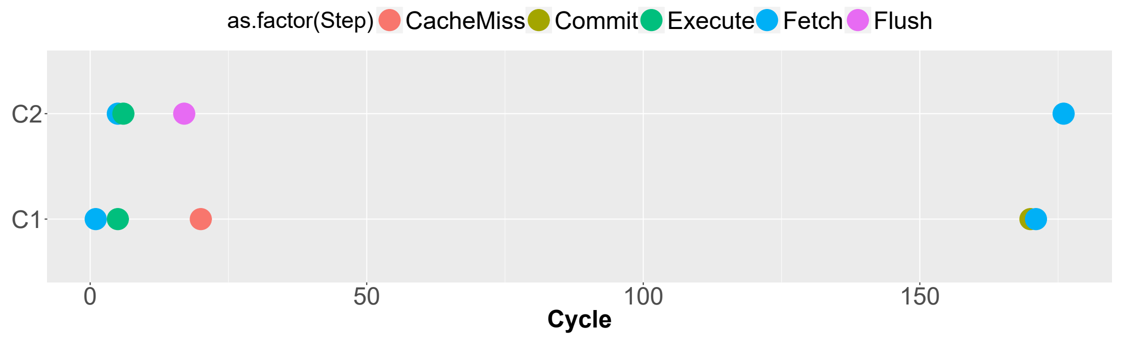

Figure 1: Mispeculation followed by a cache miss example.

1 Preface

Core Fusion allows 2 or more cores to work on the same thread by speculatively executing blocks in the same thread. To simplify explanations, in this report each core can, at best, fetch a single block and has an in order LSQ.

2 On Branch Prediction

In a 2 core fusion, the process fetching process is:

- Core 1 will fetch a block and decode it. This takes +/- 2 to 4 cycles.
- On the last decode cycle it predicts the next block and sends that PC to Core 2 which then will fetch that block.

Cores only ever forward a **single** prediction. If in the previous example Core 1 were to mispeculate then Core 2 would be flushed it would then have to wait for Core 1 to commit its block before fetching another one. In this case, mispeculation can cause a core-fusion to present no speedup.

To illustrate this problem, we can look at Figure 1, with each cores activity represented on the Y axis and the cycle count represented on the X axis. I don't have a particular set of blocks in mind, however any hyperblock can have a branch located in the middle of the block, we can imagine that in this scenario Core 1 predicted to execute a branch in the middle of the block and submits that request to C2. In Figure 1 Core 1 ends up mispeculating the branch, sending a flush request to Core 2 and then executing the rest of the block. If, as seein the figure, it were to have a full cache miss, this block could take up more than 120 cycles to finish executing; during this time Core 2 will be idle. Only when Core 1 commits, at cycle 170 will Core 2 ever be used again. In the example in Figure 1, the performance is no different than if we were to execute on a single core.

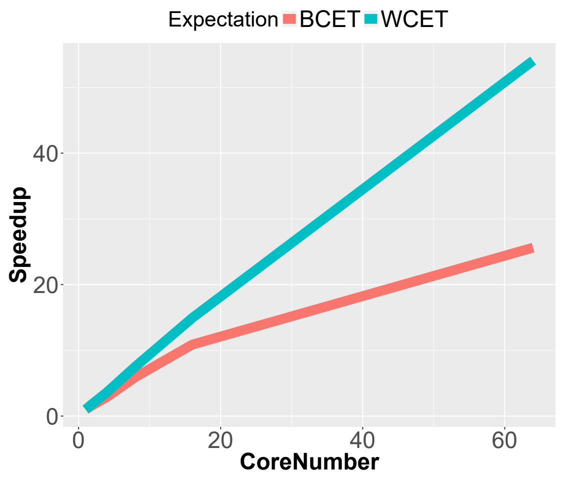

Figure 2: Executing a[i]=b[i]+1 for WCET and BCET.

Naturally, when extending this to more than 2 fused cores, this could further waste resources as if the original core mispeculates, then you will have every other core waiting for it to commit.

3 On Block Size

Block Size is important to ensure that Cores in a core-fusion are properly utilised and can give us a rough estimate as to the potential performance. However, to fully understand how useful a block is, we need to think about it in multiple terms.

- Best-case execution time.
- Worst-case execution time.
- Block dependencies.

Let's use a simple loop to illustrate the problem

```
#pragma unroll
 for(int i =0; i < 100000000; i++)
     a[i] =b[i] + 1;
```

We assume that the unrolling will maximise the size and that the block can loop on itself; since we can only have a maximum of 32 load/stores per block, let's just assume we have a block of about 60 ish instructions (load / stores + increment of b[i])

Let's imagine we have cores with a dispatch width of 2 and an inorder load-store-queue; the worst-case execution time for this block would be that each array access causes a full cache miss: 120 cycles. In this case, the worst-case execution time is going to be 2 cache misses + 16 stores (2 cycles each) + 16 adds + 14 loads (2 cycles each), so roughly 314 cycles. The best-case execution time is no cache misses, so 16 loads + 16 stores + 16 adds, so 80 ish cycles. Given this information, what can we extract ?

Figure 2 represents the speedup obtained via core-fusion compared to single core for BCET and WCET. As we can see, when the block is being executed at its BCET, the speedup is lesser than when we add more and more cores compared to WCET. This is due to the overhead of having a lot of cores in the core-fusion. Being able to determine how much that overhead is is non-trivial due to a few extra factors, mainly block dependency (and we need to take into account that blocks are not always the same).

In the case of this kind of for loop, you can estimate that the overall speedup will be:

$$\frac{AverageExecutionTime}{\frac{4*(Cores-1)+AverageExecutionTime+Dependencies*Cores}{Cores}} \tag{1}$$

In Equation 1 the constant 4 represents 3 cycles for another core to fetch a predicted block (1 cycle to fetch header, 1 to decode and 1 to send prediction) with an extra cycle for waiting for a previous commit to finish. The Dependencies variable represents the number of cycles required for a slave core to receive all the necessary data it requires from its master core to finish executing its block. In the case of our listing, this would be the value of i, which is negligeable as this can be immediately incremented by the master core and passed on to the slave core when the slave core fetches its block. As we can see, the overhead of core-fusion can be **non trivial**. This is also not taking into account that I currently consider only a 1 cycle network overhead, because this would also add to the overhead incurred by Core Fusion.

3.1 What does this tell us

Core Fusion is very sensitive to block size/execution time of blocks due to the overhead it will add just for existing. If you know you have a loop that is going to be prone to a lot of cache misses then the size wont matter because this will blow up the execution time of the block, making it easier to speed up execution. When you have a purely arithmetic block, this will then depend on the number of ALUs present on each core; for example if we have 2 ALUs per core, since each block can only be a max of 128 instructions, this can end up with having a WCET of 64 cycles which will limit the overall perfromance.

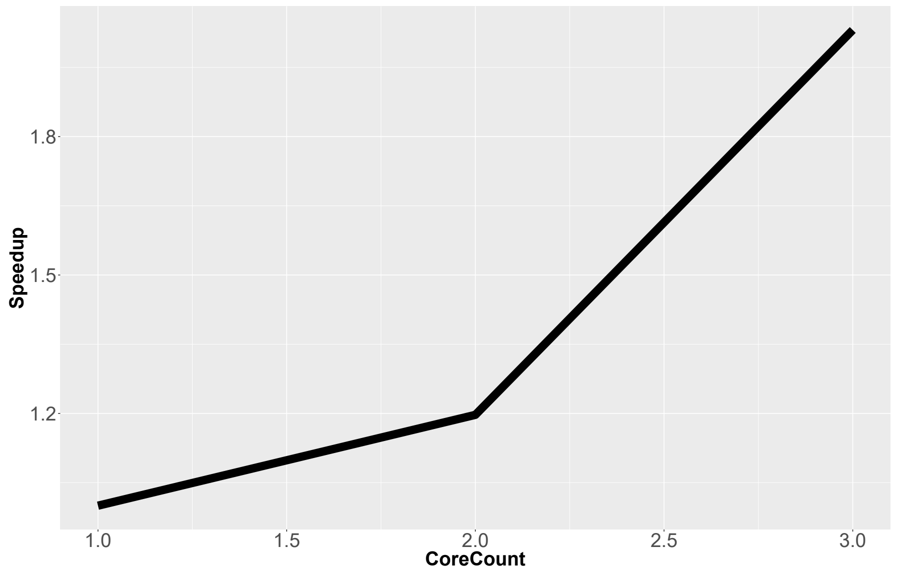

Figure 3: Executing c[i]=a[i]*b[i].

4 On Block Dependencies

I only briefly talked about dependencies in the previous section but I wanted to dig into it more right now.

Let's just focus on cache line dependencies.

If we have

```
#pragma unroll 4
    for (int i = 0; i  <100000; i++)
    {
        c[i] = a[i] * b[i];
    }
```

Where the arrays hold 64 bit integers. Figure 3 shows the speedup obtained on this loop; whilst it may look like nothing it is important to look at the speedup obtained by 2 cores, and how it sharply changes after that. A 2 Core fusion on this loop will give you a 1.19x speedup which is nothing, it picks up after this though, with a 2x speedup on 3 core fusion. Why does this happen ? The answer is somewhat straight forward, here is the EDGE block:

```
LBB0_5
        read b3l, r4
        ld t4, −16(b3l),           l[0]
        read b2l, r6
        ld t5, −16(b2l),           l[1]
        mul t6, t5, t4
        read b1l, r5
        sd t6, −16(b1l),           s[2]
        ld t7, −8(b3l), l[3]
        ld t8, −8(b2l), l[4]
        mul t9, t8, t7
        sd t9, −8(b1l), s[5]
        ld t10, 0(b3l), l[6]
        ld t11, 0(b2l), l[7]
        mul t12, t11, t10
        sd t12, 0(b1l), s[8]
        ld t13, 8(b3l), l[9]
        ld t14, 8(b2l), l[10]
        mul t15, t14, t13
        sd t15, 8(b1l), s[11]
        addi t16, b3l, 32
        addi t17, b1l, 32
        addi t18, b2l, 32
        read t0, r3
        addi t19, t0, −4
        write r4, t16
        write r5, t17
        write r6, t18
        write r3, t19
        tnez.bf t20, t19, .LBB0_5
```

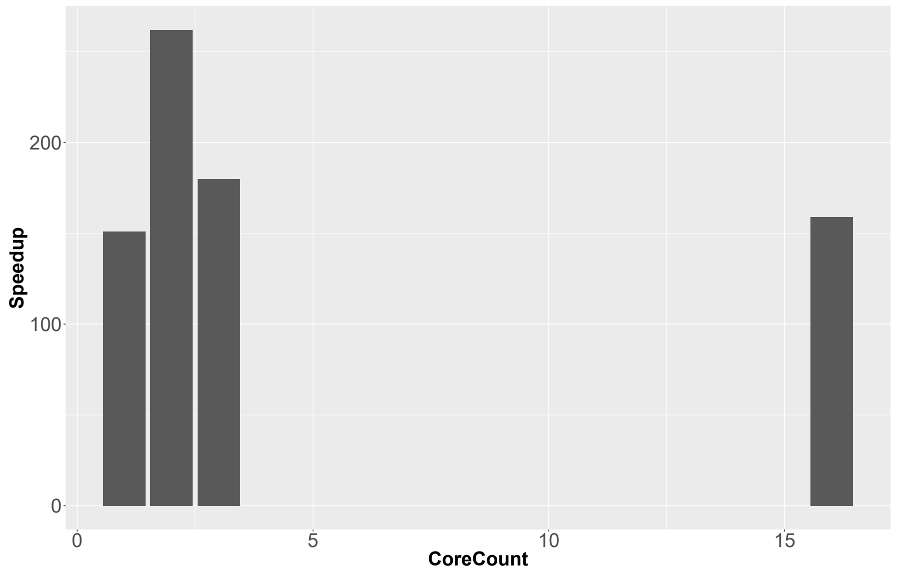

Figure 4: Cycle count for executing block given a core fusion count

The average cycle count for 4 different core counts (1,2,3,16) can be seen in Figure 4. Why does the 2 Core Count have a much higher cycle count ? Cache line dependencies. By tracing the execution on a single core, we can see that there will be a cache miss on L[9]/L[10] (issued in parallel). Since lines are 32 bytes, we will have a cache miss every L[9]/L[10] of LBB0_5. When two cores are executing LBB0_5, the second core's first load L[0] shares the same cache line as the L[9] load from the previous block.

Tracing the execution of this benchmark on 2 cores fused , shows that there exists a 120 cycle difference between the two cores fetching headers (so Core 1 fetches at cycle 0, and Core 1 fetches at cycle 120). This may have been caused by cores having fetched blocks of different size at some point, so they're not 100% in sync.

So let's imagine that Core 1 has fetched LBB0_5 at cycle 0, sends a prediction for C1 which will fetch it at cycle 120. By cycle 120, let's imagine Core 1 will have made a cache request for L[9] 20 cycles prior. This means that when Core 2 makes a request for L[0] it will have to wait 100 cycles for the cache to return the request. It will also have a cache miss on L[9]/L[10], so we have 120 cache miss + 100 cache wait + the rest of the block which equates to 259 cycles, compared to the 150 cycle count of executing the block on a single core.

Why does this not happen when we fuse more than 2 cores ? This is due to the fact that by fusing more cores we can pre-empt the cache miss using the extra cores. As we add more cores to the core composition, we can increase the number of cores executing l[0], which would prefetch the line for blocks about to execute l[9].

4.1 What does this mean?

What does it matter then ? Just fuse more than 2 cores. This should be fine in a situation such as the one previously described, but if we are interested in thread/core-fusion partitioning we must ensure that fusing 2 cores actually gets us anything, and it can, as we have just seen, easily go wrong. There is a potentially compiler fix in which you re-order the loads (as they are all independent), but of course this is only a very simple example.

5 Memory barriers render fusion useless

First off here's a refresher on memory barriers: there are two types

- Load Acquire: When a load acquire block is fetched, all following blocks cannot execute their memory instructions until the load-acquire block has fully executed all its loads (including returning data from cache).
- Store Release: If a block is a store release block then it must wait for all previous blocks to have finished executing their memory instructions before the store release block may execute its memory operations.

Why is this a problem ? Easy, let's simply imagine a core fusion with 2 cores, if the master core fetches a load-acquire block, then the slave core will have to wait for the master core to have executed all its loads before it can execute its memory operations. This completely serialises the work in a core composition; imagine having 8 cores where Core 1 fetches a load-acquire block and then hits a bunch of cache misses, then every other core is going to have to wait until Core 1 has resolved its cache misses before being able to do anything. The same will happen with store releases, if we have 4 cores in a core fusion, and core 2 fetches a store-release block, then the blocks fetched by Core 3 and 4 will have to wait on all memory operations found in Core 1 to have executed.

This essentially means that if we fuse cores in a multithreaded application that requires some form of memory barrier synchronisation, then the cores will be underutilised. These barriers are often found when trying to acquire locks, or even when modifying atomic operations as they ensure that threads share the same vision of memory at that same point.

To prove my point, when refering back to this listing

```
#pragma unroll 4
    for (int i = 0; i  <100000; i++)
    {
        c[i] = a[i] * b[i];
    }
```

We previously could get up to 14.6x speedup on 16 cores; if I set array C to be atomic, this will actually make it slower to execute on a core fusion, in fact running it on a single core will be 1.18x faster.

5.1 What does this mean?

We were interested in improving the performance of multithreaded applications that potentially have to communicate (graph algos). As we intend on doing a mix of core/thread pairings, this means we still must use atomic operations to ensure the correct execution of the application. If the main kernel of an algorithm is dependent on atomic operations, then core-fusion **will be useless**.