please explain why.

This is just one person’s opinion.

- This presentation is just a specific person’s position and point of view, though the position is one of a partisan who does not like range-based CBOs at all.
- The presentation is intentionally provocative in places to provoke conversation and debate.
- I am happy to be shown concrete examples pointing out where this presentation is wrong or misguided. I’m sure it will be in places.

Part II : Myths about range-based CBO instructions.

Nobody’s done this.

- A first, almost philosophical, objection to range-based CBOs is that no one has done range-based CBOs before. In at least 30 years or so of machines with cache control instructions, across more than one architecture, it hasn’t happened yet (that I can find -- show me if I’m wrong). That should give a level of pause.
- That does NOT guarantee that RISC-V is wrong to be the first (if it chooses to).
- However, there should be an appropriate level of fear and humility before doing this. In other words, there should a clearly articulated, good set of reasons to do this.
- What is guaranteed is that RISC-V will have the unalloyed joy of finding all the problems and limitations with range-based CBOs -- no one has done it for you. Choose wisely……

The various myths of range-based CBOs.

- In many discussions with various people, I have been given several reasons (that I now call myths) that range-based CBOs needed to be implemented.
- None of these have been at all satisfying. It’s always possible I misunderstood.
- We’ll go through each of them and point out limitations.
- These are just the set of reasons (myths to me) that I’ve heard.
- If you have more reasons for ranges, please, bring them up. There might actually be a good reason to do range-based CBOs…. I just haven’t seen one yet.

Myth #1: Trap and emulate CBOs.

One of the first reasons I heard to do range-based CBOs was to allow the CBO to be trapped and emulated efficiently.

- What? How do you emulate a CBO? Except for DCBZ that you can fake (in some cases) with stores, the only way to do the function of a CBO is to use some sort of CBO instruction.

- If you're going to trap on each CBO, and emulate it using some off-the-books "unarchitected" CBO that you already have, the CBO would have to have a range to allow you to amortize the cost of the trap over multiple CBOs (trapping on every block will kill you on overhead). Really?

Myth #1: Trap and emulate CBOs, cont'd.

- I'm not sure that hiding the trap overhead to allow recycling of native CBOs is the intent here, so I may be entirely wrong*. If I am, I apologize, but if not, there's a simpler way to deal with the trap overhead: Don't Trap. Just implement the new native CBOs.
- Do your penance now for not architecting CBOs at the beginning and get it over with.
- No one can take your native CBOs away from you and you can move your software stack over the new CBOs at your leisure after they are implemented.
- Forcing ranges into the CBOs for this reason is a short-sighted decision that adds needless complexity to the ISA that you have to live with forever. You get to live with mistakes made in RISC-V now for a really long time.

* Other than to remove trap overhead with a range, to directly implement new CBOs as native CBOs one-to-one, (or maybe to write bytes for DCBZ -- which we'll do later), I can't see any reason to EVER trap and emulate a CBO. If you have another reason, please offer it up.

Myth #2: The OS dumping the whole cache.

- Another reason I was told ranges were useful is that operating system code could look at the range and decide that it was large enough to implement the function in some other magical (my term for it) fashion that was faster than doing a series of cacheline sized CBOs. The main example was something about the OS could “dump the whole cache”.

- This one is hard to understand. First, “dumping” a cache can really only apply to CLEAN or FLUSH -- it makes no sense to speak of applying any of the other cache ops to a whole cache. An OS can’t flush or clean all the lines in a cache directly. The OS doesn’t know what lines are in the cache. That takes hardware.

- There’s room to discuss CBOs that FLUSH or CLEAN the entire local cache that a given HART is using, but that’s a microarchitecture structure-based CBO and not a mainline CBO. So, either I misheard this explanation, or it just doesn’t make any sense. I’m happy to have someone come forward with an example that does make sense.

Myth #3: range-based protocol implementations.

- The next myth was having range-based CBOs would enable implementations to use range-based operations down at the bus protocol layer and in the various pipes/queues.
- It is true that range-based CBOs would allow implementations to be dumb enough to try and use range-based operations in the bus protocols. So what?
- The question that matters is whether it's sensible to ever do range-based bus protocol operations? And even if it is sensible, is it sensible in a high enough percentage of the possible implementations to gum up the ISA with range-based CBOs?
- Some outlier implementation can, I'm sure, force range-based protocols to work. I'm not sure encouraging that design team at the cost of everyone else is a good idea. That team always has the option of having their own instructions (and probably already does).

More difficulties with range-based protocols.

- The list of issues with ranged-based bus/pipe operations is long, so this is not exhaustive.
- The interconnect protocol must now ship around two addresses (or start/length, etc.) instead of one address for each operation. This can gum up the address bus protocol in unpleasant ways and is only useful for these CBOs. Everything else has one address.
- Looking up a CBO operation in a cache (either as a snoop/probe or looking at your own cache on the executing procesor) to see if it hits is not just looking at one congruence class and possibly finding one matching line in that cache (which takes a fixed amount of time). It is now a full walk of every congruence class that may hit up to every entry in every congruence class. Or a sequencer must bust the range up into a ton of individual lookups.
- The cache lookup can now take a large and variable amount of time.

Yet more difficulties…..

- Certain coherence protocols (retrying buses) can’t tolerate long variable delays. The snooped op shows up, and ten or so cycles later the response must be ready. Range-based bus ops won’t even work in that style of coherence protocol (yes, I know, most buses in use today aren’t retrying buses… but not all of them -- ours certainly isn’t).

- Intra-thread ordering is more difficult. If you have stores before a flush that overlaps the stores, for example, you want those stores to be pushed out by the flush. The usual intra-thread collision detection machinery expects each instruction to specify one address and not a range. Yes, you can force it to work in a number of ways: inject barriers into the pipeline before and after the instruction, require the programmer to put the barriers in, make a CBO match any memory address when compared to, etc., etc. But it’s messy.

- Some protocols route traffic by address to banked caches. I’m not sure how to make a range-based operation that could span over different busses play nicely with that.

Range-based protocols are a non-starter.

- With the issues listed above, to say nothing of the added complexity that must be verified by your verification team, I don’t consider enabling range-based protocols to be anything even approaching a reason to define range-based CBOs in the architecture.
- If anything, not tempting people to try range-based protocols would strike me as a reason to NOT put range-based CBOs in your ISA.
- I could be completely wrong here and I look forward to someone who has a ranged based bus protocol explaining the errors in my thinking.
- Until then, the rest of this presentation is going to assume that cache operations are always implemented in a “Line-At-A-Time” (LAAT) fashion in the pipes and bus protocols.

Myth #4: Ranges are required for portability.

- The general idea here is that range-based CBOs somehow magically hide the line size from the user and allow software using CBOs to suddenly become portable.
- I vehemently disagree with the implied notion that “hiding” the cache line size from the user is a good idea or that doing so will somehow make the software using CBOs “portable”.
- As we’ll discuss later, there are immovable, fundamental limits on how much the line size can be hidden and how “portable” the code using CBOs can be.
- Instead of range-based CBOs, the ISA can provide a portable way to query the cache line size. This will provide as much information as having the CBO adjust the start address to the next cache line (how the current proposal works) and simplifies the CBO instruction implementations.

The CLCS “instruction”.

- As an example of a query mechanism, the original POWER architecture (not PowerPC) had an instruction called “CLCS” or “Cache Line Compute Size”.

- This instruction had a source register and a target register. The value in the source register specified which of the various line sizes (data or instruction*) was desired and the instruction returned the requested line size in the target register**.

- RISC-V could implement a similar instruction. It could also implement multiple pseudo-opcodes that map to reads of user accessible read-only CSRs with the line size values. The point is that there are options that achieve as much portability/hiding as the current plan has without requiring range-based CBOs.

* This presentation completely ignores I-side cache control instructions, but the principle applies there as well.

** Yes, having a separate instruction from the CBOs return the line size opens the possibility the line size changes dynamically between those two instructions. We’ll fix this, where necessary, later. For now, please continue to quietly assume the line size is NOT changing dynamically.

The CLCS “instruction”, cont’d.

- Yes, a CLCS instruction (or the CSR reads which I’ll refer to as CLCS from now on) is a bit redundant with the more comprehensive “device-tree”-esque discovery mechanisms that are being considered in another workgroup.
- However, cache line size discovery deserves its own specific portable mechanism even if it’s a bit redundant. Given that cache line size discovery is so intimately tied up with using the CBOs, cache line size discovery should not require a Hypervisor call, OS call, or even a “device-tree” like mechanism.
- To be useful, the cache line size discovery mechanism must be no harder to access to than the CBOs themselves. Hence the user level CLCS instruction.

Part III : The currently suggested range-based CBO instruction.

The current suggested CBO.

- The current suggestion for the CBO is an R-format instruction (op RD, RS1, RS2) where RD must be the same register as RS1 or it's an illegal form:

  CBO.<op> RS1, RS1, RS2   (R-format)

  Where:

  - "<op>" is the type of CBO: flush, clean, invalidate, allocate, zero etc.
  - Initially, RS1 is the starting byte address of the region of memory to apply the cache control operation to. When the instruction executes, the instruction updates RS1 to point at the next cache block boundary or the upper boundary of the region.
  - RS2 is one byte beyond the upper byte boundary of the region of memory to apply the cache control function to (i.e. RS2 = initial value of RS1+length of region).

The current suggested CBO: usage in a loop.

- The suggested loop construct to use around the CBO:

```
        x11 := start_boundary;          // set up start address
        x12 := end_boundary;            // set up end address -- (start_addr+length)
loop: CBO.flush x11, x11, x12           // do the CBO and move the start pointer.
        bne x11, x12, loop              // rinse/lather/repeat.
```

- There are several criticisms of this instruction. We’ll take each of them in turn and in no particular order of importance.

1st critique: start/stop is ambiguous.

- Specifying the range by using a starting byte address and an ending byte address is ambiguous and is open to multiple interpretations.

- If start=stop, is that do zero bytes or do all of memory? Or even one byte?

- If start > stop, what does the CBO do?
  - Blindly do one cacheline and move the pointer? If so, the loop would go all the way around the top of memory, thru zero, and up to start.
  - Does the implementation of the CBO have special case logic to check for start > stop and set RD to stop and do no cacheline?
  - Does the CBO check for start > stop and not do the CBO, but still update RD (RD must be updated to something or the loop will never stop). In this case we would leave memory above start to the top of memory alone and start applying the cache operation at address zero.

1st critique: start/stop is ambiguous, cont'd.

- Do you put instructions in front of the loop to test for start >= stop and skip the CBO? That at least limits what happens in those cases, but at the cost of a more complex loop?

- The fact that you must ask questions like this at all to know what the instruction does is not a good starting point.

- While ranges should not be applied to CBOs, if you must specify a range in some instruction (we'll do this later when we define range instructions ZERO_MEM, FLUSH_MEM, CLEAN_MEM, etc.) specifying the range as start address/number of bytes in region is at least unambiguous. This is more of an issue with trying to build sequencing into a single instruction with one loop like the current suggestion does. When we get to the range-based instructions later in this presentation, you can make either choice much more sensibly.

2nd critique: byte addresses don't work in general.

- For several of the cache control instructions, it's impossible to work at a byte level.

- For example, an INVALIDATE (DCBI) with start address 13 and stop address 27 on a machine with 16 byte line sizes isn't going to work. You can't INVALIDATE part of a cache-line. It just doesn't work. The allocate (DCBA) instruction has similar problems.

- Even for something like ZERO (dcbz), you can only make byte addresses work in some cases. If you're using ZERO (dcbz) to clear an ECC error from a cache line, arbitrary byte addresses can't work. In that use case, the ZERO (dcbz) must work on a whole cache block. However, if you're using ZERO (dcbz) just to zero memory bytes, one could build special hardware to do partial line updates and force byte ranges to work, but that's not a cache block instruction anymore (more of this later). And we'll always need to provide the cache-block version of ZERO (dcbz) to clear RAS errors.

3rd critique: RS2 is a terrible thing to waste

- Any range-based CBO takes two parameters to specify a range (e.g. start/end or start/length).
- Unfortunately, that means we've used up ALL our inputs (RS1 and RS2) on specifying the range. At a minimum, we needed to use RS1, but not RS2 -- it's essentially wasted here.
- It might be best to save RS2 for future expansion and not waste it on ranges for CBOs. You could do different addressing modes like base(register) + offset(register), you could use RS2 to hold additional information for microarch addressed CBOs if it doesn't fit in RS1? Maybe parmaters for stream variants of the touch instructions if those are ever done?
- RS2 might be more productively used in the future. I'm not sure we should use it up now.

4th critique: wasting some opcode points here

- This one isn't terribly important, but it does seem a bit wasteful.

- The CBO is an R-format instruction:
  - OP RD, RS1, RS2

- But the way it's used here, RD != RS1 is an illegal form.

- That's some number of opcode points being tossed out the window. If there was a good reason for it, maybe, but the reason RD=RS1 (allowing the implementation of hardware sequencers) isn't a good one (covered later).

5th critique: full width adder/byte arithmetic

- The logic to update RD here requires a full 32 bit or 64 bit adder. That's an awfully wide adder you need there. Yes, you may be able to recycle the adder you have, but even if you get away with that it's control logic and another input on the mux feeding the adder. If you use start/length and limit length to less than all of memory, the subtractor to update "bytes remaining" at least doesn't have to be full width (though I wouldn't build that either since I'd never implement ranges in CBOs).

- The arithmetic must also go down to the byte level (e.g. if start=13 and stop=130 with a 64 byte cache line you have to get from 13 to 63, then from 64 to 127, and the from 128 to 130). These distinct sizes mean you need some smaller adder/subtractor to figure out the unique input to feed to the adder/subtractor that is moving the "current" address or subtracting from the "number of bytes left".

- The better question is why gum up the pipe with anything at all? Just don't. This sort of thing is MUCH better done in software using the existing RISC-V core you already have.

6th critique: updating state inside the instruction.

- Why bother having any update of state inside the instruction (e.g. moving RS1 along to the next cache line boundary)? It just gums up the pipeline with extra gates and math (actually, there was a reason here, but that will be the next critique).

- Ideally, we'd like to have the cache control instruction update no register at all (and until we deal with line size changing dynamically, that works) or at most load RD with the cache line size for the block that was manipulated (this will be useful later for dealing with dynamically changing line sizes).

- Put another way, we'd like to get all the state update and tracking of where things are in the range moved out of the pipeline and put into the software/loops that surround the CBO. Arguably in a RISC machine that's where it belongs.

- We were building a RISC machine here, right?

7th critique: Sequencers.

- So why did we wind up with a requirement that RD (really RS1) get updated with math to move the base address along and (to a lesser extent) that RD must equal RS1?

- This was to allow implementations to implement a hardware sequencer that would directly execute the CBO for the entire range without ever falling out and taking the branch instruction. The hardware would directly re-execute the CBO while updating RS1 as it went along to cover the whole range. If a page or protection fault occurred in the middle, the handler would get invoked, clean up the problem or kill the program, and return to the instruction and keep going. External interrupts could also be satisfied. In all cases, RS1 would just tell you where to pick back up.

- But for that to work, you must have RS1=RD to fit in a classical RISC pipeline (can only writeback one register - RS1 here). If you try to use start and length in two registers instead, you'd have to update two registers per pass -- a big NO, NO in a classical RISC pipe. Updating two registers in a software loop is easy. Just don't build a sequencer.

7th critique: Sequencers, cont'd.

- Why build a sequencer?
- I'll contend there's not a good reason to do so.
- The thing that is going to dominate CBO loop performance will be the time it takes to propagate and process a CBO. Especially if the CBO leaves the local core and gets onto any sort of interconnect. The CBO will take much more time than a RISC-V core will take to do some arithmetic ops, and a branch.
- I believe a sequencer will simply fill up the store queue with CBO ops slightly faster (maybe) than the software loop would. Not terribly helpful.
- And certainly not a reason for start/stop, RD=RS1, wasted opcodes, and using RS2 up now.

7th critique: Sequencers, cont'd.

- We already have a big, terribly capable sequencer that we paid good money for: the RISC-V core. Use it! ☺.
- Implement all the sequencing using a software loop. Sequencing in software re-compiles much faster than taping out a chip again. Given that the CBOs will take longer to process than the looping code, it's essentially just as fast.
- It's also easier to verify. The CBO instruction does one thing and can't have hidden sequencing behavior with possible corner cases.

Not going to try to rehabilitate the current CBO

- In this presentation we won't try to rehabilitate the current suggested range-based CBO in any way. We'll simply move on to the cache-line based one-parameter CBOs we suggest RISC-V adopts and a different set of memory range instructions that RISC-V might adopt.

- The currently suggested CBO is too poor and brittle a foundation for the baseline CBO instructions RISC-V should have. Basing it on ranges that can't always work and trying to shoehorn a sequencer capability into it costs too much.

- Be sure to read through to the end of the proposal. We'll suggest range-based instructions (which are not CBOs) that will provide all the portability and hiding of cache line sizes that is ultimately possible.

Part IV : Initial proposal for replacing the range CBO.

Initial suggestion for a different CBO.

- The initial form of the new recommended CBO instruction could be an R, S, or B-form instruction if eventually supporting base+offset addressing and an I-form if supporting reg+immed addressing. Initially, no register is updated by the instruction. (To support dynamic line sizes later, we'll need to update an RD, and at that point, we'll need an R-form):

CBO.<op> RS1, RS2	(S, B or R-format)
CBO.<op> RS1, imm	(I-form)

  Where:

  - "<op>" is the type of CBO: flush, clean, invalidate, allocate, zero etc.
  - The byte address is formed according to the rules for the instruction form.
  - The formed byte address is interpreted as a byte somewhere within the enclosing aligned cache line which is affected by the CBO.
  - No register is updated.

The current suggested CBO: usage in a loop.

- Left as an exercise for the software folks using the CBOs.

- Now that that the CBO has no pretensions of looping in hardware and all the updating of registers to control the loop is out in the code, the ISA need not bother with this. It’s up to the software.

- Software is free to do whatever they want to specify the range (e.g. start/stop vs. start/length). Software is also free to handle the weird corner cases of start=stop or length=0 or the case of start > stop in whatever way is appropriate to the circumstances.

- Using CLCS, software can easily determine the prevailing line size and do the “right thing”. The line size doesn’t change (until a later section of the presentation).

Characteristics of these CBOs.

- They are SIMPLE. The CBO touches one cache block at the computed address, and we're DONE. We are building a simple RISC machine here, right? (seems I heard something about that).

- The CBO is well defined for any byte address in memory (manipulates enclosing line).

- Defining the ordering of these CBOs in the memory model is easier than for ranges.

- Much simpler to verify -- separates concerns between the CBOs and the arith/branching for the sequencing because CBO implementations can't (architecturally) build a sequencer. There are no sequencer corner cases the CBO implementation may have that your verification needs to worry about. The sequencing is all normal integer instructions you're verifying independently anyways.

Wait a minute?

- You might be asking… “Hey, wait a minute, that’s the same old CBO everyone has done for 30 years! What are you trying to sell here? That can’t possibly be portable and solve all those problems I have with the older cache control instructions? That violates RISC-V’s 11th commandment: thou shalt do CMOs using ranges!”.

- You’d be right that it is the same old CBO everyone has done for 30 years. That’s a good thing. It’s not likely to bite you in unexpected ways. There is a lot of experience with this.

- However, it will turn out that all the portability and hiding of line sizes that can be accomplished can be accomplished using CLCS, those CBOs, and normal stores (if the line size does not vary dynamically -- we haven’t forgotten about you Bruce ☺). If the line size does vary dynamically, a few additional minor tricks are required, but not much (those are covered in a later section).

Part V : The limits on portability and cache line size ignorance.

Transparent portability in all cases is impossible.

- As just one example, assume we have a program whose intent is to INVALIDATE (dcbi) a region consisting of three 16-byte cache blocks starting at address 16.
- That program can never be run on a machine that has a line size larger than 16.
- There is no way to INVALIDATE (dcbi) a partial cache line and any attempt to INVALIDATE more than the desired region of memory (bytes 16-63 inclusive) can corrupt the program.
- This one counterexample alone (there are several others) is enough to say that it’s impossible to safely hide the line size behind a range for all CBOs and all their various uses. It cannot be made to work.
- So, why, then, are we forcing a range-based interface onto inherently “range-unsafe” CBOs?

Maybe all hope is not lost? What if it's DCBZ?

- Let's assume the program is instead doing three ZERO (dcbz) CBOs to zero out three 16-byte blocks of memory starting at address 16.
- Again, if we can only use ZERO (dcbz) CBOs, you'll never run that program on a machine with a cache line size larger than 16 bytes -- it would write extra bytes to 0.
- However, if all you care about is zeroing the bytes in memory (and you're not, say, trying to clear ECC errors out of specific cache blocks), then one could conceptually achieve the effect with stores instead of CBOs on a machine with a line size larger than 16.
- But once you start doing things other than CBOs to make it work, it's not really a CBO anymore. The only CBO (of the ones discussed so far) that can be partially "faked" with other instructions is ZERO (dcbz).

What about FLUSH (dcbf)?

- Let’s assume the program is doing a FLUSH (dcbf) to flush out three 16-byte blocks of memory starting at address 16.
- Again, if we can only use FLUSH (dcbf) CBOs, you’ll never be able to precisely flush the 48-byte region on a machine with a cache line size larger than 16 bytes if that’s your intention.
- However, unlike ZERO (dcbz) or INVALIDATE (dcbi), sometimes it’s safe for FLUSH (dcbf) to “over-flush”. For example, if the machine is fully coherent (i.e. all writes, including DMA, go through a coherence protocol and all reads, including DMA, go through the coherence protocol), you can do a FLUSH (dcbf) to anywhere in memory at any time and it won’t hurt anything. In fact, you can NOP the FLUSHes in such a machine ☺.
- So, here we can safely overachieve by doing a set of larger line size FLUSHes that cover the 48-byte region (e.g. two 32-byte flushes at address 0 and 32 if the line size is 32).

A case that is portable using only CBOs.

- There is a case where a memory region may be processed by any CBOs in a portable fashion (i.e. write the code once and it works on any cache line size machine):

  The length of the region of memory to be processed is a non-zero integer multiple of the length of the largest cache line size the program will encounter, and the starting address of the region is aligned to the size of the largest cache line size the program will encounter. (also: the line size does not change dynamically).

- In this case, the program merely needs to obtain the cache line size (use CLCS) and loop over the buffer doing CBOs of the system's size starting at the buffer starting address. Whatever line size the system is using will completely cover the reigon with no overlap.

- In the extreme case where you can only use CBOs, you insist on using any CBO for any circumstance, and you insist on running the code on any machine, your regions must start on 4096 multiples and be sized to a multiple of 4096 (largest line size is 4096).

So clearly, there are limits to this

- So clearly there are some immovable limits in terms of alignment and length to safely applying arbitrary byte ranges to CBOs.

- Applying ranges safely may only apply to a subset of a given CBO's function (e.g. a ZERO (dcbz) used to write zeros vs clearing out an ECC error

- What is the boundary between range-safe and range-unsafe?

Characteristics of CBOs.

- To frame the conversation about when CBOs can be range-safe, it will be helpful to define some characteristics of CBOs:

  Destructive: A CBO is destructive when applying the function of the CBO to any part of any cache block other than the one called for could lead to erroneous program behavior. The destructiveness of a CBO can vary depending on the intended usage of the CBO or the characteristics of the system.

  Byte-Emulatable: A CBO is byte-emulatable when the full function or some useful subset of the function of the CBO can be emulated to a byte level in an implementation agnostic way using instructions other than CBO’s. The byte-emulatableness of a CBO can vary depending on the intended usage of the CBO (i.e. which subset of the CBO’s behavior is needed) or system characteristics (coherent or non-coherent system).

Destructive CBOs

- Which CBOs are destructive? **All of them**… depending on how you use them:

Name	Description.
FLUSH (dcbf)	Non-Destructive: When the machine is fully coherent (all reads -- cores and DMA -- participate in the coherence protocol and find the latest copy of the block wherever it may be). In such a machine you can NOP the FLUSH (dcbf) operation and software can't tell. Destructive: As an example, any time when the machine is not fully coherent. FLUSHes that "overreach" past their boundaries can destroy data managed by other threads (see Appendix).
CLEAN (dcbst)	Non-Destructive. Same as FLUSH (dcbf). Destructive: Same as FLUSH (dcbf).
ZERO (dcbz)	Destructive (zeroing any part of any block other than the one addressed can always corrupt the program).
INVALIDATE (dcbi)	Destructive.
TOUCH_RD (dcbt)	Non-Destructive. Same as FLUSH (dcbf). Destructive: In at least machines doing software cache coherence. This moves the line into the cache just like a load would.
TOUCH_WR (dcbtst)	Non-Destructive. Same as FLUSH (dcbf). Destructive: In at least machines using software cache coherence This moves the line into the cache just like a store would.
ALLOCATE (dcba)	Destructive.

Byte-emulatable CBOs

- Which CBOs are byte-emulatable? And for what subset of the function?

Name	Description.
FLUSH (dcbf)	Not byte-emulatable -- There is no meaningful byte-level definition of FLUSH (dcbf).
CLEAN (dcbst)	Not byte-emulatable -- There is no meaningful byte-level definition of CLEAN (dcbst).
ZERO (dcbz)	Byte-emulatable if one only cares about getting zeros in the memory locations (use stores). Not byte-emulatable if the intended usage of DCBZ is to clear a RAS error from a cache block holding a given address.
INVALIDATE (dcbi)	Not byte-emulatable.
TOUCH_RD (dcbt)	Not byte-emulatable.
TOUCH_WR (dcbtst)	Not byte-emulatable.
ALLOCATE (dcba)	Not byte-emulatable.

Safely applying byte-address ranges to CBOs

- CBOs are inherently block based operations and don't generally apply to arbitrary byte ranges. Given that, the question to ask is "Are there cases where arbitrary byte ranges can safely be applied to a CBO's function or some subset of the CBO's function".

- The answer is you can't apply arbitrary byte ranges to all the functions of any of the listed CBOs. However, if you're willing to limit arbitrary byte ranges to a subset of the CBOs and their functions, it is possible if one of the following conditions is met:

  1. The CBO is non-destructive. In this case, we can "carpet-bomb" the arbitrary byte range with a covering set of CBO block operations that encloses the range, but that may also touch other locations.
  2. The CBO is destructive but also byte-emulatable. In this case, we can use the byte-emulatability to process the bytes that aren't in a full cache line and use the CBO to manipulate the full cachelines. No bytes outside the requested range are affected.

- Moral of the story: things can be range-safe ONLY in a strict subset of the possible cases.

Memory Range Operations (MROs).

- When we're working with ranges, the operation is no longer a Cache Block Operation (with emphasis on the "**Block**" part). We're now dealing with a byte range and not a single block.

- We will call these things "Memory Range Operations" (MROs) to distinguish them from your more ordinary decent CBOs.

- Separating MROs out from CBOs as distinct functions also allows for a flexible software-based implementation of MROs (later) and makes it clear to software where it's safe to use an arbitrary byte range and where it is not safe to use an arbitrary byte range.

- MROs are implemented using a series of CBOs and/or stores.

Possible Memory Region Operators (MROs)

- Possible Memory Range Operations (with the CBOs so far):

Name	Semantics:
FLUSH_MEM	Will FLUSH the cache lines containing the desired region which may affect bytes outside the region (each flushed cache line will contain at least one byte in the range). The set of cache line FLUSHes is not atomic nor are the individual FLUSHes ordered in any way.
CLEAN_MEM	The semantics are the same as FLUSH_MEM with the exception that the line is CLEANed instead of FLUSHed.
ZERO_MEM	The bytes in the specified region are written to zero. The writes are not atomic and may occur as if they were written by an unordered series of byte stores covering the region.
TOUCH_RD_MEM	Same semantics as FLUSH_MEM, but the affected cache lines are read as if by a load instruction.
TOUCH_WR_MEM	Same semantics as FLUSH_MEM, but the affected cache lines are read as if by a store instruction.

- Note how the semantics for the MROs carefully require the underlying CBOs to be used in a non-destructive way. For ZERO_MEM, any ZERO (dcbz) CBOs internally used must be aligned and fully contained in the region. For the other MROs, the CBOs are explicitly allowed to touch other bytes as necessary to cover the region requested.

Many ways to possibly implement MROs.

- Not at all. You may decide you don't need MROs and MROs should be optional.
- As a single instruction that always traps to an "alignment" handler and uses code in the handler to sequence and implement the MRO's semantics. Obviously, this instruction would NOT have a loop around it. (We'll use this one to illustrate MROs). Note that "alignment" there could be the existing alignment hander we add MRO support or some new exception specific to supporting MROs.
- As a single instruction with all the sequencing implemented in hardware (PLEASE DO NOT do this. It's hideously complex and you should lose you RISC processor designer's license).
- As a single instruction that in some cases (easy ones) might be implemented in hardware and will trap to an "alignment" handler for the more complex cases (while this is possible, it might still get your RISC processor designer's license revoked).

Many ways to possibly implement MROs, cont’d

- The MROs could just be purely be canonical software routines with no particular single instructions used to call them:
  - Provide MROs as library routines that software can access instead of instructions.
  - As a suggested canonical piece of code in an appendix in the ISA manual. Software would be required to directly implement the routine, as necessary. (#define macro, anyone?).
- This would also allow MROs to be a separate optional extension. Implementations could pick and choose pieces of the MROs and cast them purely in software.

Alignment handlers for MROs: general info.

- If the MRO is implemented in a software routine, the debate between start/stop and start/length to specify the range is less urgent. The handler/subroutine/macro code is a full program that runs once per MRO and can do the appropriate math and do conditional tests at the beginning of the alignment handler to weed out the ambiguous cases.

- For these examples, we'll just pick the start/stop formulation since it's a bit easier to describe. Changes for a start/length formulation should be obvious from these descriptions.

- While in the handler emulating the MRO using CBOs and/or stores, a page fault, protection fault, or another alignment fault (ZERO (dcbz) to a cache-inhibited page can cause this) may occur (assuming the MRO is done in an exception handler). The exception handler machinery needs to be able to handle this nesting of exceptions. According to some experts who code this sort of thing for Power linux, this is not difficult or even new to them. Your mileage may vary or may vary in some other OSes? Even if this does not work in your OS, the MROs can be provided as code you replicate or as a subroutine.

MRO alignment handler for all but ZERO_MEM

- For the MROs built out of non-destructive CBOs, we just move from start to stop doing an contiguous set of CBOs as described below. Affecting extra bytes is allowed by the definition of the semantics of the underlying non-destructive CBOs.

- The handler returns immediately if start >= stop. That’s dumb and we cut these cases out. Otherwise CLCS is used to obtain the cache line size (which can’t change). The lower order bits of the start address that address within a cache line are set to zero (called “normalizing” from now on) to produce an aligned cache line address at the start byte of the cache line covering the beginning of the range. This address is the “current address”. The stop address is also normalized.
- The CBO is done to the current address.
- The cache line size is added to the current address. If this addition overflows (did last block in memory) or results in a value greater than the normalized value of the stop address, the handler is done and returns. Otherwise a new CBO is executed at the current address and the process repeats.

MRO alignment handler for ZERO_MEM

- ZERO_MEM is somewhat more complex in that stores be done for the pieces of the region that don't consist of a full cache block, but otherwise the idea is the same: we proceed left to right filling in the bytes with ZERO (dcbz)s where we can and stores where we can't:

- The handler returns immediately if start >= stop. That's dumb and we cut these cases out. Otherwise CLCS obtains the cache line size and the "current" address value is set to start.
- The handler determines if the "current" address is at a cache line boundary and there are enough bytes left to consume a full cache line. If so, a ZERO (dcbz) at the current address is done. Otherwise, a series of stores is done to zero the bytes until the next cache line boundary or the end of the range whichever comes first.
- The number of bytes in a cache line (if the code did a ZERO (dcbz)) or the number of bytes stored is added to the "current" address. If this addition overflows (did last block in memory), the handler returns. If the result of the addition is greater than (unsigned) the stop address, the handler returns. Otherwise the process repeats.

MROs implemented in handlers/SW are adaptable

- Let’s imagine someone in the back of the audience stands up and says I need an “ordered ZERO_MEM” (First Law of RISC-V: No matter how specific or obscure an idea is… it’s key to some RISC-V business plan somewhere. ☺).
- After some more discussion, we learn that the “ordered ZERO_MEM” insists that the updates to zero happen in ascending order.
- We an easily accommodate this by adding FENCEs in the right places in the alignment handler. You really don’t want to be accommodating all these variation in hardware.
- If the request is too obscure to put it in the ISA directly, this user could use a user-defined opcode, have the implementation trap on that, and use a variation of the canonical ZERO_MEM handler with FENCEs as the handler code.

Handling cache-inhibited pages in the handler.

- What happens if the alignment handler is emulating the MRO and strays into (or starts in) a page that is cache-inhibited?

- For FLUSH_MEM and CLEAN_MEM it’s not a problem. The underlying CBOs ignore the cache-inhibited attribute and perform the safe function (see Part IX: trivia below).

- For MEM_ZERO, the alignment handler is either executing stores (which are simply cache-inhibited now) or is executing a ZERO (dcbz) CBO. If the ZERO (dcbz) targets a cache-inhibited page, the alignment handler takes a nested alignment interrupt for the ZERO (dcbz) in the alignment handler code emulating the MEM_ZERO. The alignment handler knows that if an alignment interrupt is caused by a ZERO (dcbz), that interrupt is due to a cache-inhibited page. The handler executes a series of stores to fill out the current cache line size hole in memory at the ZERO (dcbz) CBO address and returns back into the alignment hander which continues emulating the rest of the MRO.

Part VI: Bruce Hoult’s been very patient. Time to deal with dynamically varying line sizes. (as much as we can)

This is not the common case.

- Mixed-line size machines are NOT the common case. Not even close. So while it’s worth discussing ways to accommodate this issue and possibly even accommodating the issue, it should not dominate the CBO or MRO architecture.

- The only place it makes sense to support time-variant line size systems is for the MRO operations. If a CBO is destructive, there’s no way to safely use it if the line size can change at any time. So, the only things that are left are the MROs based on non-destructive CBOs (all but ZERO_MEM) and the ZERO_MEM MRO which can be made safe using a new “safe” version of ZERO (dcbz) CBO that is used in the alignment handler for ZERO_MEM.

- Now we’ll look at how to modify the alignment handler loops to handle time varying line sizes in MROs.

A brief detour - mixed cache line sizes are hard

- Machines with mixed line sizes require a good deal of support to even begin to work, long before you add CBO’s and MRO’s to them.

- To a poor first approximation, whenever locations are accessed by an agent with one cache line size, any other agents with differing cache line sizes would FLUSH the lines to memory first. This would ensure that any given set of locations was only held by agents of a uniform cache line size at any given point in time. Other more aggressive schemes involving “smaller” accesses affecting partial cache lines in larger caches where possible and “larger” accesses being internally broken into several smaller accesses at agents of a smaller line size are possible. These techniques allow for concurrent differing line sizes for a set of locations but are much more complex.

- Essentially these techniques are needed to allow any given HART to operate as if the HART’s native line size is the line size in the entire system. We don’t try to motivate any of this here and it’s assumed the system provides this already.

Fixing MROs based on non-destructive CBOs

- The good news about non-destructive CBOs is that they are non-destructive, and you can execute them to any memory location at will without harming anything.

- So, even if the line size changes dynamically while working through the range the MRO is to manipulate, the worst that will happen is the MRO might “over-do” it a bit too much or repeat some locations, but that breaks nothing.

- However, we do need to know what size cache block the non-destructive CBO touched, so the alignment handler loop can advance the “current” address the proper amount.

Have the CBOs return the line size.

- To support time-variant line sizes, the following non-destructive CBOs are modified to return, in RD, the line size the CBO manipulated (CLCS can't be used. The line size may change at any time, so the CBO itself must report the line size manipulated).

    – FLUSH (dcbf)
    – CLEAN (dcbst)
    – TOUCH_RD (dcbt)
    – TOUCH_WR (dcbtst)

- INVALIDATE (dcbi) and ALLOCATE (dcba) can't be safely used in a machine that can change the line size for a HART at any time, so we don't modify value. The programmer must know the line size and that it is stable before using these CBOs.

- With this extra information, we can modify the MRO alignment handler to tolerate varying line sizes for these operations.

New alignment hander for non-ZERO_MEM MROs

- To support time-variant line sizes, the alignment handler for the non-ZERO_MEM MROs uses the new CBOs that return the size of the line manipulated at each CBO.

- After each CBO in the loop, the CBO returned cache line size is used to normalize (see above) the “current” address in the loop so that it points at the first byte of the line actually manipulated by the CBO. The returned line size is then added to the normalized “current” address which moves to the next byte not manipulated. If the addition overflows (did last block in memory) or if the address is greater than the value of the stop address, the loop ends and the handler returns. The handler can also do the initial CBO without looking up the line size with CLCS and normalizing the start address by setting current to start. The CBO returned line size value will move the “current” address to the next cache block properly.

- Since the current address adjusts to the starting address of the block immediately after the block manipulated at each step, the line size can change at every step in the loop and the “current” address will accurately track across the blocks until the range is covered.

Fixing the ZERO_MEM MRO is trickier.

- The CBO used in ZERO_MEM is destructive. Therefore it cannot be executed unless the line size it will manipulate matches what the loop expects. Otherwise, the ZERO (dcbz) could alter memory locations not in the range. Currently, however, there's no way to know the size without executing the destructive CBO (CLCS doesn't help since the line size can change after the CLCS and before the ZERO (dcbz) executes).

- To solve this problem, we create one additional CBO called SAFE_ZERO (has no analogue in Power). The SAFE_ZERO CBO takes an address in RS1 and an expected size in bytes in RS2 (yes, we lose base+offset addressing for this one, but we can live with that since this instruction is only intended for use in the alignment handler for a ZERO_MEM MRO).

- When the SAFE_ZERO executes it compares the line size to the expected value. If they do not match, the SAFE_ZERO does not alter memory. The SAFE_ZERO always returns the current line size in RD. Think of it as a "conditional ZERO" that reports the current line size.

New alignment hander for ZERO_MEM MRO

- To support time-variant line sizes, the alignment handler code for the ZERO_MEM MRO is largely the same except in how the ZERO (dcbz)'s in the alignment handler are processed.

- First the ZERO (dcbz)'s are replaced with SAFE_ZEROs that use the line size obtained from CLCS as a best guess as to the line size for the first SAFE_ZERO.

- Each time the SAFE_ZERO is processed, if it returns the expected line size, processing proceeds as it did in the non-time varying MRO handler. Otherwise, stores to the end of the returned line size or the end of the region whichever comes first are used to zero memory, and the current line size is set to the value returned by the SAFE_ZERO. If the stores finished the region, the routine ends. If not, the loop continues at the current address.

- Essentially any DCBZ that fails is replaced with stores to the end of the region or the end of the cache line reported by SAFE_ZERO and we pick up from there.

There are other ways to code the MROs

- Doubtless there are other ways to code the MRO alignment handlers.

- The ones described above (for both time-variant capable and not) are just suggestions. Since the alignment handler code implicitly defines the semantics of these instructions, there should be a through discussion of the exact details of the routines.

- The time-variant capable routine for ZERO_MEM as coded is guaranteed to eventually finish. At each step where the SAFE_ZERO fails it's replaced unconditionally with stores. It could be coded to try a new SAFE_ZERO based on the returned size. When the line size changes, this might be slightly faster than mowing down bytes with stores. However, that may never finish if the line size is changing faster than the SAFE_ZERO is reattempted, so such a handler must eventually give up on trying to match the varying line size with SAFE_ZEROs and resort to stores. The example given here does that for all failed SAFE_ZEROs.

New semantics for the MROs

- The semantics for the time-variant line size MROs also need a small change from non-time-variant MROs.

- The non-destructive MROs need to say locations may be affected more than once. The non-time-variant MRO loops never touch a location more than once, but the time-variant MRO loops can repeat locations.

- Similarly, the ZERO_MEM MRO loop semantics need to be changed to say they can zero some locations in the range more than once. The loop shown here won't because it reverts to stores immediately whenever a SAFE_ZERO fails, but if the loop were to try SAFE_ZERO again at a different size, some bytes would be written more than once (I opted for the simpler loop for discussion here, but the production code may not want to do that).

Costs of supporting time-variant safe MROs

- So what's the final cost to support MROs in time variant line size machines?

  1) Need a SAFE_ZERO that takes an address in RS1, expected size in RS2, and only writes memory if the expected size equals the current system size. SAFE_ZERO always returns the current line size of the machine.
  2) The FLUSH, CLEAN, TOUCH_RD, TOUCH_WR instructions are modified to return the line size of the line actually manipulated.
  3) The alignment handlers for the MROs will need to be re-coded using the new instructions.

- Are these costs worth it? That's a worthwhile debate. It's not clear than a fully encompassing abstraction for time variant line-size machines is even possible. In an incoherent machine, I believe it will be impossible to naively move threads across different line sizes unless the software is written assuming the largest line size in the machine could (see the appendix below). Also, what value does CLCS return in a multiple line size machine? There are so many issues here.

Part VII: PREFETCH64.. really? Must we?

Can we please get rid of this?

- Adding the PREFETCH64 leads to at least two issues:

  1) We now have yet another “shape” for a CBO command, so that’s inconsistent.

  2) Specifying the cache line size directly in the instruction is a mess for any implementation that has the audacity not to match that line size. If the line size is smaller, this command needs to cause multiple line accesses. If the line size is larger, the prefetch affects partial cache lines.

- Seems that the TOUCH_RD_MEM or TOUCH_WR_MEM MRO gives you all you could want or need here? Or for that matter use TOUCH_WR and TOUCH_RD and CLCSes to tell you the line size you have (if you don’t time-variance).

Part VIII: Summary

Range-based CBOs are a leaky abstraction.

- A range-based interface for CBOs is a leaky abstraction. It cannot be made safe for all use cases and machine designs.

- The current CBO proposal is chasing all the far corners of anything someone might want to do with CBOs and trying to fit all of it into one abstraction. It’s what my dearly-departed father used to call a “blivit”: ten pounds of manure in a five-pound bag. It just doesn’t fit.

- There are large amounts of users that will need to use CBOs in ways such that the line size cannot be hidden from them. This code has no pretensions of being generally portable. This might be most of the users for CBOs.

- These users will need a simple set of Cache Block Operations (CBOs) that manipulate the cache blocks clearly and individually. Their uses of CBOs must be supported directly and can’t be delivered safely in all cases with range-based CBOs.

Use MROs for portability and line size hiding

- However, if the intended use of the CBO is non-destructive or if the CBO can be emulated at a byte level by instructions other than CBOs, it is possible to apply arbitrary byte ranges to the desired use of the CBO. For these CBOs, we propose the creation of range-based instructions (or library routines or subroutines) called Memory Range Operations or MROs. We also propose mechanisms to make MROs time-variant line size capable.

- When using an MRO, software is guaranteeing that the desired use of the cache operations within the MRO is “safe” to abstract away to arbitrary byte ranges. We suggest that the MRO be implemented in software that performs an appropriate series of CBOs and/or stores to perform the function of the desired CBO across a range.

- By splitting the problem into two interfaces, we recognize the cases where CBOs can’t be safely handled with ranges while still providing an MRO interface that allows those cases where byte ranges work to be addressed in a portable way.

Line size discovery.

- We also propose a CLCS instruction which provides a line size discovery mechanism that is as accessible as the CBOs themselves (even on the smallest microcontroller).

- A line size discovery technique will have to be provided no matter how the CBOs are built.

- For destructive CBOs, software needs to know the line size ahead of time for those to be safely usable at all. Every CBO has a destructive use, so software must have a way to discover the line size to use the destructive CBOs (unless we're expecting software to just "know" the value and hard code it?… hardly portable).

Part IX: Random loose ends on CBOs.

Random bits of trivia.

- This section will contain random bits of trivia about CBOs.
- New items will be added as necessary.

CBOs on cache-inhibited pages.

- It’s important to pick reasonable behaviors for CBOs for cache-inhibited memory regions.
- For ALLOCATE (dcba), TOUCH_RD (dcbtf), and TOUCH_WR (dcbtst), it’s easy. They’re NOPs. All of these try to establish a cache line and that doesn’t work on a cache-inhibited page.
- For ZERO (dcbz), you can take an alignment interrupt that will write zeros with stores. This is safe because the only use of ZERO (dcbz) in a cache-inhibited page is to write zeros.
- The FLUSH (dcbf) and CLEAN (dcbst) operations ignore the cache-inhibited control and flush/clean any caches with the line. This allows switching from cacheable to cache-inhibited. ZERO_MEM the region first, change the translate to cache-inhibited, and then FLUSH (dcbf) the region to remove cached copies that may overwrite memory on cache-writebacks. The page is now safely cache-inhibited. INVALIDATE (dcbi) also ignores the cache-inhibited control.

- Change Log

Change Log

- 09/13/2020 - Version 0.9
  - Pre-release review draft.
- 09/14/2020 - Version 0.91
  - Incorporates some initial feedback.
- 09/17/2020 - Version 0.94
  - Added Appendix section.
  - More clean up based on feedback.
- 09/20/2020 - Version 1.0
  - Final clean up round.

- Backup/Appendix slides.

- Appendix A: All CBOs destructive?

Are all the CBOs really destructive in some case?

- It can be difficult to believe that all the CBOs are destructive in at least some use cases (except ZERO (dcbz) which is obviously destructive if you go out of the bounds).

- Is FLUSH (dcbf) destructive? The touch instructions? In some uses cases they are destructive. Examples are given below.

- Even if the specific examples below are overturned, the broader question remains. Is there a proof for each CBO that it isn't destructive? That sort of proof can be hard to get right and is always subject to the undiscovered counterexample. Discretion might be the better part of valor here.

How is FLUSH (dcbf) destructive?

- In a fully coherent machine (i.e. informally, a machine where all accesses from processors and DMA go through a coherence mechanism and always see the “latest” value for any location), we don’t believe FLUSH (dcbf) is destructive.

- In a machine that does not have full coherence, FLUSH (dcbf) can be destructive.

- Consider a machine with two HARTs on different single-threaded cores. Each core has a write-back L1 cache, a bus, and then a main memory. This machine has no cache coherence. So writes by one HART only get to memory by being FLUSHed (dcbf) by that core or though a cache-writeback. Each core writes memory independently. Data is loaded into the cache only due to LD or ST instructions or via TOUCH_RD (dcbt) or TOUCH_WR (dcbtst) instructions -- no hardware prefetches for example. CBOs only affect the local cache, they aren’t snooped by the other caches (the caches are incoherent after all).

How is FLUSH (dcbf) destructive, cont'd?

- Assume these threads:

```
Thread0                                   Thread1

ST mydata_a, 1   // mydata_a is at addr 0-3    ST mydata_b, 2  // mydata_b at 4-7.
FLUSH 0-3        // flush bytes 0-3            FLUSH 4-7       // flush bytes 4-7.
```

- Now assume we put this program on a machine with 16-byte cache lines.

- The following occurs: Thread0 executes the store, and then executes the flush and pushes data back to memory. Thread1 does its store, executes the flush and pushes data back to memory. Thread1's write back will obliterate Thread0's store. WHOOPS. That's not good.

How is FLUSH (dcbf) destructive, cont'd?

- The hardware guy/ISA architect would look at this and go "The data is too close together -- you can't do that, it's a bad program. Even without the DCBFs, normal writebacks of the line with destroy the data. You have to spread those things out!"
- The programmer then goes "Spread them out? I told your hardware to flush 0 to 3 on the one thread and to flush 4-7 on the other thread. I did NOT say blow away the other addresses? Where does the hardware get off doing that? And how was I supposed to know the cache line size anyways? You've hidden it all behind ranges?".
- Moral of this little story: ranges can never fully hide the line size from the programmer in all cases.
- Our programmer huffs off swearing at the hardware guy/ISA architect and spaces the variables out to 16 bytes apart…. and all is well… for a time.

Six months later with FLUSH

- Software guy 1 has quit and works somewhere else now. Software guy 2 (we'll just number them ☺) has taken over.
- The program is ported to the new machine with 32-byte lines…. crashes. Same reason. Big shock to everyone -- after losing 3 weeks to figure the bug out -- because no one now remembers the first time this happened. Everyone believes that they are safe because the flush of 0-3 and 16-19 shouldn't be killing the other variable, right?
- That code is inherently non-portable (unless you space things 4096 bytes apart -- might be excessive in a small machine). If the ISA only has range-based CBOs it implies this non-obvious and draconian 4096-byte spacing requirement on programs to be portable?
- I don't think that makes any sense. The reason for ranges was to hide portability problems -- not inject new constrains that are non-obvious. Range-based CBOs can't fix this. DCBF is quite destructive in this case.

Is there even a consistent definition for FLUSH?

- Can we even make a workable definition for a range-base FLUSH (dcbf) CBO instruction?

- On the one hand, we'd like the instruction to NOT affect any locations outside the range we gave in RS1/RS2. Software guy 1 and 2 are happy. Hardware guy can't build this. FLUSH (dcbf) can only be built to work on cache line sizes. Happy programmers. Impossible to build.

- If the CBO is defined as a range and allowed to affect locations outside that range, programmer 1 and 2 can't use the instruction. Unhappy programmers, but happy hardware guy can build it.

- You can't meet both constraints in one range-base operator (for all the CBOs, not just FLUSH). To cope, we have an MRO, defined to be able to overreach, for the range cases. Programmer 1 and 2 know they can't use the MRO. They use the single cache block CBO and know their line sizes (both to use the FLUSH (dcbf) and to space the variables).

Incoherent DMA can bite you here as well.

- You can create the same sort of problem even if the caches are coherent, but DMA is incoherent.
- Imagine the core thinks it’s working on variables in bytes 0-31 and the programmer does a FLUSH 0, 31 to flush bytes 0-31 back to memory (“Hey it was a 32-byte line size when I wrote this program!” says programmer 3). DMA is writing to the block in 32-63 (“This really used to be a 32-byte machine!”).
- On a 64-byte line size machine, the processor does a store to bytes 0 to 3, pulling in a copy of the line from 0 to 64. DMA then writes bytes 32-63 in memory. Processor executes a FLUSH (dcbf) from 0 to 31. The line from 0 to 63 is evicted and obliterates the DMA data in 32-63. WHOOPS. FLUSH (dcbf) seems quite destructive.

What about CLEAN (dcbst) ?

- Same exact problem as FLUSH (dcbf). Same exact examples.
- Only difference is that the CLEAN leaves all previous non-modified copies in place, writes-back any modified copy, and changes the state of the modified copy to some non-modified cache state. The newly non-modified copy will never be written back in the future.
- The difference between CLEAN (dcbst) and FLUSH (dcbf) doesn't matter a bit for the examples here.

How is TOUCH_RD/TOUCH_WR destructive?

- How can TOUCH_RD (dcbt) and TOUCH_WR (dcbtst) be destructive? Again, in a fully coherent machine they aren’t. In a non-coherent machine they are.

- TOUCH_RD (dcbt) and TOUCH_WR (dcbtst) both fail in the same way, so you can use either example for one instruction for the other, so we only show a TOUCH_RD (dcbt) case.

- We have the same “machine” as the prior FLUSH (dcbf) example with two threads, but this time a producer-consumer example with the variables laid out like so:

  – data (piece of data being produced): 4 bytes at address 0-4.
  – flag (flag indicating data is ready): 4 bytes at address 64.
  – x (an unrelated piece of data used for something else): 4 bytes at address 32.

How is TOUCH_RD (dcbt) destructive?

- These threads are running on a 64-byte line size machine (coded for a 32-byte line size).

Producer

```
ST data, 1   // set data at 0-3 to 1
DCBF 0-3   // flush "data"
<fence>       // order data/flag.
ST flag, 1    // set flag at 64-67 to 1
DCBF 64-67 // flush "flag"
```

Consumer

```
DCBT 32-35      // get "unrelated" x (pulls in 0-63)
loop: LD flag      // get flag
        CMP flag, 1 // is it set?
        BEQ ldit      //  yes!!
        DCBI 64-67 // erase local copy to see update.
        BR loop     // go try again now.
 ldit:   LD data      // can't see update. DCBT loaded
                      // stale copy of data at beginning.
                      // No way for producer to update.
```

- Here thread1 does an unrelated DCBT that should have only loaded 32-63 (and it did on a 32-byte line size machine), but really loaded 0-63. We have a stale copy of data on consumer now. The producer's updates won't affect this copy. When thread1 reads the flag as 1 and then tries to read the data it will erroneously get the stale copy from the L1 cache.

INVALIDATE, ALLOCATE, ZERO?

- ZERO (dcbz), ALLOCATE (dcba), INVALIDATE (dcbi) are all obviously destructive.
- ZERO will write too many bytes if you go outside the range and ALLOCATE and INVALIDATE can corrupt data that shouldn't be corrupted if you go outside the asked for range.
- You might try to implement INVALIDATE by FLUSHing, but FLUSH is destructive.
- You could implement ALLOCATE (dcba) by doing stores for the partial cache lines, but that would defeat the whole purpose of ALLOCATE (dcba) which is to fetching from the memory controller. Also, the "extra" bytes (outside the range) fetched in these partial cache line(s) are just as unsafe on an incoherent machine as the "extra" bytes fetched by TOUCH_RD (dcbt) or TOUCH_WR (dcbtst). See the example on the prior slide.

- The End.